Erosion Risk Modelling: An Improved Screening Tool for Urban Watershed Management

by

Hettige Dona Thiruni Dulara Thirimanne

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Applied Science

in

Civil Engineering (Water)

Waterloo, Ontario, Canada, 2024

© Hettige Dona Thiruni Dulara Thirimanne 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Urbanization alters hydrological responses by increasing impervious surfaces, leading to elevated runoff, altered streamflow regimes, and heightened flood risks (Paul & Meyer, 2001; Walsh et al., 2012). The impact of land-use changes is a crucial consideration for urban watershed management (Bochis-Micu & Pitt, 2005; Walsh et al., 2012). SPINpy 2 is a screening tool that utilizes digital elevation model (DEM)-based methods of stream power mapping from Vocal Ferencevic and Ashmore (2012) to integrate land-use data into its modelling framework.

This study presents the development of two of SPINpy 2's Land Use (LU) based analyses: i) the No Stormwater Management (NSM) Scenario and ii) the Engineered Stormwater Management Pond (ESM) Scenario. Incorporating Nature-based Solutions (NbS), such as stormwater management ponds, into SPINpy 2 can model methods to alleviate the adverse effects of urbanization by promoting infiltration and stabilizing stream banks. This feature is particularly valuable for urban watersheds at high erosion risk, where NbS can help reduce the effects of impervious surfaces, lower flood risks, and stabilize channels. SPINpy 2 facilitates the modelling of NbS, assessing their effects on stream power, discharge, and erosion sensitivity and providing a decision-support tool for urban watershed managers. It helps evaluate the long-term benefits of NbS in reducing runoff and enhancing ecosystem resilience. By modelling the effects of reducing peak flows on erosion risk, SPINpy 2 simulates how stormwater management measures can mitigate erosion and offers insights into effective strategies for enhancing channel stability.

The model was applied to urbanized watersheds such as Cooksville Creek to assess its utility in high-risk environments. The simulation results provide insights into the potential of NbS to reduce flood risks and improve channel stability. The application of SPINpy 2 demonstrated that incorporating NbS significantly mitigates the impacts of urbanization. Comparisons between scenarios with and without NbS interventions highlighted the importance of infiltration-based solutions in stabilizing stream channels and reducing sediment transport. SPINpy 2 also provided spatially explicit maps showing locations of high erosion risk and areas where NbS would be most effective.

The findings underscore the potential of SPINpy 2 as a decision-support tool for urban watershed managers. By simulating the impacts of land-use changes and NbS interventions, SPINpy 2 offers a proactive approach to addressing hydrological and geomorphological challenges posed by

urbanization. The ability to model diverse NbS scenarios enhances the tool's applicability in high-risk watersheds, such as Cooksville Creek, where impervious surfaces dominate and flood risks are heightened. The results demonstrate that NbS can substantially reduce runoff and stabilize channels, promoting ecosystem resilience and sustainable development.

Overall, SPINpy 2 serves as a screening tool for decision-makers, enabling them to simulate and evaluate the impacts of land-use changes and NbS interventions, promoting sustainable development and environmental stewardship in urban environments. Its comprehensive approach allows watershed managers to tackle the unique challenges posed by urbanization and supports the development of cost-effective and environmentally sound infrastructure and policies. This proactive, integrative approach positions SPINpy 2 as a key resource for managing urban watersheds.

Acknowledgements

This thesis would not have been possible without the support and encouragement of many people, and I am extremely grateful for their help along this journey.

First and foremost, I would like to thank my supervisor, Dr. Bruce MacVicar, for his guidance, patience, and the wealth of knowledge he shared throughout this research process. Their unwavering support and constructive feedback pushed me to reach new academic and personal growth levels, for which I am deeply appreciative. I would also like to thank Dr. Bryan Tolson for guiding me during the calibration and development of the discharge equations. I am grateful to Kimisha Ghunowa, Christine Zimmer, and Emma Haug-Kindellan from Credit Valley Conservation (CVC) Authority for providing the field data and resources essential to my analysis and their insights, collaboration, and interest in my work. I would also like to thank Ivana Vouk and Sean Fergusen from the National Research Council Oceans, Coastal, and River Engineering (NRC-OCRE) Research Centre for their collaboration and insights throughout the project.

I want to extend my gratitude to the Writing Center staff, whose assistance and insightful feedback were invaluable in helping me refine my writing skills. Their support enabled me to communicate my ideas more clearly and effectively. To my colleagues and friends in the River Hydraulics Research Group (RHRG), thank you for fostering a collaborative and inspiring environment. Your shared enthusiasm for research, willingness to discuss new ideas, and friendship have made this journey more productive and incredibly rewarding. I owe a debt of gratitude to my family, especially my parents, sister and my partner, who supported me unconditionally and believed in me every step of the way. Their love and encouragement were my foundation, grounding me through the ups and downs of this process. Lastly, I would like to thank the funding bodies, including CVC and NRC-OCRE, who provided the financial support necessary to pursue this research and present it at conferences, helping me connect with a broader academic community. This project was partially funded by National Research Council's Climate Resilient Built Environment Initiative, in support of delivering the Government of Canada's Adaptation Action Plan, and towards achieving commitments under the National Adaptation Strategy

This thesis is dedicated to all who have contributed in big and small ways to this work. Thank you for being part of my journey.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements v

List of Figures ix

List of Tables x

List of Abbreviations xi

Chapter 1 Introduction 1

1.1 Motivation 1

1.2 Objectives and Scope of the Thesis 2

1.3 Structure of the Thesis 4

1.4 Project Background & Partnerships 4

Chapter 2 Literature Review 6

2.1 Urbanization Effects on Urban Hydrology 9

2.1.1 Hydrology 9

2.1.2 Erosion and Sediment Transport 11

2.1.3 Channel Morphology 13

2.1.4 Urbanization 16

2.1.5 Nature-Based Solutions (NbS) for Watershed Management and Erosion Reduction 18

2.2 Urban Watershed Management 21

2.3 Review of Existing Tools 22

2.3.1 Stream Assessments to Quantify Morphologic and Hydrologic Changes 22

2.3.2 Existing Tools that Model and Quantify the Effects of NbS 24

2.3.3 Review of Existing SPIN Versions 28

2.4 Research Gaps and Objectives 31
Chapter 3 SPINpy 2 Scenarios & Study Areas 33
3.1 Data Used for SPINpy 2 Scenario Calibration 34
3.1.1 Discharge Equation 34
3.2 SPINpy 2 Application to Several Case Studies within the CRW 37
3.2.1 NSM Single Watershed (SW) 37
3.2.2 ESM Single Watershed (SW) 48
3.2.3 NSM Multiple Watersheds within a Boundary (MW) 50
3.2.4 Cooksville Creek Future Scenario ESM Single Watershed Analysis 53
3.3 Conclusion 54
Chapter 4 Code Development 55
4.1 Limitations 1-3: Land Use Analysis Revisions 57
4.1.1 Limitations 1 & 2: NSM Land Use Scenario Development 57
4.1.2 Limitation 3: ESM Land Use Scenario Development 57
4.2 Limitation 4: Modelled Scenario Revisions 59
4.3 Limitation 5: Multiple Watersheds within a Boundary Analysis 59
4.4 Limitation 6: Equation Calibration and Validation 60
4.4.1 Calibration Data Preparation 60
4.4.2 SPINpy 2 NSM LU Urban Discharge 63
4.4.3 SPINpy 2 ESM LU Urban Discharge 66
4.5 Conclusion 68
Chapter 5 Results of Applying SPINpy 2 to Cooksville Creek in the CVC Watershed 71
Chapter 6 Discussion and Conclusions 77
6.1 SPINpy 2 Application 78

6.2 Tool for Urban Watershed Management 79
6.3 Limitations of this Study/Future Research 80
6.4 Conclusions 81
References 82
Appendix A Urban Width and Discharge Methods 96
Appendix B Results Figure Sets 112
Appendix C SPINpy 2 Instruction Manual 129
Appendix D SPINpy 2 Code 145

List of Figures

Figure 2-1: Connectivity of themes discussed in this literature review..... 8
Figure 2-2: Depiction of the additive nature of a unit hydrograph. Retrieved from CUHP User Manual, 2014..... 11
Figure 2-3: Timeline of land use changes and channel conditions. Retrieved from Charlton (2007).. 17
Figure 2-4: SPINarc Tools..... 28
Figure 3-1: Credit River Watershed (CRW) Land Use Classification. 33
Figure 3-2: Depiction of the watersheds used to calibrate the NSM and ESM discharge equation and the gauge stations used to validate the ESM discharge equation. 36
Figure 3-3: Subwatersheds Used for the Single No Stormwater Management Analysis 38
Figure 3-4: Cooksville Creek Land Use Classification 42
Figure 3-5: Land Use Scenario Inputs..... 43
Figure 3-6: Modelled Analysis Inputs..... 44
Figure 3-7: Levi Creek Watershed Characteristics..... 46
Figure 3-8: Land Use Scenario Inputs..... 47
Figure 3-9: Modelled Analysis Inputs A) 1 m Levi Creek digital elevation model B) Detailed hydrologic model stations..... 48
Figure 3-10: Levi Creek Watershed Basin and Stormwater Management Ponds 49
Figure 3-11: Multiple Boundary Analysis..... 51
Figure 3-12: Multiple Watershed Land Use Analysis with Political Boundary..... 52
Figure 3-13: Multiple Watershed Modelled Analysis with Political Boundary..... 52
Figure 3-14: Cooksville Creek Future Pond Addition Scenario..... 54
Figure 4-1: NSM Drainage Area and Imperviousness by Subwatershed..... 61
Figure 4-2: ESM Drainage Area and Imperviousness by Subwatershed 62
Figure 4-3: SPINpy 2 Possible Scenarios..... 69
Figure 5-1: Cooksville Creek Rural and Urban NSM and ESM Discharge Outputs 72
Figure 5-2: Cooksville Creek NSM and ESM SP Outputs..... 73
Figure 5-3: Cooksville Creek NSM and ESM SSP Outputs 74
Figure 5-4: Cooksville Creek NSM and ESM SSPR Outputs..... 75

List of Tables

Table 1-1: SPIN versions, features and input requirements 3
Table 2-1: Examples of Causes and Effects of USS 7
Table 2-2: Empirical Discharge Equations in Literature 10
Table 2-3: Width Equations in Literature 15
Table 2-4: NbS approaches and its erosion reduction mechanisms and environmental benefits 20
Table 2-5: Overview of Existing Tools that Model and Quantify the Effects of NbS 26
Table 2-6: Research Gaps And Objectives 32
Table 3-1: Subwatersheds Used in the Calibration of the NSM and ESM Discharge Equations 35
Table 3-2: Application of SPINpy 2 to Study Areas 37
Table 3-3: Impervious percent assigned to each land use type necessary for the LU Analysis inputs. 39
Table 4-1: Summary of Limitations Addressed in SPINpy 2 56
Table 4-2: Equations Considered for NSM Discharge Equation Calibration 63
Table 4-3: Discharge Equation Calibration Results 64
Table 4-4: Summary of Statistics for the Calibrated Equation Using Training Dataset 65
Table 4-5: Equations Considered for Calibration 66
Table 4-6: Discharge Equation Calibration Models Considered 67
Table 4-7: Summary of Statistics for the Calibrated Equation (Training Data) 67
Table 4-8: SPINarc, SPINpy, and SPINpy 2 program inputs based on type of analysis 70

List of Abbreviations

Abbreviation	Description
CVC	Credit Valley Conservation Authority
SPIN	Stream Power Index Network
SPINarc	SPIN (ArcGIS Toolbox) developed by Ghunowa et al. is used for stream power analysis (Ghunowa et al., 2021). It allows users to analyze stream power using HEC-RAS and Land Use data input.
RROIT	Risk and Return on Investment Tool used to evaluate the monetary value associated with the erosion of rivers and streams.
SPINpy	Version of the SPINarc Toolbox within RROIT and written entirely in Python. This script allows the user to perform stream power analyses using HEC-RAS data input. However, does not allow the user to perform stream power analyses using Land Use data.
SPINarc_width	A version of SPINarc updated by Victoria Barlow and Peter Ashmore at the University of Western Ontario to account for width changes in response to urbanization and increase of discharge (Barlow, n.d.).
SPINarc_speedy	A version of SPINpy was optimized for speed in partnership with partners from the Computer Systems Group at the University of Waterloo.
SPINpy 2	The revised version of SPINpy allows the user to perform analyses with land use data and HEC-RAS data. This version retains the existing features of SPINpy and introduces a new method.
LU Analysis	Refers to SPINpy 2's stream power analysis that uses land use data inputs.
Modelled Analysis	Refers to SPINpy's stream power analysis, which uses detailed hydrology model inputs.
NSM	No Stormwater Management Scenario (NSM) in SPINpy 2
ESM	Engineered Stormwater Management Pond Scenario (ESM) in SPINpy 2

Chapter 1
Introduction

1.1 Motivation

As urban areas expand to accommodate growing populations, anthropogenic activity leads to significant land use changes that contribute to adverse climate change effects. While urban development is important, the challenge lies in ensuring sustainable development, meaning there is a balance between urbanization's social and economic benefits and its adverse environmental impact. A consequence of rapid urban expansion is the degradation of river systems, which intensifies erosion processes. Rivers are complex geomorphic systems that create a network within a catchment controlled by hydrology and sediment regimes (Vocal Ferencevic & Ashmore, 2012). For this reason, practitioners and watershed managers rely on watershed-scale erosion assessment tools to make planning decisions.

Channel assessment tools often use stream power as a metric for predicting channel morphological changes and sediment transport capacity. Stream power measures the energy expended per unit area of the riverbed and is directly related to the erosive capacity of water flow (Bagnold, 1968, 1980; Bledsoe, 2002; Charlton, 2007; Ghunowa et al., 2021; Marcinkowski et al., 2022; Parker et al., 2015; Rhoads, 1987). Stream power metrics are necessary for watershed managers to prioritize management efforts and allocate existing fiscal budgets effectively, enabling them to identify areas of high erosion risk. Since field measurements of erosion across all stream segments within a watershed is often impractical, there is a growing need for triaging tools to help focus on the most significant erosion-prone areas. This creates a demand for reliable, validated tools capable of conducting stream power analysis at a broad scale to evaluate the erosion sensitivity of urbanized watersheds. For example, a channel assessment tool allows watershed managers to leverage such validated models to conduct channel assessments efficiently. As such, the tool enables watershed managers and practitioners to analyze large spatial datasets, forecast erosion risks, and make data-driven decisions without requiring extensive field surveys.

Notable channel assessment tools used for erosion assessments include the Topographic Analysis Kit (TAK) for TopoToolbox (Forte & Whipple, 2019; Schwanghart & Kuhn, 2010), Fluvial Corridor (Roux et al., 2015) and Sediment Transport: Reach Equilibrium Assessment Method (ST: REAM)

(Parker et al., 2015). TAK for TopoToolbox aids in topographic and hydrologic analysis, while Fluvial Corridor and ST: REAM are suitable for sediment transport and river morphology evaluations. Although these tools exist, they do not allow users to conduct rapid analyses with easily accessible land use data. The Stream Power Index for Networks (SPINarc), built as an ArcGIS Toolbox, was introduced to address the limitation of rapid assessments using minimal and readily available user inputs, offering a more comprehensive assessment of urban-induced changes in stream power metrics (Ghunowa et al., 2021). SPINarc was created to support informed watershed management and planning decisions. Its abilities include stream power analyses using two data input methods. The first data input method is modelled data from a detailed hydrologic model output. The second input type is land use data, which allows stream power analyses to be conducted in data-scarce environments or as a preliminary desktop analysis before further detailed modelling.

Despite the availability of various channel assessment tools, current models have limitations in reliably representing the complexities of urban environments, particularly in capturing how urbanization alters channel width and discharge patterns. Many tools also fail to account for the benefits of Nature-Based Solutions (NbS), such as green infrastructure and vegetative buffers which are essential for providing infiltration opportunities for surface runoff. This limitation in existing tools is significant because NbS can offer a sustainable erosion management approach that alleviates further erosion and detriment to existing streams.

Current algorithms like those in SPINarc face limitations in modelling NbS interventions, as they cannot quantify the impacts of NbS on stream power dynamics without relying on detailed hydrologic inputs. These limitations highlight the need for validated rapid assessment tools to address these challenges and provide accurate predictions of erosion risks. To address the gap in rapid channel assessment tools, developing models that integrate NbS into channel erosion risk assessments is essential. The development of such tools would improve prediction accuracy and allow watershed managers to evaluate the effectiveness of NbS in mitigating erosion risks in urban watersheds at a watershed level.

1.2 Objectives and Scope of the Thesis

The goal of this thesis is to advance the understanding and management of erosion in urban systems with an updated SPINarc tool, called SPINpy 2. SPINpy 2 is intended to serve as a high-level watershed scale erosion screening tool for watershed managers to identify locations experiencing

adverse conditions occurring over a long period of time. This approach enables practitioners to focus investments and detailed field studies on the identified areas, promoting efficient resource allocation. This new tool updates a beta Python version of SPINarc called SPINpy (Table 1-1). Specific objectives of this thesis are to i) consolidate different versions of SPIN that have been created to address specific management questions, ii) improve the speed, accuracy, and flexibility of the stream power analysis, and iii) calibrate and validate empirical equations within SPIN that allow the estimation of stream power metrics from land use data (called the "Land-Use (LU)" Analysis in Table 1-1). The "Modelled" Analysis in Table 1-1 refers to the existing SPINpy capability of using flow and channel data from modelled hydrology scenarios.

Table 1-1: SPIN versions, features and input requirements

Features	SPINarc	SPINpy	SPINpy 2
Rural Stream Power Scenario • DEM • Empirical Equation	x	x	x
Urban Stream Power Scenario			
Urban Modelled Analysis: • Modelled Discharge • Modelled Width	x	x	x
Urban Land Use Analysis: • DEM • Land Use Layer • Impervious % • Empirical Equations	x	-	x
Financial Assessment Integration (RROIT)	-	x	x
Erosion Management Scenario (RROIT, Green Infrastructure)	-	-	x

SPINpy 2 will consolidate the existing version of SPINarc while addressing algorithm limitations in SPINarc. The limitations of SPINarc's Land Use Analysis include an empirical urban discharge equation that has not been validated for the study region. Additionally, the urban width equation was identical to the rural scenario, resulting in unrealistic urban width predictions as it failed to consider channel morphology due to urbanization. SPINpy 2 improves the accuracy of the stream power predictions for the land use data-driven scenario by revising the channel width and urban discharge equations. Finally, SPINarc lacks the ability to assess erosion risk within a user-specified boundary.

This added functionality will be added to SPINpy 2, allowing watershed managers to run their analyses across boundaries such as political, municipal, or neighbourhood boundaries

SPINpy 2's flexibility will be enhanced by adding methods to account for urban development scenarios and NbS strategies. Specifically, these improvements will expand the tool's capabilities by incorporating the ability to conduct rapid erosion risk assessments that consider stormwater management ponds. SPINpy 2's improvements will assess the impacts of NbS on erosion risk within urban watersheds, including the influence of stormwater management (SWM) systems. SPINpy 2 will be applied to watersheds within the Credit Valley Conservation (CVC) jurisdiction in Ontario, Canada, explicitly focusing on the City of Mississauga. The model's accuracy will be validated using key metrics such as channel width, discharge, and erosion rates, ensuring that SPINpy 2 effectively captures the influence of urbanization and green infrastructure on erosion sensitivity in this area.

1.3 Structure of the Thesis

This thesis is structured as follows:

- Chapter 2: A literature review on watershed-scale erosion and stream power assessment tools.
- Chapter 3: Description of study areas used to test SPINpy 2.
- Chapter 4: Description of development, enhancement, and validation methods.
- Chapter 5: Application and results.
- Chapter 6: Summary of key findings and conclusions.

1.4 Project Background & Partnerships

This research takes place within the Credit River Watershed (CRW), part of the Credit Valley Conservation Authority (CVC) jurisdiction. The CRW consists of the 90 km long Credit River flowing into Lake Ontario and consists of 23 subwatersheds. Passing through nine municipalities, the Credit River is a vital natural asset that provides a vibrant and biodiverse natural area for citizens to enjoy. The CRW is primarily comprised of natural and agricultural land use types (36.3% and 36.0% respectively). To sustain the watershed's social and ecological benefits, watershed managers must safeguard, restore, and effectively manage the watershed. Researchers are developing tools such as SPINpy 2, which can assess the erosion risk of a river using a stream power approach to measure and maintain watershed sustainability.

This research was supported through an ongoing collaboration with the CVC and the National Research Council Oceans, Coastal, and River Engineering (NRC-OCRE) Research Centre. The tool previously developed as part of a prior research agreement is SPINarc - a GIS-based toolbox developed in collaboration with the Toronto and Region Conservation Authority (TRCA) and CVC. SPINarc has been integrated into CVC's Risk and Return on Investment Tool (RROIT) to estimate erosion damage costs and inform updates to Official Watershed Plans for two watersheds in the TRCA jurisdiction.

The current research agreement continues the previous work completed with the NRC and CVC by addressing the limitations of SPINarc. Partial funding for this research was provided by NRC's Climate Resilient Built Environment Initiative in support of delivering the Government of Canada's Adaptation Action Plan and towards achieving commitments under the National Adaptation Strategy. These initiatives, including the SPINpy 2 Tool, aim to support climate-informed decision-making for designing, retrofitting, and upgrading buildings and infrastructure to improve climate resilience.

Chapter 2
Literature Review

As the global population increases, the demand for essential resources such as housing, education, and transportation also increases. Satisfying these demands requires the construction of housing, transportation systems, and recreational and educational facilities to accommodate the growing population. These activities, collectively referred to as 'urbanization,' often disrupt the natural balance by modifying landscapes and erecting infrastructure to cater to expanding urban populations (Alberti, 2005; Bernhardt & Palmer, 2007; Decker et al., 2002; Poff et al., 1997; Wohl et al., 2005). Modifying the landscape by converting vegetated land cover to impermeable surfaces disturbs the hydrological cycle, resulting in heightened surface runoff and diminished infiltration. These changes can lead to erosion and instability of channels (Booth et al., 2002; Paul & Meyer, 2001; E. Tillinghast et al., 2012; E. D. Tillinghast et al., 2011; Walsh et al., 2012; Winston et al., 2016; Woods Ballard & Construction Industry Research and Information Association., 2015).

Urbanization adversely affects the environment and humans, resulting in the loss of natural habitats, declining biodiversity, air and water contamination, and increased susceptibility to natural disasters such as flooding and erosion. Characteristics associated with stream deterioration due to urbanization are described as Urban Stream Syndrome (USS). USS is identified by a range of symptoms (Table 2-1), including changes in flow regimes, elevated nutrient concentrations, and alterations in channel morphology.

Table 2-1: Examples of Causes and Effects of USS (Booth et al., 2016; Meyer et al., 2005; Paul & Meyer, 2001; Vietz et al., 2016; Walsh et al., 2005, 2012)

Field	Effects	Causes
Hydrology	Flashy hydrograph (increased erosive flows, reduced baseflows)	Land Use Changes (Imperviousness) Transport of runoff using piped systems
Water Chemistry	Elevated nutrient concentrations and contaminants	Poor management of catchment nutrient retention
Channel Morphology	Altered channel morphology (increased depth, width) Reduced channel complexity	Small, frequent high-flow events
Biology	Reduced biotic richness	Increased biomass from small frequent storm events that are rich in phosphorous (P)
Macroinvertebrates	Reduced abundance of native species and dominance of tolerant species	Catchment urbanization

In response to USS, NbS are employed to mitigate the USS, reducing erosion and flooding while delivering social and environmental advantages. NbS plays a crucial role in ameliorating adverse effects, enhancing urban areas, and fostering the harmonious coexistence of urban development and nature (Cohen-Shacham et al., 2019; Fletcher et al., 2013; Lafortezza et al., 2018).

The rapid expansion of urban areas and its adverse environmental consequences, such as erosion, have created a need to prioritize watershed preservation and safeguard the well-being of residents. Setting up monitoring or field visits for every river reach is not always feasible, and desktop analyses that can be used to predict erosion-prone sections of the river network can be useful for watershed managers. Watershed managers rely on tools to regulate the safety and sustainability of rivers. However, existing tools often fall short of accurately evaluating erosion risks due to limitations in their equations, algorithms, and adaptability to specific watershed boundaries.

Additionally, the effective implementation of NbS depends on accurately predicting NbS performance within the urban landscape. Due to computationally intensive algorithms, existing erosion risk assessment tools, such as SPINarc, do not model the presence of NbS. These modelling limitations highlight the necessity for improved tools to effectively assist watershed managers in mitigating the impacts of urbanization.

The identified limitations represent a significant gap in stream power desktop analysis tools. Particularly inefficient erosion risk modelling for detailed DEMs, integrating sustainable stormwater management practices, such as NbS, and the effectiveness and accuracy of existing modelling tools to operate within diverse watershed contexts. The main objective of this literature review is to provide a review of the intersection between urbanization, hydrology, and erosion while providing practical solutions for watershed managers (Figure 2-1). Additionally, this literature review will assess the existing tools employed for watershed management, focusing on the SPINarc tool.

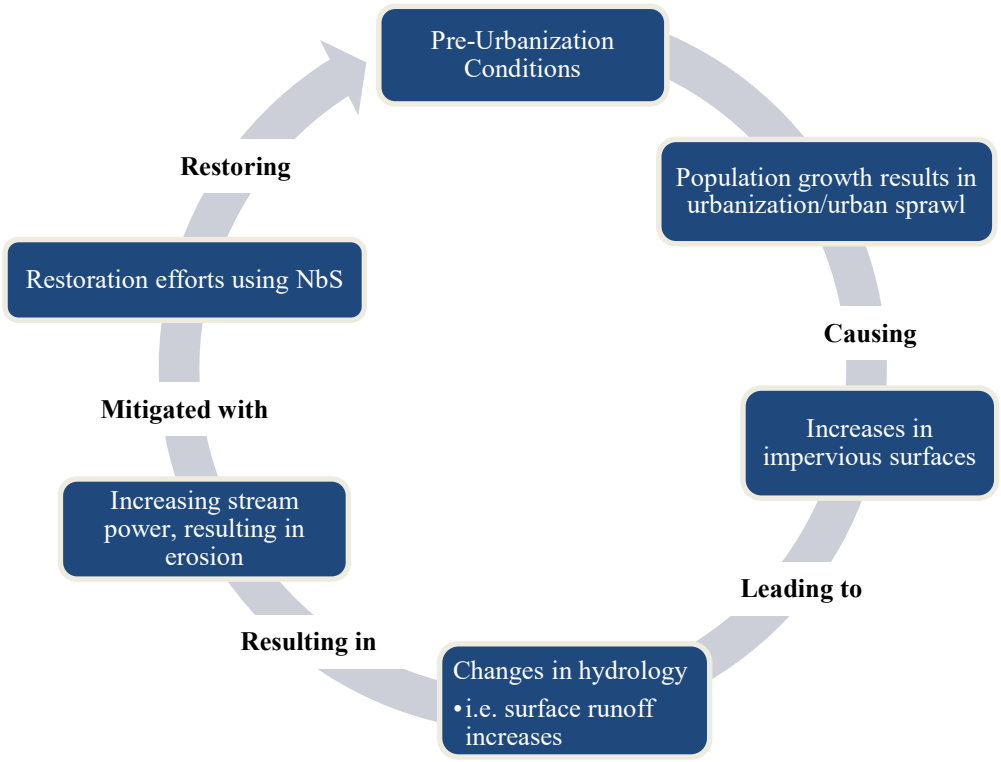


Figure 2-1: Connectivity of themes discussed in this literature review. These central themes were considered throughout the project, and the figure illustrates the conceptual flow of theories while developing SPINpy 2.

2.1 Urbanization Effects on Urban Hydrology

2.1.1 Hydrology

The hydrologic cycle is a fundamental concept that describes the continuous movement of water through the Earth's atmosphere, surface, and subsurface. Key processes in this cycle include precipitation, runoff, infiltration, and evapotranspiration. However, the expansion of urban areas due to human activities has resulted in impervious surfaces, such as roads and buildings, that disrupt these natural processes. As a result, the hydrological cycle is altered, with a significant increase in surface runoff and a decrease in infiltration and evapotranspiration (Cahill et al., 2004). This alteration has various consequences, including higher discharge volumes, increased flooding risks, and enhanced downstream erosion. It is worth noting that the 2-year discharge event is significant in shaping the channel's morphology, as it often represents the flow responsible for significant channel-shaping activities (Charlton, 2007).

Discharge is a fundamental parameter that measures the volume of water that flows through a cross-section of a stream or river over time. The 2-year bankfull discharge is notable as it represents the event that performs the most geomorphic work. It impacts sediment transport rates and erosion capacity (Bizzi & Lerner, 2015; Rhoads, 1987). Urbanization, with its associated increase in impervious surfaces, amplifies discharge levels. This increase in discharge raises stream power, exacerbates erosion risks, and affects sediment transport dynamics. Higher discharge can result in more frequent and severe flooding events, further altering natural flow patterns and contributing to the degradation of urban streams.

Accurately modelling discharge is a significant challenge in urban hydrology. In urban watersheds, increased peak discharges are strongly associated with the growing imperviousness of urban developments (Anderson, 1970; Bledsoe & Watson, 2001; Huang et al., 2008; Leopold, 1968). Impervious surfaces reduce the frictional resistance of overland flow, resulting in increased surface runoff volume and rate (Charlton, 2007). Empirical relationships have been developed to predict discharge in ungauged sites based on the impervious area, drainage area, and flood magnitude and frequency (Bledsoe & Watson, 2001; Harman et al., 2001). Traditional discharge equations are inadequate for urban contexts, leading to the development of alternative models and calibration techniques (Bledsoe & Watson, 2001; Cahill et al., 2004). Understanding discharge dynamics requires a comprehensive framework that considers changes in land cover and runoff. As

imperviousness increases, so does the runoff, necessitating revised approaches to effectively model hydrological impacts and erosion (Johnson & Hunt, 2020). Empirical discharge equations for predicting the 2-year discharge in rural and urban watersheds that are prominent in the literature are listed in Table 2-2.

Table 2-2: Empirical Discharge Equations in Literature

Equation	Study	Dependent On	Study Context
$Q=0.248(DA)^{0.91}$	(Phillips & Desloges, 2014)	DA(sq.km)	Rural watersheds in Southern Ontario
$Q=182(DA)^{0.706}$	(Bledsoe & Watson, 2001)	DA(sq.mi)	Rural watersheds in Alabama
$Q=150(DA)^{0.70}(IA)^{0.36}$	(Bledsoe & Watson, 2001)	DA(sq.mi), IA(%)	Urban watersheds in Alabama
$Q=0.64(DA)^{0.734}$	(Gringas et al., 1994)	DA(sq.km)	Ontario and Quebec
$Q=0.239(DA)^{0.946}$	(Gringas et al., 1994)	DA(sq.km)	Lake Ontario, Erie, and St. Clair
$Q=2.11(DA)^{0.70}(IA)^{0.30}$	(Ward and Trimble, 2004)	DA(sq.mi), IA(%)	Urban watersheds in Georgia
$Q=0.248(DA)^{0.91}(IA)^{0.30}$	(Ghunowa et al., 2021)	DA(sq.km), IA(%)	Urban Watersheds
$Q_{Total} = Q_{DCIA}+Q_{UIA+PA}$	(Deng et al., 2020)	DCIA, UIA, PA	Urban Watersheds

Note: DA is the drainage area, and IA is the impervious percent

The discharge equations in Table 2-2 consider local conditions, such as the relationship between imperviousness, perviousness, and drainage area. Phillips & Desloges (2014) and Bledsoe and Watson (2001) depict a predictive equation for rural watersheds, while Bledsoe and Watson (2001) and Ghunowa et al. (2021) propose urban discharge equations. Urban watersheds consider the changes in impervious areas (IA) and drainage areas (DA). Another method often considered is that runoff from numerous surface types in a catchment is additive and forms the storm hydrograph (Figure 2-2). Deng et al. (2020) display this discharge relationship by considering that the total discharge is the discharge from the directly connected impervious area (DCIA), the impervious area routed to the pervious area (UIA), and the discharge that occurs from the pervious area (PA). This process is used by the Colorado Urban Hydrograph Procedure (CUHP) developers for a given catchment (Figure 2-2). Where UIA is the impervious area connected to a pervious area, RPA is the receiving pervious area, which receives flows from the impervious area. DCIA is impervious areas directly connected to an underground stormwater sewer system, and SPA is pervious areas not receiving runoff from impervious areas. The hydrograph's additive nature can translate into the

discharge equation such that the discharge from the pervious and impervious surfaces is calculated individually with unique coefficients.

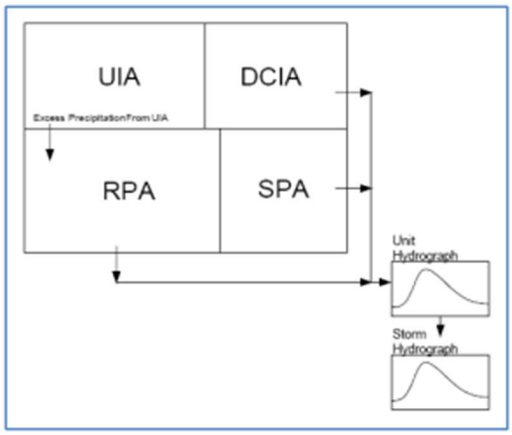


Figure 2-2: Depiction of the additive nature of a unit hydrograph. Retrieved from CUHP User Manual, 2014 (Urban Drainage and Flood Control District, 2014).

2.1.2 Erosion and Sediment Transport

Erosion is a natural process in which water carries soil particles from riverbeds and banks, shaping river channels over time (Julien, 1995). Various mechanisms contribute to erosion, including mass failure when banks are exceeded and fluvial entrainment, which is influenced by factors like soil properties, vegetation density, and channel geometry (Bizzi & Lerner, 2015; Charlton, 2007; Clark et al., 2008; Papangelakis et al., 2022; Phillips & Desloges, 2014). Effective erosion and sediment transport management involves land management techniques and restoration efforts to promote vegetation growth, restore hydrological regimes, and mitigate erosive forces (Parker et al., 2015). Understanding the interaction between discharge, stream power, and erosion is crucial for assessing and managing river systems, especially in urban areas.

Sediment transport is closely connected to erosion, transport, and deposition within a channel reach. These processes are influenced by sediment supply from upstream sources and local sediment erosion from the riverbed and banks. The size distribution of sediment is as essential as its volume, as the behaviour of sediment transport varies between coarse and fine materials. Fine particles, like clay, silt, and sand, are carried as suspended loads and can be transported over long distances, while coarser materials are transported closer to the riverbed as bedload (Charlton, 2007). Sediment supply

and transport fluctuate with changes in sediment supply from the drainage basin and channel network processes.

Human activities such as land use changes, construction, and deforestation intensify erosion and sediment transport processes, leading to significant environmental and engineering challenges. Urbanization, with its increase in impermeable surfaces, amplifies runoff and enhances the energy of flowing water, thereby accelerating erosion (Bledsoe, 2002). Deforestation further increases soil vulnerability to erosive forces, increasing sediment delivery to river channels and heightened flood peaks due to reduced infiltration rates (Charlton, 2007).

Total stream power (Equation 2-1), a crucial metric for understanding erosion and sediment transport, measures the energy expended per unit length of the riverbed and is directly related to the erosive capacity of water flow (Bagnold, 1968, 1980; Charlton, 2007; Ghunowa et al., 2021; Marcinkowski et al., 2022; Rhoads, 1987). Total stream power, which is crucial for predicting morphological changes and sediment transport capacity, is written as:

$$\Omega = \rho g Q S \quad \text{(Equation 2-1)}$$

where Ω is total stream power (W/m), ρ is the density of water (1000kg/m^3), g is the acceleration of gravity (m/s2), Q is discharge (m^3/s), and S is the channel slope (m/m) (Bledsoe, 2002; Parker et al., 2015). When total stream power exceeds the energy required to exceed flow resistance, the excess energy may lead to sediment transportation and bank erosion. If the sediment transport rate exceeds the amount of sediment supplied to the channel from upstream, channel degradation may occur.

Specific stream power (Equation 2-2) is the energy expended per unit area and is effective for estimating thresholds of sediment movement during rainfall events and assessing the sensitivity of river networks to erosion and deposition (Bizzi & Lerner, 2015; Bledsoe & Watson, 2001; Rhoads, 1987). Specific stream power can be written as:

$$\omega = \frac{\Omega}{w} \quad \text{(Equation 2-2)}$$

where ω is the specific stream power (W/m^2), and w is the width (m). Modelling variations in specific stream power provides valuable insights into sediment transport processes, allowing for the identification of potential discontinuities in sediment transport and serving as a robust assessment tool for understanding the cumulative impacts of hydrological regimes on river networks (Bledsoe, 2002; Bledsoe & Watson, 2001; Ferguson, 2005; Vocal Ferencevic & Ashmore, 2012). These concepts are

closely related to channel morphology, as stream power and discharge play integral roles in erosion modelling frameworks.

2.1.3 Channel Morphology

Channel morphology refers to the river channel form and relevant physical processes operating at different spatial and temporal scales (Charlton, 2007). The stability of channels depends on multiple variables, including discharge, sediment supply, channel size, and valley gradient. Changes in these variables, often driven by climate, topographic changes, or human activities, can destabilize a river, resulting in permanent changes to its shape and dynamics (Ashmore et al., 2023; Charlton, 2007). Urbanization significantly impacts channel morphology by modifying streamflow patterns, often leading to the enlargement of stream channels (Hammer, 1972). This enlargement occurs as channels adjust their geometries to restore equilibrium in response to changes in flow and sediment patterns (Charlton, 2007).

Geomorphologists recognize that urbanization transforms natural rivers by introducing artificial structures to stabilize networks and mitigate erosion, flooding, and infrastructure damage (Nelson et al., 2006). However, these structures' stability is often temporary; major disturbances, like extreme flooding or significant land-use changes, can overwhelm interventions, forcing rivers to either return to a prior state or adapt to a new equilibrium condition (Charlton, 2007). Channel size, primarily influenced by upstream discharge, typically increases downstream with the drainage area. While natural channel size typically increases downstream as the drainage area increases, urban development can disrupt this progression, influencing erosion rates and sediment transport patterns and impacting channel morphology and stability (Ashmore et al., 2023; Charlton, 2007).

The flow regime of a river plays a critical role in determining channel form and stability. It refers to the variations in flow over time and can be considered the "climate" of a river. It is characterized by seasonal variations, flood frequency, magnitude, and the frequency and duration of low flows (Charlton, 2007). The characteristics of the flow regime significantly shape channel morphology due to the influence of discharge on stream power, velocity, and bed shear stress. Of particular interest is the bankfull discharge, which represents the flow at which the channel is filled with water. It serves as a morphological threshold between within-bank and out-of-bank flows. In many alluvial channels, particularly in humid temperate regions, the bankfull discharge tends to occur with a frequency of one

to two years (Charlton, 2007). It is closely associated with the flow that transports the most significant amount of sediment (performs the most geomorphic work) (Charlton, 2007).

Traditional methods for modelling stream power involve measuring morphological variables such as slope, width, and discharge along cross-sections and extrapolating values between these sections (Winterbottom & Gilvear, 1997). However, these methods are often limited in spatial scope, time-consuming, and costly (Jordan & Fonstad, 2005). High-resolution digital elevation models (DEMs) have emerged as a valuable tool for overcoming these limitations, allowing for more efficient and cost-effective calculation of stream power over larger areas. The DEM approach combines spatial analysis of DEMs with hydraulic geometry equations to calculate discharge and width, and it has become a recognized assessment tool (Vocal Ferencevic & Ashmore, 2012).

To understand these processes, regime theory suggests that stream channels maintain a quasi-equilibrium with their flow regime, and urbanization often leads to channel enlargement to restore this balance (Leopold & Maddock, 1953). Accurately estimating channel width is crucial for modelling the effects of urbanization on stream morphology. Leopold and Maddock first proposed the empirical equation (Equation 2-4) using downstream hydraulic geometry (Charlton, 2007; Leopold & Maddock, 1953):

$$W = aQ_2{}^b \text{(Equation 2-3)}$$

where Q_2 is the 2-year discharge, b is a constant 0.5, and a varies for each river. Discharge varies over space and time, increasing downstream in response to precipitation. Hydraulic geometry describes how flow velocity, width, and depth change in response to these discharge variations (Charlton, 2007). The equation models a proportional relationship between discharge and channel width; when the discharge increases, the channel width will also increase (Charlton, 2007; Leopold & Maddock, 1953).

The width equations used in recent studies (Table 2-3) are often based on rural watersheds and do not model channel width changes under varying flow conditions, which is unsuitable for urban watersheds. However, Ashmore et al. (2023) have developed a relationship that considers stream power and urban-specific variables, which is sensitive to the effects of urbanization on channel widths (Ashmore et al., 2023; Bizzi & Lerner, 2015; Marcinkowski et al., 2022).

Table 2-3: Width Equations in Literature

Equation	Study	Dependent On	Study Locations
$W=4.631(Q_2)^{0.136}$	(Ashmore et al., 2023)	$Q_2(m^3/s)$	Rural Watershed
$W=1.16(DA)^{0.508}$	(Phillips & Desloges, 2014)	DA(sq.km)	Rural Watershed
$W = Q_2^{0.70}$	Reinfields et al. 2004	$Q_2(m^3/s)$	Rural Watershed
$W=2.69(DA)^{0.36}$	Vocal Ferencevic and Ashmore, (2012); Annable, (1996)	DA(sq.km)	Rural Watershed
$W=0.234(Q_2)^{0.533}$	(Ashmore et al., 2023)	$Q_2(m^3/s)$	Urban Watershed
$W=0.87(Q_2)^{0.559}D_{50}^{-0.445}$	(Ashmore,2001)	$Q_2(m^3/s)$, D_{50}	Urban Watershed

Reinfields et al. (2004) also explored the relationship between flow magnitude and channel width by focusing on the flow associated with a 2-year return period. While this equation offers insight into channel-forming flows in rural environments, it may not adequately capture the variability in flow regimes that urban channels experience.

Unlike the rural equations proposed by Phillip & Desloges (2014), Reinfields et al. (2004), and Vocal-Ferencevic & Ashmore (2012), which do not account for the sensitivity of channel width to urbanization, Ashmore et al. (2023) proposed an equation based on stream power that explicitly captures the effects of urbanization on channel width. As impervious surfaces increase due to anthropogenic changes, the stream power at the channel segment also increases, leading to wider channels. This sensitivity is crucial for modelling urban watersheds, where the hydrologic response to precipitation is more intense, leading to flashier flows and higher erosive potential.

Additionally, Ashmore et al. (2023) proposed a more detailed method for calculating channel width by incorporating particle size (D50). This approach recognizes that the relationship between flow and sediment transport is highly complex in urban environments, and particle size distribution is an essential factor influencing channel dimensions. The D50-based method reflects the sensitivity of channel morphology to sediment transport dynamics.

The examination of width equations demonstrates a progression from simplistic rural empirical models tomulti-variate approaches that consider the complexities introduced by urbanization. Continually refining these equations is essential for effectively modelling channel morphology and managing erosion and sediment transport in urbanized watersheds. Accurate predictions of channel

width are critical for designing erosion control measures and NbS, particularly in regions experiencing rapid urbanization and increased hydrologic stress.

2.1.4 Urbanization

Urbanization is defined as the concentration of the population and the transformation of society's economic, social, and cultural aspects (Hussain & Imitiyaz, 2018), exacerbates runoff and sediment yields, although these effects are often localized. This process involves the movement of people and activities to urban areas, leading to increased urban populations and the expansion of urban spaces (Hussain & Imitiyaz, 2018; Tisdale, 1941). Urban sprawl, a byproduct of urbanization, is characterized by low-density, car-dependent development on the outskirts of city centers and older suburbs, frequently encroaching on sensitive greenfields and agricultural land (Banai & DePriest, 2014; Bourne, 2001). Sensitivity to land use change varies across regions and depends on soil erodibility, climate, topography, and slope-channel coupling within the drainage basin. Deforestation to make way for urban environments initially increases sediment yields, which stabilizes over time (Wolman & Schick, 1967). However, Charlton (2007) and Wolman and Schick (1967) observe that urbanization of agricultural areas leads to a rapid increase in sediment yields during the construction phase, followed by a decline as construction ends and soil conservation measures are implemented (Figure 2-3).

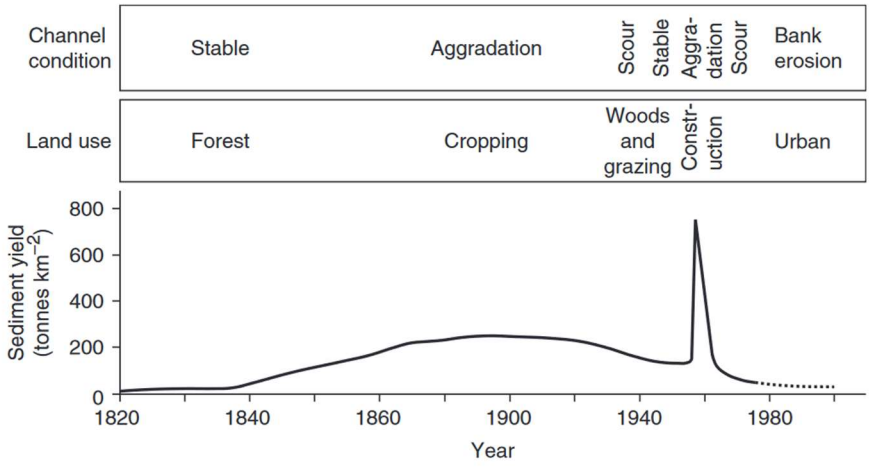


Figure 2-3: Timeline of land use changes and channel conditions. Retrieved from Charlton (2007). The pattern shown above is typical of expanding urban areas, where sediment yields surge during construction but later decrease, causing the flow regime to undergo permanent alteration. Impermeable surfaces, such as roads and parking lots, increase runoff, reduce infiltration and increase the magnitude of flood peaks, particularly for minor storm events. The alteration in flow dynamics can lead to channel enlargement and other significant geomorphological changes.

Ashmore et al. (2023) and Bevan et al. (2018) documented channel transformations due to urbanization in Toronto watersheds (Highland and Wilket Creek). Highland Creek exhibited progressive changes in flow regime and sediment dynamics over several decades, driven by impervious surfaces and stormwater infrastructure (Ashmore et al., 2023; Bevan et al., 2018). This increased both the magnitude and frequency of peak flows, elevating stream power by nearly ten times, which in turn catalyzed substantial channel widening and incision. These changes created a highly energetic system where frequent, high-intensity flows altered the geomorphology, transforming meandering channels into wider, straighter reaches. Similarly, Bevan et al. (2018) identified a comparable pattern in Wilket Creek, where urban development and impervious surfaces initially led to increased sediment loads. However, as the watershed matured, reduced sediment delivery and elevated discharge led to a steady channel enlargement and a shift from sediment deposition to erosion. The Wilket Creek study area exemplifies how urbanization-induced flow increases can amplify erosion, transforming channel width and depth, often necessitating intervention to stabilize banks and protect nearby infrastructure.

2.1.5 Nature-Based Solutions (NbS) for Watershed Management and Erosion Reduction

NbS integrates the goals of environmental conservation and sustainable development to address the degradation of natural resources and its impacts on biodiversity and human well-being (Alberti, 2005; Burns et al., 2012; Cohen-Shacham et al., 2019; Paul & Meyer, 2001; E. D. Tillinghast et al., 2011). NbS encompasses a range of ecosystem-based approaches, including engineered and green infrastructure interventions (such as the creation of ponds, wetlands, parks, and forests) designed to emulate natural processes within urban environments. These ecosystem-based approaches offer numerous benefits, such as restoring hydrologic balance, improving water quality, flood management, erosion reduction, enhanced biodiversity, social benefits, and urban watershed resilience (Charlton, 2007; Lafortezza et al., 2018; Paul & Meyer, 2001).

To ensure that NbS interventions meet required mitigation measures, modelling is crucial before their physical construction. By simulating the hydrologic, geomorphic, and ecological effects of NbS, models can help assess the capacity of NbS interventions to mitigate adverse urban impacts like increased runoff and erosion. NbS interventions, such as detention-based and infiltration-based stormwater control measures (SCMs), wetlands, forests, and parklands, mitigate erosion by employing natural processes to manage runoff and stabilize soils. While detention-based SCMs focus on reducing peak flow and confining flows within channels, infiltration-based SCMs like Low Impact Development (LID) and Sustainable Urban Drainage Systems (SUDS) address surface runoff at its source, enhancing infiltration and promoting groundwater recharge (Bhatt et al., 2019; Fletcher et al., 2015).

Vegetation along the banks is also critical in this dynamic, providing additional erosion resistance and contributing to channel stabilization (Charlton, 2007). Wetlands, forests, and parklands contribute by absorbing excess water, stabilizing soils, and intercepting rainfall, offering various ecosystem services, including flood control and water quality improvement (Fletcher et al., 2013; Kumar et al., 2021). By enhancing riparian vegetation, restoring natural channel forms, and incorporating green infrastructure, NbS can improve the resilience of urban streams against the pressures of increased runoff and sediment transport.

A central theme in NbS is the concept of water balance, which plays a crucial role in comprehending the impact of NbS interventions on hydrological processes. Water balance modelling is fundamental in urban settings because it assesses the interactions between precipitation, infiltration,

and runoff. Water balance models can be crucial for evaluating the effectiveness of NbS in restoring the water balance and managing runoff and erosion after urbanization (Mitchell et al., 2001; Xu et al., 2023). By simulating changes in hydrological processes and runoff dynamics, water balance models inform sustainable land management practices and policy decisions to mitigate the adverse effects of urbanization (Walsh et al., 2012; Xu et al., 2023).

Effective watershed management decisions to mitigate the adverse effects of urbanization use tools like SPINarc to incorporate hydrologic and geomorphic processes to predict erosion risk but overlook the effect of NbS in reducing peak flows in the watershed. Incorporating NbS components into such models could significantly improve their predictive capacity, especially in urbanized watersheds where vegetation cover, permeable surfaces, and restored floodplains can play a significant role in mitigating erosion (Bizzi & Lerner, 2015; Julien, 1995). The effectiveness of NbS in reducing erosion is well supported in case studies that demonstrate the ecological mechanisms that stabilize soils, manage stormwater runoff, and promote vegetation cover (Table 2-4). NbS interventions like reforestation, wetland restoration, and riparian buffer zones enhance ecosystem services and provide physical structures that reduce sediment transport and erosion (Cohen-Shacham et al., 2016, 2019). Deep-rooted vegetation stabilizes soil, particularly during high-flow events, by anchoring soil and reducing surface runoff (Charlton, 2007). Modelling these processes allows for a detailed evaluation of NbS interventions in various scenarios, enabling practitioners to predict their performance under different storm intensities and urban development pressures.

Through erosion and sediment transport models, NbS can be evaluated to ensure their effectiveness before physical implementation. These models integrate landscape resilience factors, such as biodiversity, soil structure, and hydrological cycles, to assess the full range of benefits provided by NbS. For example, reforestation models can simulate the long-term effects of root structure on soil stability, while wetland restoration models can track changes in water infiltration and sediment retention (Cheng et al., 2023). Such predictive modelling ensures that NbS interventions meet the required mitigation goals for reducing erosion and enhancing landscape resilience before being physically established.

Table 2-4: NbS approaches and its erosion reduction mechanisms and environmental benefits

NbS Approach	Erosion Reduction	Additional Benefits	References
Detention-Based SCMs	Trap sediments and attenuate flow through ponds and other detention measures. Reduce peak flow rates and confine flows within channels to decrease erosional events.	Flood control, water quality improvement, peak flow reduction	Bledsoe & Watson, 2001; Rodrigues et al., 2023; Tillinghast et al., 2011
Infiltration-Based SCMs (LID, SUDS)	Reduce surface runoff at the source by enhancing infiltration through bioretention systems, permeable pavements, and rain gardens.	Groundwater recharge, reduced flood risks, improved water quality	Bhatt et al., 2019; Fletcher et al., 2015; Lee et al., 2023
Wetlands	Stabilize soil through vegetation, slow water flow, and act as natural retention ponds to absorb excess water.	Sediment filtration, flood control, water quality improvement	Fletcher et al., 2013; Kumar et al., 2021; Suits et al., 2023
Forests	Intercepts rainfall, reduces surface runoff and stabilizes soils through deep-rooted vegetation. Urban forests mitigate stormwater runoff and enhance ecosystem stability.	Flood risk reduction, biodiversity conservation, ecosystem services	Li et al., 2020; Meyer et al., 2005; Olding et al., 2004
Parklands	Utilize Green Infrastructure (GI) elements such as permeable pavements and rain gardens to reduce runoff and promote infiltration.	Ecosystem services, improved urban environments, stormwater management	Brawley-Chesworth, 2023; Jefferson et al., 2017; Tong et al., 2022

While the mitigation effects of the NbS approaches described in Table 2-4 are beneficial at a larger watershed scale, sediment transport is complex, so challenges remain at site-specific limitations. These challenges can hinder the long-term effectiveness of NbS, mainly when applied in isolation. Integrating NbS into broader watershed-scale strategies, such as combining LID with detention-based SCMs, has shown promise in addressing these challenges (Burns et al., 2012). As cities expand, leveraging these solutions with advanced modelling tools like SPINarc is crucial for enhancing resilience and fostering a sustainable coexistence between built and natural environments (Cohen-Shacham et al., 2019; Kumar et al., 2021). Therefore, NbS provide a sustainable and practical approach to managing erosion and urban hydrology.

2.2 Urban Watershed Management

Urban watershed management is a field focused on addressing the impacts of urbanization on water resources and ecosystem health. Currently, urban watershed management practices include a range of strategies designed to mitigate the adverse effects of increased impervious surfaces and altered hydrological regimes. These strategies encompass the implementation of green infrastructure (GI) practices such as permeable pavements, green roofs, and bioretention systems, which are intended to manage stormwater at its source, reduce runoff, and improve water quality (Guswa et al., 2020; Qin et al., 2013). The importance of urban watershed management lies in its ability to enhance resilience to flooding, control erosion, and improve overall water quality in increasingly developed areas. Effective management relies on predictive modelling to maintain the watershed's ecological balance, preventing infrastructure damage and safeguarding public health in urban environments.

The detrimental consequences of erosion and flooding on urban populations, including infrastructure loss, property damage, and reduced water quality, have led watershed managers to adopt NbS. These solutions restore natural processes and enhance ecosystem resilience, offering additional benefits such as improved water quality, stormwater management, and recreational spaces for urban residents (Cohen-Shacham et al., 2019; Kumar et al., 2021). Integrating NbS into urban planning requires a comprehensive approach that combines various techniques to tackle the diverse challenges of urbanization. Adverse effects can be mitigated by restoring pre-urban water balance conditions, stabilizing soil, and improving hydrological processes (Cohen-Shacham et al., 2019; Fletcher et al., 2013). The successful implementation of NbS requires careful consideration of local conditions and ongoing monitoring to ensure long-term efficacy (Cohen-Shacham et al., 2019; Fletcher et al., 2013).

Applications of NbS practices are widespread, including the integration of site-scale LID techniques in new developments, retrofitting existing infrastructure with SCMs, and restoring natural hydrological processes through riparian buffer zones and wetlands (Bhatt et al., 2019; Fletcher et al., 2013). Watershed managers play a crucial role in mitigation efforts, utilizing tools such as hydrological models, GIS, and monitoring systems to assess and address watershed conditions. They are responsible for identifying areas needing intervention, developing and implementing strategies that balance urban development with environmental protection, and ensuring land use planning incorporates sustainable water management practices (Olding et al., 2004). The tools available to

watershed managers include simulation models for predicting hydrological impacts, decision support systems for evaluating management options, and field-based monitoring equipment for assessing the effectiveness of implemented measures (Giese et al., 2019).

Urban watershed management is vital for mitigating urbanization's effects on water systems. With the support of various tools and technologies, watershed managers can effectively identify areas adversely affected by urban development. These efforts are crucial for promoting sustainable urban environments and ensuring the long-term health of aquatic ecosystems.

2.3 Review of Existing Tools

Sections 2.1 and 2.2 discussed the sediment and flow regime theory that drives changes seen in urban environments and how watershed managers rely on tools to model regime changes for predictions of extreme environmental conditions such as erosion and flooding. Section 2.3 will discuss the algorithms and decision support tools that are currently available. It will be split into two sections: first, 2.3.1 will discuss tools that can quantify and model morphologic changes, and next, 2.3.2 will discuss tools that model NbS. This will facilitate a discussion of the existing tool, SPINarc, its limitations, and how they can be addressed.

2.3.1 Stream Assessments to Quantify Morphologic and Hydrologic Changes

Quantifying morphological and hydrological changes in stream systems is crucial for understanding how catchments evolve in response to natural processes and human interventions, such as urbanization. One of the most notable advancements in recent decades is the application of GIS-based approaches to assess stream power and associated morphological parameters. Like Rosgen's (1996) qualitative classification method, traditional stream assessments often lacked the precision and repeatability needed for broader scientific applications, particularly in urbanizing landscapes. The Vocal-Ferencevic and Ashmore (2012) method emerged in response to these limitations, offering a GIS-based, catchment-scale total stream power analysis. This method was designed to be more accessible, requiring fewer manual inputs and less technical expertise while producing robust, scientifically valid outputs. The key innovation was to use a Digital Elevation Models (DEM) based process to extract relevant morphological parameters such as slope, drainage area, flow direction, and flow accumulation. Then calculate stream power using empirical discharge and total stream power equations.

The Vocal-Ferencevic and Ashmore method integrates Digital Elevation Models (DEMs), which provide detailed representations of terrain morphology, to extract relevant parameters and automatically calculate stream power. Stream power is a key indicator of erosive potential and sediment transport capacity. Vocal-Ferencevic and Ashmore's method employs GIS to calculate unit stream power at various locations along the stream network. This method identifies areas prone to morphological changes, helping researchers and practitioners understand how urbanization or other land-use changes affect channel stability and sediment dynamics. DEMs are processed to fill voids and generate flow accumulation patterns using algorithms like D8 flow direction models (Tarboton et al., 1991). These hydrological algorithms allow for extracting critical metrics such as watershed boundaries, channel slopes, and flow pathways. Early applications of DEM analysis were limited by the resolution of the available data and the simplicity of the processing tools. However, as DEM resolution improved and processing algorithms became more sophisticated, Vocal-Ferencevic and Ashmore could apply these tools to calculate stream power at multiple spatial scales accurately.

The method combines raster processing techniques to extract hydrological metrics with minimal user intervention, making it practical and reliable for larger datasets. The reliance on GIS software allowed for streamlined analysis and offered a more nuanced understanding of channel conditions. While the Vocal-Ferencevic and Ashmore method requires d_{50} field measurements which can be a limiting factor for users without access to field data, the Vocal-Ferencevic and Ashmore method can be adapted to use general land use information. The adaptation of the Vocal-Ferencevic and Ashmore method provides a robust framework for initial stream power assessments where d_{50} field data is not readily available. Their 2012 approach was later adapted into automated tools like SPINarc (Ghunowa et al., 2021; Papangelakis et al., 2022), which builds upon the step-by-step manual calculations initially proposed by Vocal-Ferencevic and Ashmore. This development further increased the efficiency and accessibility of stream power analysis, making it possible to generate comprehensive assessments at the catchment scale with minimal user input.

SPINarc simplified the process, offering an efficient way to calculate stream power along the channel network for catchment scale applications with minimal user input. The tool integrates all outputs into a single shapefile, removing much of the complexity of manual data handling. The development of geospatial analysis software has significantly enhanced the ability to quantify stream power throughout river catchment networks, providing comprehensive assessments of stream dynamics and sediment transport processes. Tools like SPINarc are vital in understanding and

managing river systems, especially in urban environments. These tools utilize DEMs to characterize terrain morphology, extract watershed boundaries, and quantify channel morphology metrics, discharge, and stream power in urban channels.

Numerous spatial analysis tools, such as the Soil and Water Assessment Tool (SWAT) (Arnold et al., 1998; Arnold & Williams, 1987), EPIC (Williams & Arnold, 1997), Topographic Analysis Kit (TAK) for TopoToolbox (Forte & Whipple, 2019; Schwanghart & Kuhn, 2010), Fluvial Corridor (Roux et al., 2015), Sediment Transport: Reach Equilibrium Assessment Method (ST: REAM) (Parker et al., 2015), and the Stream Power Index for Networks (SPINarc) (Ghunowa et al., 2021). SPINarc is the focus of this project and was specifically developed using Python for ArcGIS to allow a rapid assessment of the impact of urbanization on stream power metrics. These tools utilize Digital Elevation Models (DEMs) to characterize terrain morphology, extract watershed boundaries and channel morphology metrics, and calculate discharge and stream power in urban channels. These tools and others are suitable for flood modelling and offer comprehensive solutions for riverine landscape characterization and morphological analysis at various scales. This analysis is important at a watershed scale, as it allows for identifying areas prone to morphological changes and channel instability (Ghunowa et al., 2021). Moreover, advancements in spatial analysis tools and high-resolution DEMs have enhanced the accuracy and efficiency of stream power calculations, making it an indispensable tool for studying river morphology and behaviour (Forte & Whipple, 2019).

These advancements underscore the evolution of stream assessment tools and the ongoing need for increasingly automated, user-friendly methods that balance accessibility and practical application. As stream power assessments evolve, tools like SPINarc offer a promising direction for improving morphological and hydrological assessments' efficiency, accuracy, and scalability in complex and urbanized environments.

2.3.2 Existing Tools that Model and Quantify the Effects of NbS

Several tools and models are available to quantify and model the effects of NbS, particularly in terms of hydrological processes, erosion control, and sustainable stormwater management. Many tools and models are available for modelling channel morphology and erosion in different environments, ranging from empirical equations to complex hydrological models. These tools vary in scope, complexity, and applicability depending on the type of NbS, the scale of intervention, and the

environmental setting. This section reviews existing tools and outlines their benefits and limitations in modelling NbS.

Models such as KINEROS2, BasinMaker and Lake-TopoCat exist to quantify the hydrological and morphological impacts of NbS (Han et al., 2023; Korgaonkar et al., 2020; Sikder et al., 2023) (Table 2-5). For instance, KINEROS2 effectively models runoff and erosion in small, urbanized watersheds with green infrastructure (GI) components such as rain gardens and permeable pavements. However, its complexity and data demands make it challenging for large-scale applications. Other models like BasinMaker, a GIS-based tool, are ideal for hydrological routing in large, lake-dominated watersheds, although they require significant customization and data for detailed NbS modelling. Similar to BasinMaker, Lake TopoCat provides detailed lake and river hydrology products but is less suited for urban settings without large water bodies.

Table 2-5: Overview of Existing Tools that Model and Quantify the Effects of NbS

Tool	Type	NbS Types Modelled	Benefits	Limitations
Numeric Modelling Tools				
MIKE+ 2D (DHI, 2024)	2D hydrodynamic model + urban drainage modeling	NbS (bioretention systems, permeable pavements, detention ponds)	- Urban flood modelling, stormwater management, and evaluating the impact of NbS. - Integration of stormwater management infrastructure and natural processes. - Can be used in conjunction with SWMM for rainfall-runoff modelling	- High computational requirements for large datasets. - Detailed input data (high-resolution DEMs, land cover, and sewer network configurations).
KINEROS2 (Korgaonkar et al., 2020)	Hydrological model	Urban-scale NbS	- Models small watersheds with detailed urban features. - Can represent pervious areas and stormwater practices like rain gardens and permeable pavements.	- Complex data inputs. - Limited ability to simulate large-scale NbS or regional hydrological effects.
RECHARGE (Richards Equation) (Aravena & Dussaillant, n.d.)	Soil infiltration model	Infiltration-based SCMs (rain gardens, infiltration trenches)	- Models soil water movement under unsaturated conditions. - Simulates rain garden water balance.	- Limited to small-scale interventions. - Detailed soil and hydraulic data is necessary for simulations.
SWMM (Rossman, Lewis A. & Simon, Michelle A., 2022)	Hydrological modelling and stormwater management	NbS (green roof, bioretention systems, permeable pavement, etc.)	- Widely used and thorough documentation - Models urban stormwater runoff, and hydrology and integrates representation of various NbS - Can simulate quantity and quality of stormwater - Open-source, can be modified	- Limited scalability for watershed scale applications - High computational power for large and complex areas - Detailed input data (catchment parameters and NbS configurations)
GIS Based Routing Modelling Tools				
Lake TopoCat (Sikder et al., 2023)	GIS-based hydrological tool	Lake-dominated subwatersheds	- Generates hydrological routing products for lakes and rivers. - Allows for detailed lake effects on hydrological processes in various resolutions.	- Primarily focused on lake-influenced hydrology, less suitable for non-lake urban settings.
BasinMaker (Han et al., 2023)	GIS toolkit	Large-scale detention-based SCMs (ponds, wetlands) and NbS in lake watersheds	- GIS integration for basin delineation. - Flexible hydrological models - Open-source and customizable. - Flow routing through lakes	- Complex data requirements. - Originally focused on lake-dominated watersheds. - Need to be customized for detailed NbS modelling.

The models and tools outlined in Table 2-5 illustrate the various approaches taken to simulate NbS. KINEROS2, for example, is well-suited to small-scale urban green infrastructure (GI) interventions such as rain gardens and permeable pavements, which address urban runoff in compact areas (Korgaonkar et al., 2020). For infiltration-based systems, RECHARGE is particularly valuable as it applies the Richards equation to estimate soil water balance. This model enables detailed simulation of rain gardens by incorporating soil infiltration data; however, it is constrained by the need for accurate, high-resolution soil data and is thus generally limited to minor interventions (Dussaillant et al., 2004).

In contrast, models like BasinMaker address large-scale NbS, such as detention ponds and wetland systems, which require extensive hydrological routing capabilities. BasinMaker excels in lake-dominated watersheds where basin delineation and flow routing are critical. However, it demands significant GIS and high-resolution data support, which can restrict accessibility (Han et al., 2023). The Lake TopoCat tool provides unique insights into lake-dominated hydrological systems, an essential aspect for NbS in regions with significant lake influence. Although primarily focused on hydrological processes in lake systems, its lake-specific functionality restricts its broader applicability in urban NbS projects that lack prominent water bodies (Sikder et al., 2023). MIKE+ 2D integrates 2D hydrodynamic modelling with urban drainage networks, offering robust solutions for simulating flood-prone environments and surface runoff in urban areas. While its capabilities include assessing NbS impacts like detention ponds and permeable pavements, its reliance on detailed input data and high computational demands can pose challenges. Additionally, MIKE+ 2D specifically focuses on hydrodynamics, which may limit direct erosion or sediment transport modelling.

Each tool presents trade-offs between precision, environmental scope, and computational intensity, indicating that current models, while varied in capabilities, lack the flexibility to integrate multiple NbS types or environmental settings within a single framework. Models such as KINEROS2, RECHARGE, and SWM effectively address small-scale urban GI but are unsuitable for watershed-scale applications, while BasinMaker addresses large-scale NbS at the expense of usability in smaller or more urbanized areas. The limited capacity for combined erosion and hydrological modelling, essential in urban NbS, underscores the need for tools that can scale across diverse interventions and geographies. The limitations presented by existing tools underscore the importance of advancing models that support the simultaneous assessment of diverse NbS interventions, particularly as erosion models are often isolated from other hydrological dynamics. Expanding on existing methodologies

and developing a more flexible, multi-scale model would aid sustainable land management efforts. This approach could guide better-informed decisions around NbS and enhance resilience against urban watershed erosion.

2.3.3 Review of Existing SPIN Versions

With urbanization, increased flows with more energy are observed; therefore, higher stream power is observed. SPINarc calculates the specific stream power for rural and urban scenarios. Using nine tools, SPINarc can assess an urban river's stability at the watershed scale using a stream power approach (Figure 2-4). These nine tools include the ability to analyze a DEM to extract river networks, find the channel segments, calculate the average slope, calculate the stream power for rural scenarios, and calculate the urban scenarios using modelled data and land use data (Ghunowa et al., 2021).

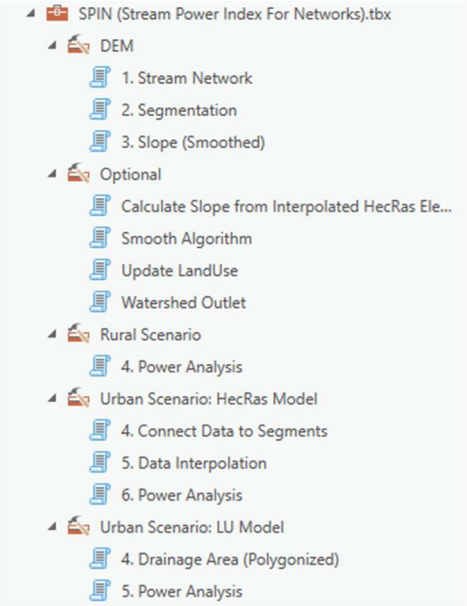


Figure 2-4: SPINarc Tools

SPINarc has two possible analyses, rural and urban and all SPINarc analyses require a DEM input. Equations 2-2 and 2-3 highlight the importance of Q and W for stream power analyses in rural and urban contexts. The rural analysis in SPINarc requires rural empirical equations (Equations 2-4 and 2-5) to calculate the specific stream power. A rural scenario is calculated based on an analysis of a

digital elevation model (DEM) and empirical relation to determine the stream network and values for S, W, and Q. In SPINarc, channel width (W) is calculated from the drainage area (DA, km^2):

$$W_{PD} = a(DA)^b \text{ (Equation 2-4)}$$

where a and b are derived from statistical regression, and the subscript PD refers to a rural watershed regional study (Phillips & Desloges, 2014) from which the values for a and b were obtained. The implicit assumption of the Equation is that width can be modelled as a power function of DA (Equation 2-7). However, SPINarc does not account for channel enlargement. Channel deformation may occur due to numerous reasons including erosion, deposition, increase of impermeable surfaces resulting in increased flows from urbanization (Charlton, 2007; Gregory et al., 1992). Therefore, using the rural equation (Equation 2-4) in an urban context will reflect incorrect channel morphologic conditions. In rural areas, Q_r can similarly be expressed as a power law.

$$Q_r = c(DA)^d \text{ (Equation 2-5)}$$

where c and d are regional coefficients, and the subscript r refers to the rural scenario. The urban analysis has two scenarios that can be calculated a) Modelled Analysis, which requires hydraulic model outputs, and b) LU Analysis, which requires GIS land use data. Following Bledsoe and Watson (2001), the LU Analysis calculates an urban estimate of Q_u that is sensitive to the percent imperviousness (IA), which can be estimated from the land use information as:

$$Q_u = c(DA)^d(IA)^e \text{ (Equation 2-6)}$$

where e is an empirical coefficient, and the subscript u refers to the urban scenario. The LU Analysis is beneficial when hydraulic models, built with detailed knowledge of watershed hydrology and channel geometry, are unavailable. In SPINarc, the specific stream power ratio (SSPR), calculated as the ratio of rural to urban specific stream power, can be used to assess the impact of urbanization and interpret erosion patterns (Ghunowa et al., 2021; Papangelakis et al., 2022).

$$\text{SSPR} = \frac{Urban\ Specific\ Stream\ Power}{Rural\ Specific\ Stream\ Power} = \frac{\frac{\Omega_u}{W_u}}{\frac{\Omega_r}{W_r}} = \frac{\frac{\rho g Q_u S}{W_u}}{\frac{\rho g Q_r S}{W_r}} \text{ (Equation 2-7)}$$

Once the urban stream power is calculated, the specific stream power ratio or the erosion risk can be calculated by dividing the urban by the rural. This indicates the magnitude of change from an

undeveloped rural condition. Watershed managers can use the erosion risk value to identify areas needing restoration efforts or any other interventions.

While SPINarc's empirical equations were effective, SPINarc faced performance limitations, particularly in processing speed and memory constraints. These issues stemmed from the polygon method used for extracting impervious surfaces, which affected computational efficiency when handling large DEM datasets (Ghunowa et al., 2021). The polygon approach extracts the imperviousness contributing to each channel segment. The polygon method includes squaring and multiplying the cell size of the DEM by the drainage area upstream of each segment (Ghunowa et al., 2021). The drainage area polygons intersect with land use polygons to calculate the area covered by each land use type. The total impervious area is determined and summed for each polygon by multiplying each land use area by its associated imperviousness. The ratio of the total impervious area to the total drainage area is then calculated as a percentage and assigned to each discharge point feature of the reaches (Ghunowa et al., 2021).

To address these challenges, the SPINpy tool was developed by Ghunowa et al. (2021) using Python, leveraging libraries such as GDAL, rasterio, NumPy, and Whitebox Geospatial Tools (Lindsay, 2016) to improve processing speed and handle detailed DEMs more efficiently. SPINpy further reduces the need for manual intervention, offering faster processing and improved support for high-resolution datasets. Table 1-1 outlines the capabilities of each iteration of the tool. SPINarc's capabilities have already been incorporated into the CVC's Risk and Return on Investment Tool (RROIT) as SPINpy. SPINpy was the next version created to address the accessibility barriers posed by the ESRI license in SPINarc and to allow for an open-source script. SPINpy was developed in Python for CVC to run the Modelled Analysis with imported toolboxes from Whitebox. SPINpy only allows for the Urban-Modelled Analysis but not the Urban-Land Use Analysis.

In addition to SPINarc and SPINpy, several other versions with various refinements have been developed. Other notable sub-branches are SPINarc_width and SPINpy_speedy. SPINarc_width is a sub-branch that Victoria Barlow and Dr. Peter Ashmore updated at the University of Western Ontario to account for width changes in response to urbanization and an increase in discharge. SPINpy_speedy is the second version of SPINpy that was optimized for speed in partnership with partners from the Computer Systems Group at the University of Waterloo.

Despite the advancements in the SPIN series, there are significant limitations, particularly in the context of assessing the effectiveness of stormwater management solutions, both engineered and nature-based. This gap underscores the need for more comprehensive analytical frameworks capable of capturing the benefits of NbS at a high level. As urban areas increasingly adopt NbS to manage stormwater, it is essential to explore dedicated tools that effectively evaluate their impacts at a watershed scale for screening purposes.

2.4 Research Gaps and Objectives

Although SPINarc has proven useful for stream power analysis, its application to watersheds provided by the CVC has highlighted several limitations in its algorithms and overall performance (see Table 2-6). These limitations reveal the need for enhancements to improve accuracy, flexibility, and usability, particularly for urbanized watersheds.

The overall objective of SPINpy 2 is to build upon the existing SPINpy tool to add a Land Use Analysis and further develop an open-source platform that facilitates wider use of its algorithms. The project aims to improve the performance and prediction accuracy of the SPINarc tool by creating an open-source tool called SPINpy 2 so the algorithms can be more widely used. Additionally, SPINpy 2 will model engineered stormwater management and NbS infrastructure in SPINpy 2 to depict the effects of stormwater management solutions (specifically stormwater management ponds) on stream power reduction and, thus, erosion risk. Table 2-6 outlines the specific research gaps identified in SPINarc's current capabilities and the corresponding objectives to overcome these challenges.

Table 2-6: Research Gaps And Objectives

Limitation #	Research Gap	Research Objective
1	Memory limitations within ArcMap. SPINarc is an ArcToolbox package, large watersheds and detailed DEMs require the data to be split and recompiled outside the tool. Additionally, the cost of an ESRI license is a barrier for users.	Bypass memory and licensing limitations by creating an open-source Python tool.
2	The LU Analysis uses a computationally intensive polygon method to determine IA. While beneficial, the polygon method often exceeds ArcMap's memory capacity. Causing significant processing delays based on the size and detail of the input watershed DEM.	Revision of the algorithm used to extract the IA values.
3	Stormwater management will modify the peak discharge in urban areas, but the current empirical methods do not capture any benefits from the implemented stormwater management solutions (engineered or nature-based)	Develop a justifiable method to model the effects of ponds.
4	Modelled Analysis periodically creates discontinuities between the modelled width values and the rural empirical width equation and is not sensitive to the additional flows due to urbanization.	Introduce boolean checks within the Python script to ensure the urban channel width does not go below the rural empirical width.
5	SPINarc is a single-watershed tool, but decision-making and prioritization are often done across multiple watersheds following political boundaries.	Another analysis option should be added based on a boundary that identifies the upstream watersheds based on the pourpoints exiting the boundary. Then, it executes, compiles, and clips the SPINpy 2 results to the provided boundary.
6	The LU Analysis discharge equation (Equation 2-6) is not widely tested, so there is only a limited understanding of the variability of the exponent e.. Ensuring the discharge equation can accurately predict the 2-year discharge in the LU Analysis is crucial, as the LU Analysis relies on available land use and DEM data. The equation and its coefficients must accurately reflect urbanization conditions to represent the 2-year discharge conditions without detailed hydrologic data.	Calibrate a new empirical discharge equation using existing hydrologic modelling outputs of the 2-year discharge.
7	The LU Analysis computes a single-width estimate. Equation 2-4 for both rural and urban scenarios overlook channel enlargement due to increased flows from urbanization and will overestimate SSPR.	Calibrate a new width equation that accurately represents expected channel widening with urbanization.

The review of existing tools and methodologies reveals significant gaps in erosion risk prediction, particularly for urbanized watersheds. These challenges have motivated the development of SPINpy 2, which specifically addresses watershed managers' needs by compiling existing SPIN versions, improving discharge predictions, and incorporating NbS. The enhancements developed in SPINpy 2 aim to increase accessibility, prediction accuracy, and modelling flexibility. By integrating new empirical discharge and width equations along with modelling NbS, the completion of SPINpy 2 will provide a comprehensive tool for erosion risk assessment.

Chapter 3
SPINpy 2 Scenarios & Study Areas

The Credit River Watershed (CRW) was used to calibrate and test the SPINpy 2 tool developed for the current research. A comprehensive understanding of the Credit River Watershed (CRW) is essential to effectively apply and demonstrate the capabilities of SPINpy 2.

The Credit Valley Conservation Authority (CVC) oversees the CRW, which includes the 90 km long Credit River and several smaller watercourses flowing into Lake Ontario. The CRW is primarily natural (36.3%), agricultural and open space (36.0%) in the northern part of the watershed. The land classified as urban areas (27.7%) is primarily concentrated in the southern portion of the watershed near Lake Ontario (Figure 3-1).

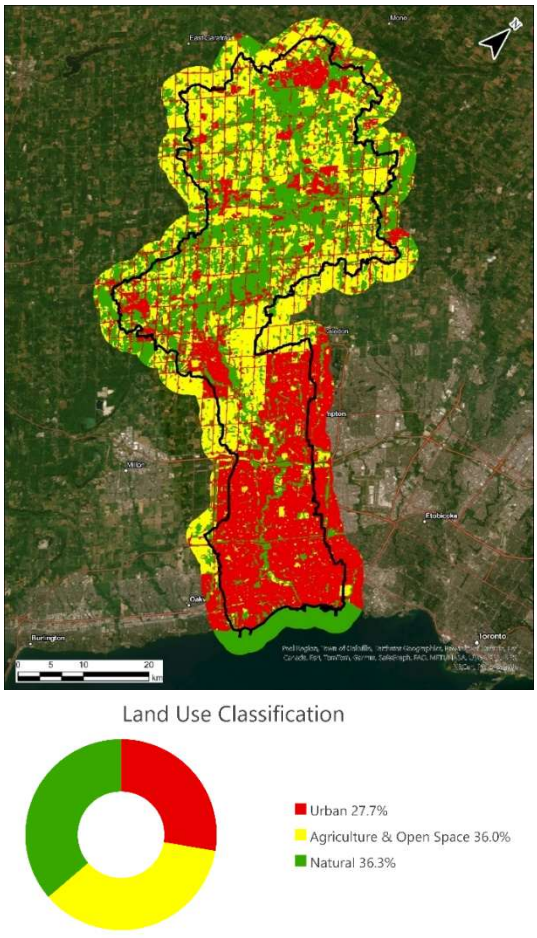


Figure 3-1: Credit River Watershed (CRW) Land Use Classification.

Detailed hydrologic data for eight (8) subwatersheds located within the CRW was used to build and test SPINpy 2. This chapter provides a summary of the subwatersheds used for calibration and validation. While all of the data provided by the CVC was used for calibration, validation, and analyses, only a select few subwatersheds were chosen to be presented in this thesis to demonstrate the abilities of SPINpy 2.

The latter half of this chapter describes the topographic and land use conditions of the subwatersheds used to demonstrate the SPINpy 2 algorithms in this thesis. This information provides geographic context and establishes the boundaries for this thesis's calibration and validation analysis, offering a foundation for the scenarios discussed in Chapters 4 – 6. A detailed explanation of SPINpy 2's scenario algorithms will be discussed in Chapter 4. This chapter will also introduce the scenarios available in SPINpy 2, the No Stormwater Management (NSM) Scenario and the Engineered Stormwater Management (ESM) Scenario.

NSM Scenario: The NSM Scenario allows users to run a Modelled and LU Analysis, assuming that the area of interest lacks stormwater management measures (Figure 3-2). The users can run the NSM for either a Single Watershed (SW) or Multiple Watersheds (MW) within a user-specified boundary.

ESM Scenario: The ESM Scenario allows users to run a Modelled and LU Analysis with a user-provided pond layer. Similar to the NSM Scenario, users can run the ESM for a Single Watershed (SW) or Multiple Watersheds (MW).

3.1 Data Used for SPINpy 2 Scenario Calibration

3.1.1 Discharge Equation

The CVC has provided 2-year bankfull discharge and width outputs from detailed hydrologic models and measured width data to calibrate and validate the revised equations. The hydrologic model is created using instantaneous 15-minute data from the CVC flow gauges. The data provided in Table 3-1 was used to calibrate the NSM and ESM LU Scenario urban discharge equations. The new algorithms and equations are calibrated and validated using 1m x 1m digital elevation models (DEM), land use shapefile, pond shapefile, detailed hydrologic model outputs, and gauge data provided by the CVC.

Table 3-1: Subwatersheds Used in the Calibration of the NSM and ESM Discharge Equations

Subwatershed			NSM Scenario Calibration	ESM Scenario Calibration		
Name	*Area (km^2)*	*Average Impervious (%)*	*Total Data Points*	*Total Data Points*	*Number of Ponds Recorded*	*Corresponding Gauge Name*
Turtle	1.51	24.85	77			
Serson	2.03	25.22	24			
Birchwood	3.24	20.91	141			
Applewood	3.47	29.46	87			
Loyalist	6.13	32.52	159	159	5	Loyalist Creek at 5th Line
Levi	22.58	14.39	207	207	12	Levi Creek at Financial Drive
Cooksville	25.74	34.62	319			
Fletchers	38.52	24.00	260	260	47	Fletchers Creek at 2nd Line

Detailed hydrologic models outputs for eight subwatersheds were used to calibrate, validate, and analyze the discharge empirical equation in the NSM Scenario (Figure 3-2). The models represent a diverse range of conditions within the watershed and can convey characteristics of having no stormwater management controls in the NSM urban discharge equation.

The CVC had the most records available for stormwater management ponds, thus, ponds were the first NbS approach introduced to SPINpy 2. The CVC has also provided discharge and width outputs for three (3) subwatersheds from their hydrologic models and gauges data to be used as calibration and validation data for the ESM Scenario urban discharge equation (Figure 3-2). Few gauging stations exist for the relatively small urban basins being investigated. However, validating the ESM Scenario discharge equation using the few gauges available will allow a more accurate representation of urban hydrology and uncertainty in the simple equation. Three gauges are installed within the ESM subwatersheds of interest (Table 3-1 & Figure 3-2), and the gauges recorded 15-minute data from 2015 to 2022. The data provided included the winter months; however, for our analyses, the data for the winter months of January, February, March, and December were omitted from the dataset.

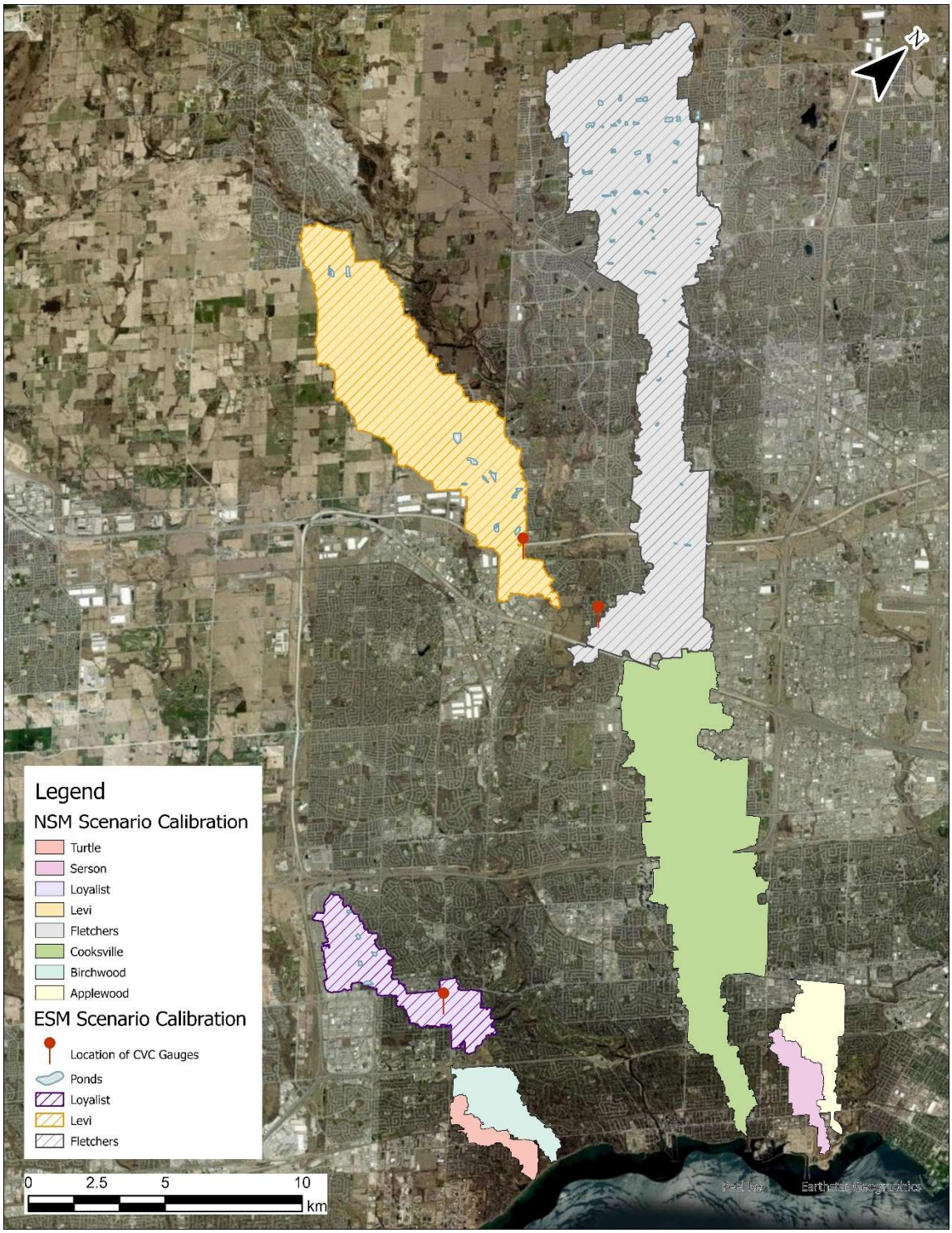


Figure 3-2: Depiction of the subwatersheds used to calibrate the NSM and ESM discharge equation and the gauge stations used to validate the ESM discharge equation.

3.2 SPINpy 2 Application to Several Case Studies within the CRW

To illustrate the capabilities of SPINpy 2, specific subwatersheds within the CRW were selected for analysis. While data from the CVC was utilized for calibration and validation, this thesis presents a select few of these subwatersheds to demonstrate the application of SPINpy 2 in different scenarios. SPINpy 2 was tested on both single-watershed and multiple-watershed NSM Scenarios and the ESM Scenario to verify its flexibility and performance in urbanized environments.

The subwatersheds chosen for these scenarios include Cooksville Creek, Serson Creek, Levi Creek, and Applewood Creek, as detailed in Table 3-2. These subwatersheds were selected due to their varying degrees of urbanization, stormwater management infrastructure, and topographic features, offering a comprehensive view of SPINpy 2's ability to handle different hydrological conditions.

Table 3-2: Application of SPINpy 2 to Study Areas

NSM		**ESM**
SW	*MW*	*SW*
Cooksville Creek	Serson	Levi Creek
Levi Creek	Applewood	-

The following section will briefly describe the topographic conditions of the selected subwatersheds discussed in this thesis and provide the inputs required to run each scenario.

3.2.1 NSM Single Watershed (SW)

An NSM Single Watershed (SW) analysis allows users to conduct a LU or Modelled data-based stream power analysis for one watershed of interest. The NSM Scenario does not consider the effects of stormwater management ponds within the analysis. Two study subwatersheds, Cooksville Creek and Levi Creek, were selected to demonstrate the scenario because the sites offer a contrast between urban and rural sites due to their similar size but contrasting levels of urbanization (Figure 3-3).

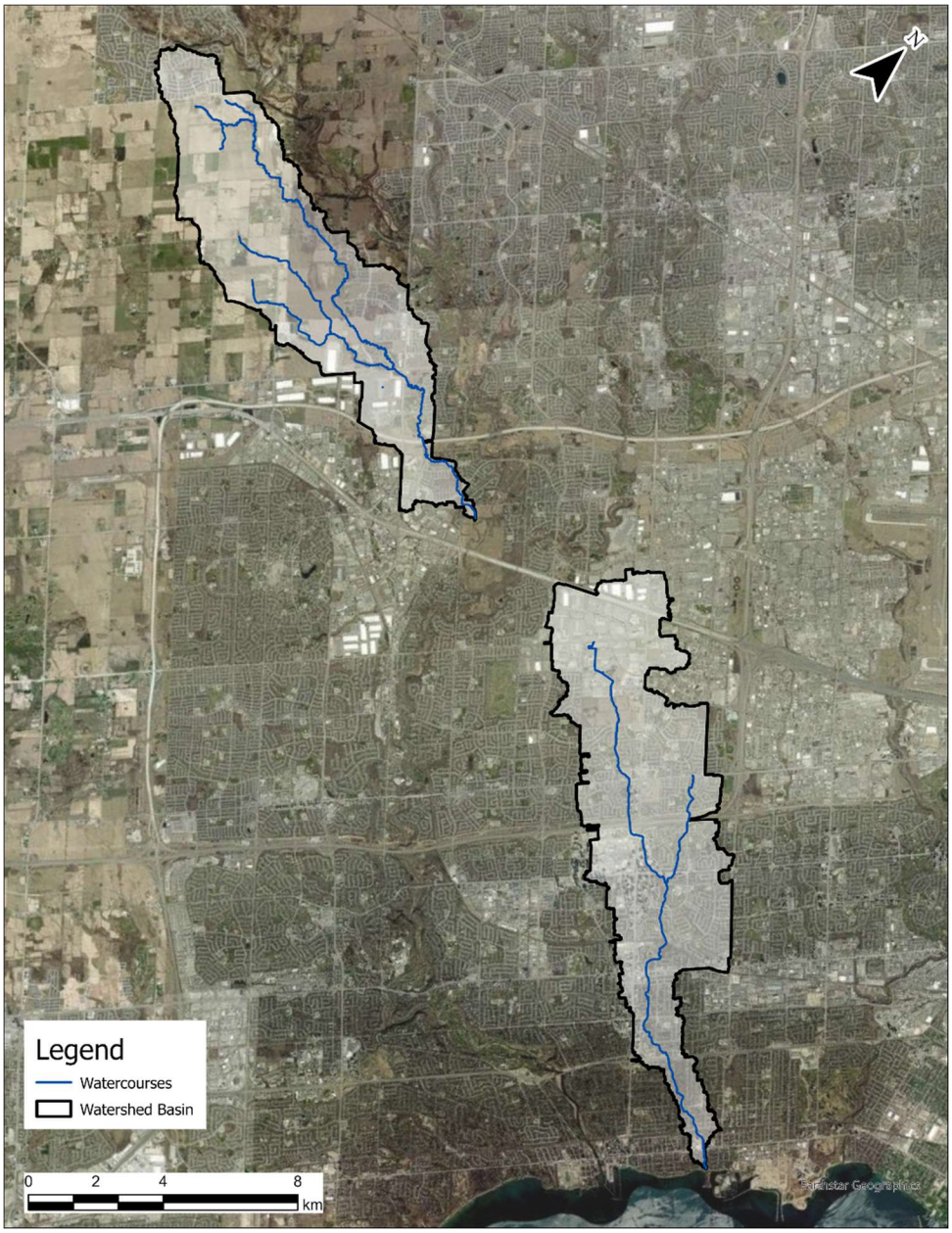


Figure 3-3: Subwatersheds Used for the Single No Stormwater Management Analysis

Several key inputs are necessary to execute the SW analysis, depending on whether the Land Use (LU) or Modelled Analysis is being applied. For the LU Analysis, users must provide:

- Digital elevation model (DEM). The input resolution of the DEM determines the outputs resolution.

- Ecological Land Use Type (or other type) of shapefile that has a column that includes unique land use types;
- An Excel spreadsheet that assigns an imperviousness for each unique land use type (Table 3-3) and;
- An Excel spreadsheet with approximate headwater coordinates of watercourses of interest which are used to extract the stream network.

Table 3-3: Impervious percent assigned to each land use type necessary for the LU Analysis inputs. Impervious percent values are obtained from various sources (Guthrie, Rick & Deniseger, John, 2001; Tilley, Janet & Slonecker, E. Terrence, 2006)

Type	Imperviousness	Type	Imperviousness	Type	Imperviousness
Active aggregate	1	Inactive aggregate	1	Private open space	1
Airport	90	Institutional open space	1	Railroad	90
Collector	90	Intensive agriculture	1	Recreational open space	1
Commercial / industrial	70	Landfill	80	Regional road	90
Commercial / industrial open space	1	Low density residential	25	Residential estate	6
Coniferous forest	1	Major trail	1	Rural development	8
Coniferous plantation	1	Manicured open space	1	Shrub beach / bar	1
Coniferous swamp	0	Marsh	0	Shrub bluff	1
Construction	90	Medium density residential	35	Shrub bog	0
Cultural hedgerow	1	Mixed forest	1	Shrub clay barren	1
Cultural meadow	1	Mixed plantation	1	Shrub fen	0
Cultural savannah	1	Mixed residential	42.5	Shrub sand barren	1
Cultural thicket	1	Mixed shallow aquatic	0	Shrub talus	1
Cultural woodland	1	Mixed swamp	0	Submerged shallow aquatic	0
Deciduous forest	1	Non-intensive agriculture	1	Thicket swamp	0
Deciduous plantation	1	Open aquatic	0	Treed beach / bar	1
Deciduous swamp	0	Open beach / bar	1	Treed bluff	1
Educational / institutional	80	Open bluff	1	Treed bog	0
Floating-leaved shallow aquatic	0	Open clay barren	1	Treed cliff	1
General urban	80	Open fen	0	Treed fen	0
High density residential	35	Open rock barren	1	Treed sand barren	1
High rise residential	67.5	Open sand barren	1	Treed talus	1
Highway	90	Other open space	1	Wet meadow	0

For a Modelled Analysis, users must provide:

- DEM;
- Shapefile containing the stations from a detailed hydrologic model;
- An Excel file containing the hydrologic model outputs of a 2-year storm event.

This subsection will provide a brief description of the geographic location, topography, land use, elevation, predominant soil types, and SPINpy 2 inputs specific to Cooksville and Levi Creeks.

3.2.1.1 Cooksville Creek

3.2.1.1.1 Topographic and Land Use Conditions

Cooksville Creek is a highly urbanized 26 km^2 subwatershed in the City of Mississauga that outlets to Lake Ontario. The diverse land use and significant urbanization within the Cooksville Creek subwatershed provide a comprehensive basis for analyzing the impact of land use on watershed erosion and flood risk explored in subsequent sections. The subwatershed's physical features and land use patterns are listed below to better understand the watershed's characteristics.

- The subwatershed's headwaters lie approximately 200 masl and slope down to 73 masl at the outlet to Lake Ontario.
- A review of the Ontario Soil Survey (OMAFRA) indicates that the land is a built-up area with no identified soil type. The predominant soils in this area have poor drainage due to the high built-up areas, contributing to limited infiltration and increased surface runoff.
- Sharpe et al. (1997) describes the geology in the upper watershed near Heartland Town Center as Halton Till. The lower watershed near Square One is described to have Glacial Lake Deposits and bedrock (limy mudrock and clastic sedimentary rock) (Sharpe, DR et al., 1997).
- Cooksville Creek subwatershed exhibits a high degree of urbanization with pockets of natural, agricultural, and open spaces (Figure 3-4). Based on the Ecological Land Classification of Ontario, Cooksville Creek is 81.1% urban, 8.8% natural, and 10.2% agricultural land.
- The natural land cover includes woodland, forests, and meadows.
- The agricultural and open spaces include farmland, parklands, and institutional, recreational, and commercial open spaces.
- The urban areas consist of low- to high-density residential, commercial, and industrial developments with significant impervious surfaces that reduce infiltration capacity. These highly impervious areas have limited infiltration, increasing runoff and higher peak flows. Cooksville Creek, a highly flood-prone subwatershed, has historically experienced

flooding, with notable flood events recorded in recent decades (Credit Valley Conservation Authority et al., 2021).

The land use composition and physical characteristics of the Cooksville Creek subwatershed have substantial implications for erosion and flood risk within the subwatershed.

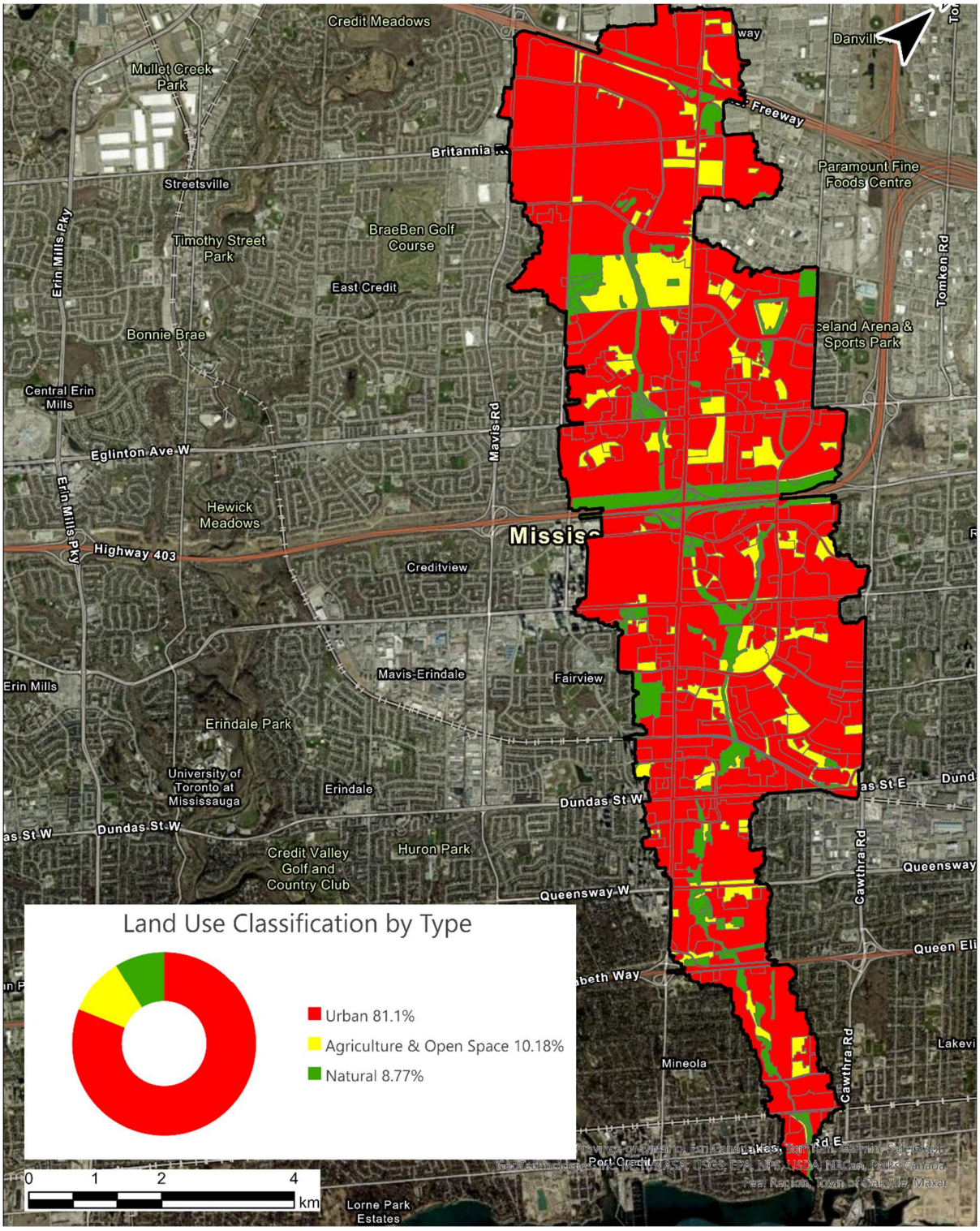


Figure 3-4: Cooksville Creek Land Use Classification

3.2.1.1.2 Modelled and LU Analysis Inputs

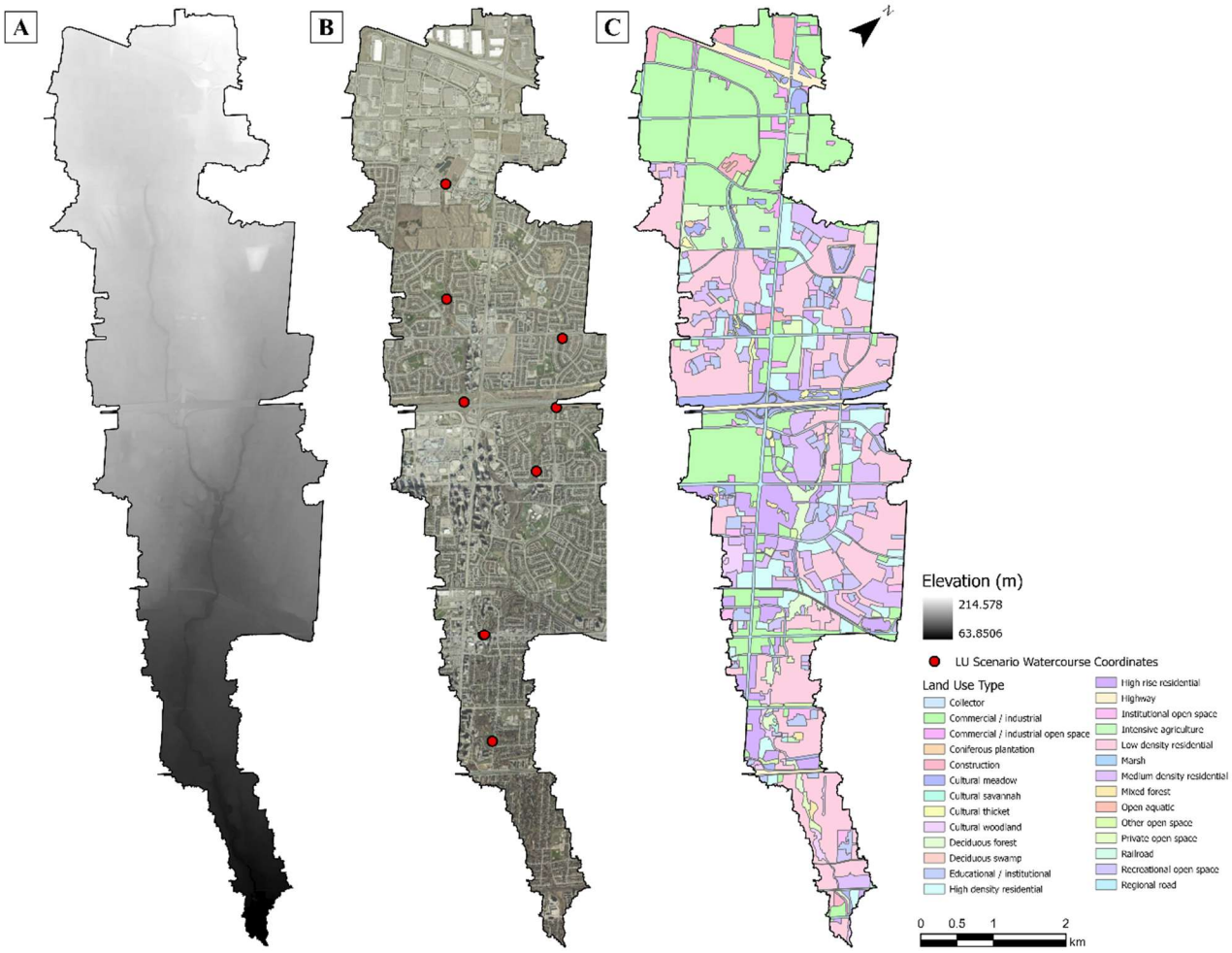


Figure 3-5: Land Use Analysis Inputs A) 1 m Cooksville Creek digital elevation model B) LU Analysis watercourse coordinates C) Cooksville Creek land use by type

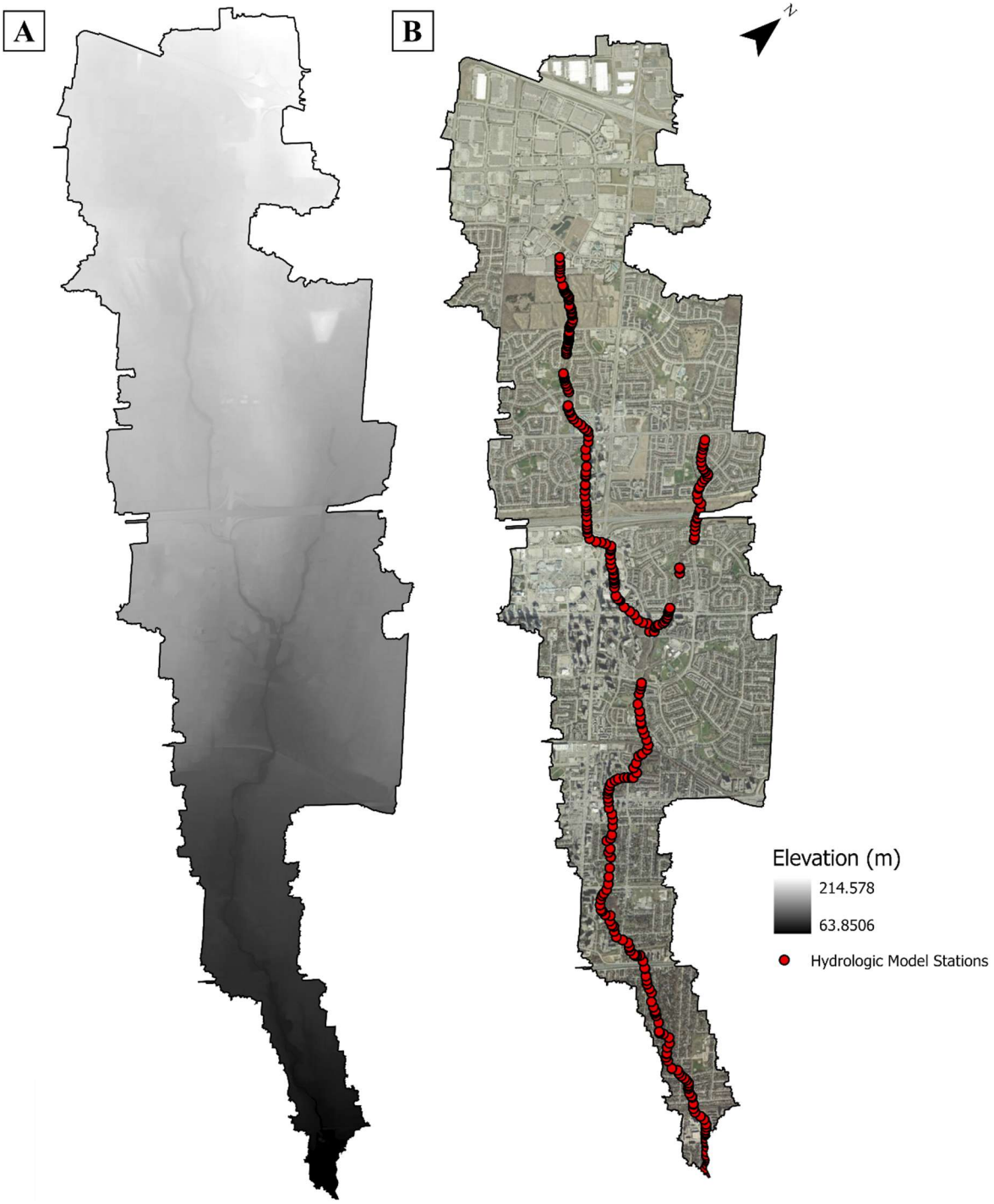


Figure 3-6: Modelled Analysis Inputs A) Cooksville Creek digital elevation model B) Detailed hydrologic model stations

3.2.1.2 Levi Creek

3.2.1.2.1 Topographic and Land Use Conditions

Levi Creek is a 23 km^2 subwatershed beginning in the City of Brampton, travelling southeast to the City of Mississauga and outletting to the Credit River. The subwatershed's physical features and land use patterns are listed below to provide a clear understanding of the watershed's characteristics.

- The subwatershed's headwaters lie at an elevation of approximately 252 masl, descending to 161 masl at the outlet to the Credit River.
- A review of the Ontario Soil Survey (OMAFRA) indicates that the subwatershed contains several soil types, including bottomland, Chinguacousy clay loam, Fox sandy loam, Jeddo clay loam, and Oneida clay loam (Figure 3-7B). These soils vary in drainage capacities, with some areas being well-draining and others poorly draining (Figure 3-7D).
- Levi Creek is 31.7% urban, 8.7% natural, and 49.5% agricultural (Figure 3-7A).

While Cooksville and Levi Creeks are similar in size, their urbanization levels are different. With Cooksville being highly urbanized and Levi Creek having a more balanced land use distribution. This difference allows for a comparative analysis using SPINpy 2's NSM Scenario, highlighting the variations in erosion risk and runoff characteristics between highly urbanized and balanced watersheds.

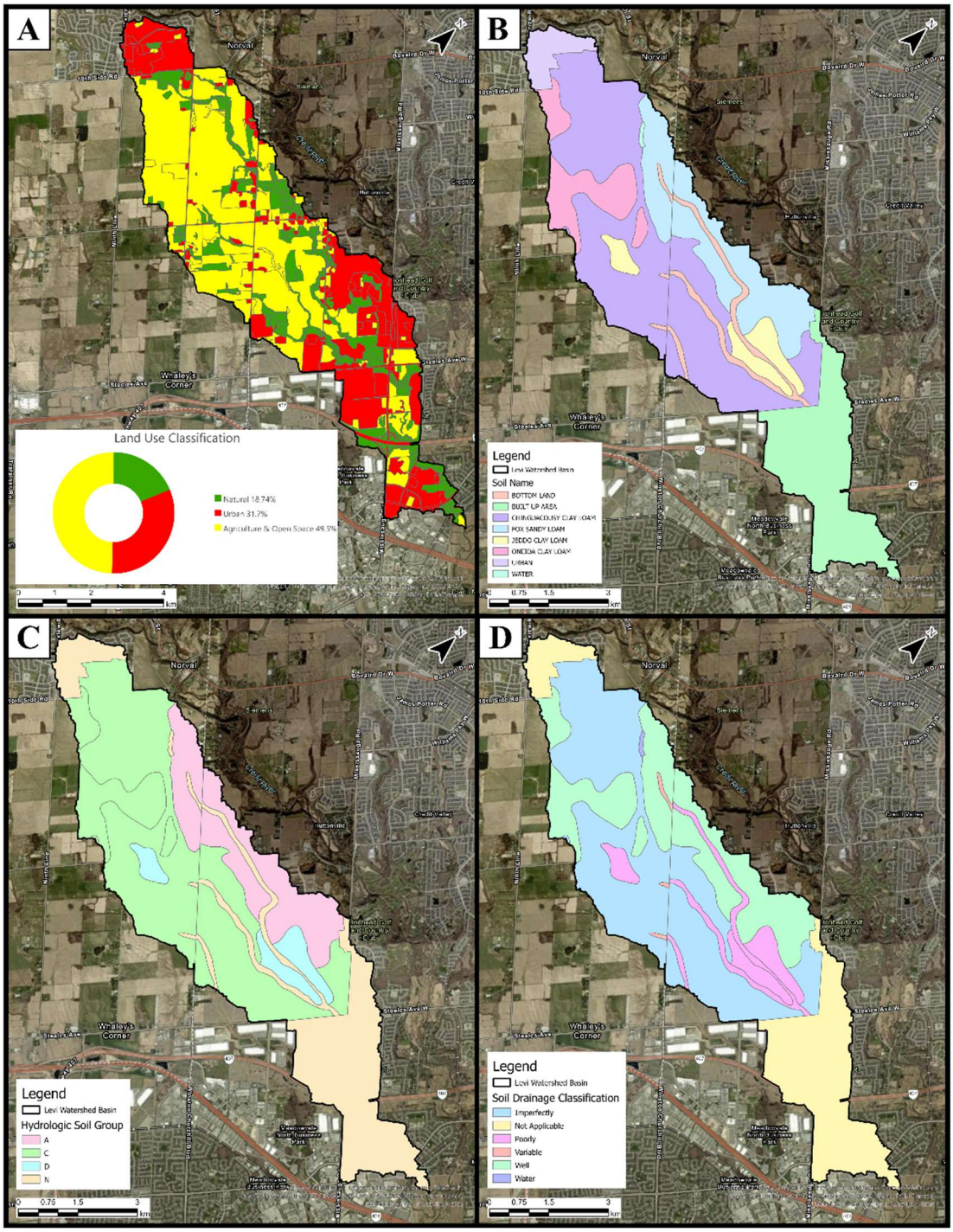


Figure 3-7: Levi Creek Subwatershed Characteristics A) Levi Creek Land Use Classification B) Levi Creek Soil Types C) Levi Creek Hydrological Soil Group D) Levi Creek Soil Drainage Classification

3.2.1.2.2 Modelled and LU Analysis Inputs

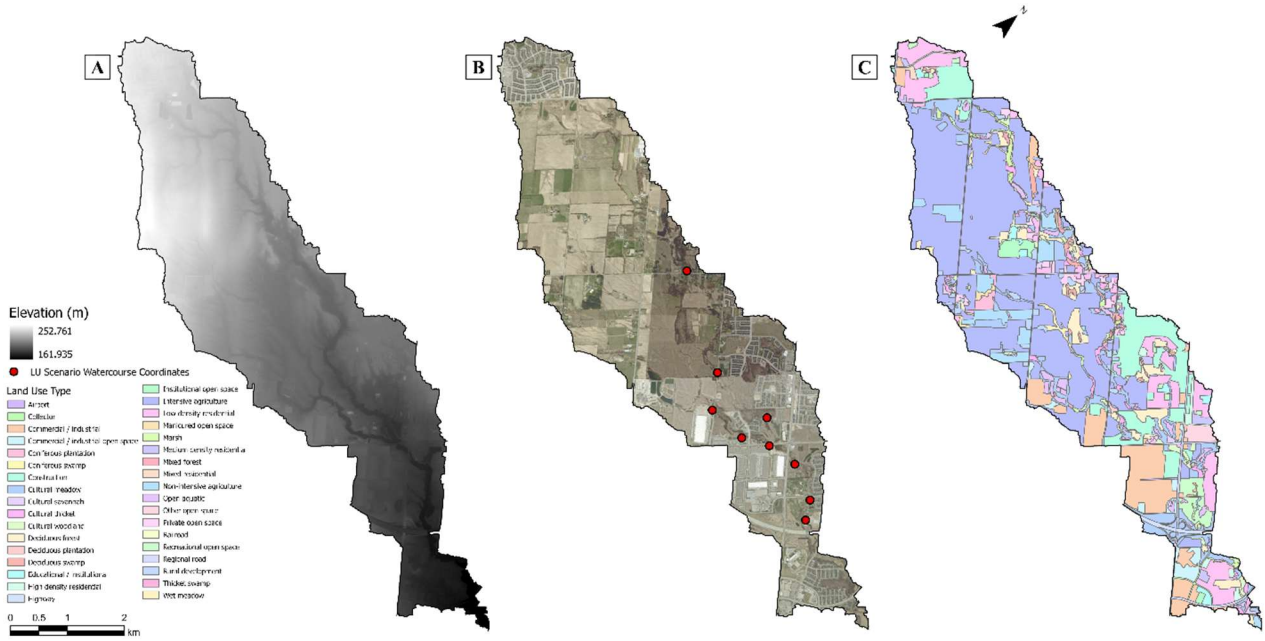


Figure 3-8: Land Use Analysis Inputs A) Levi Creek digital elevation model B) LU Analysis watercourse coordinates C) Levi Creek land use by type

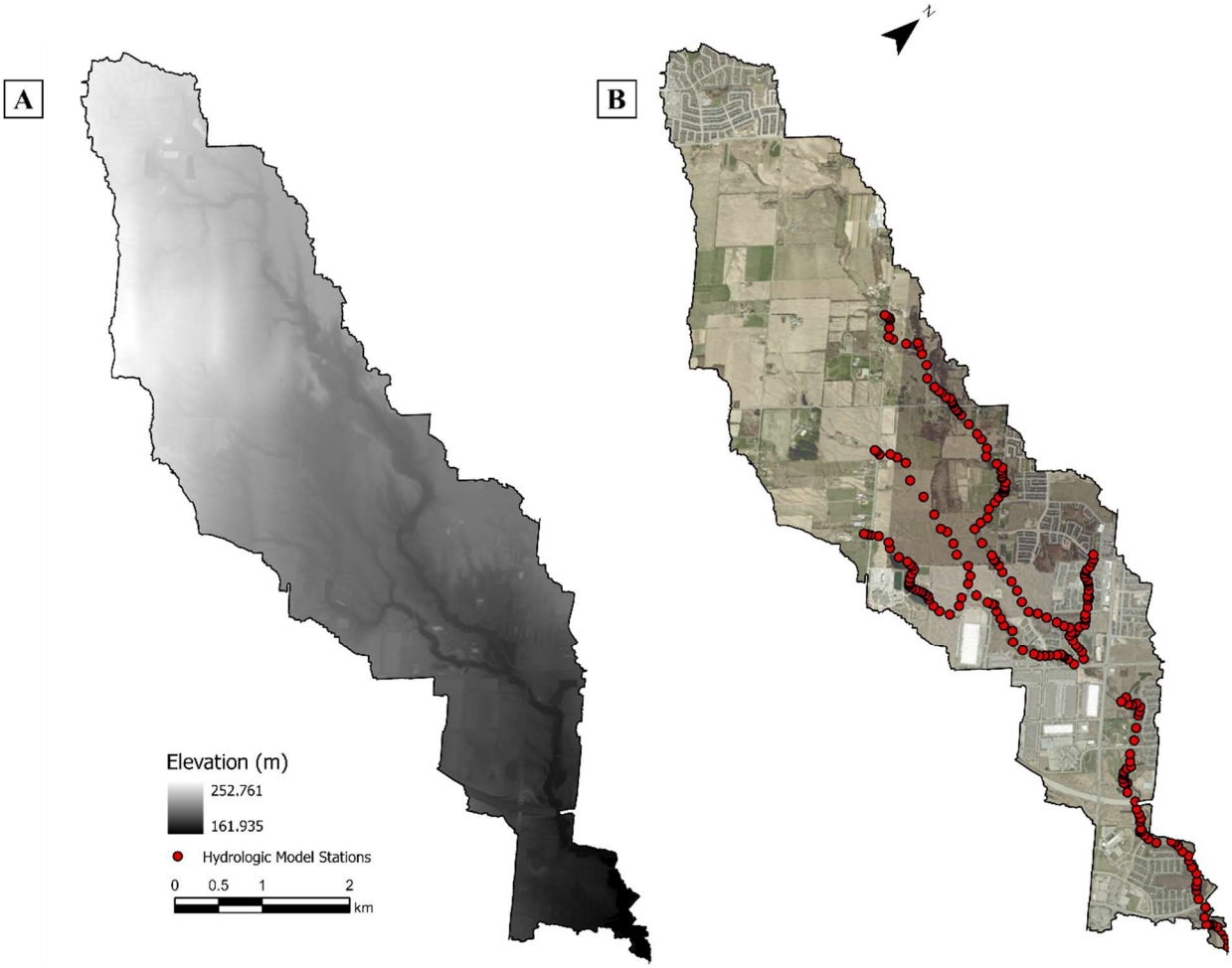


Figure 3-9: Modelled Analysis Inputs A) 1 m Levi Creek digital elevation model B) Detailed hydrologic model stations

3.2.2 ESM Single Watershed (SW)

3.2.2.1 Levi Creek Ponds Information

The ESM analysis models the reduction of surface runoff due to the presence of the ponds. The ESM analysis uses the CVC's spatial pond data for Levi Creek. Levi Creek has 12 stormwater management ponds strategically located throughout the subwatershed to manage and mitigate the impact of urban runoff (Figure 3-10). The effectiveness of these ponds in reducing peak flows is a crucial focus of the analysis. By capturing and slowly releasing stormwater, the ponds help to decrease the volume and speed of runoff entering the creek, which reduces erosion and sediment transport. A comparison of the ESM and NSM analyses evaluates how ponds contribute to overall watershed health and stability,

providing valuable insights for future stormwater management planning. The analysis results can inform strategies for optimizing stormwater management pond placement and design, ultimately enhancing the resilience of subwatersheds like Levi Creek against urbanization impacts.

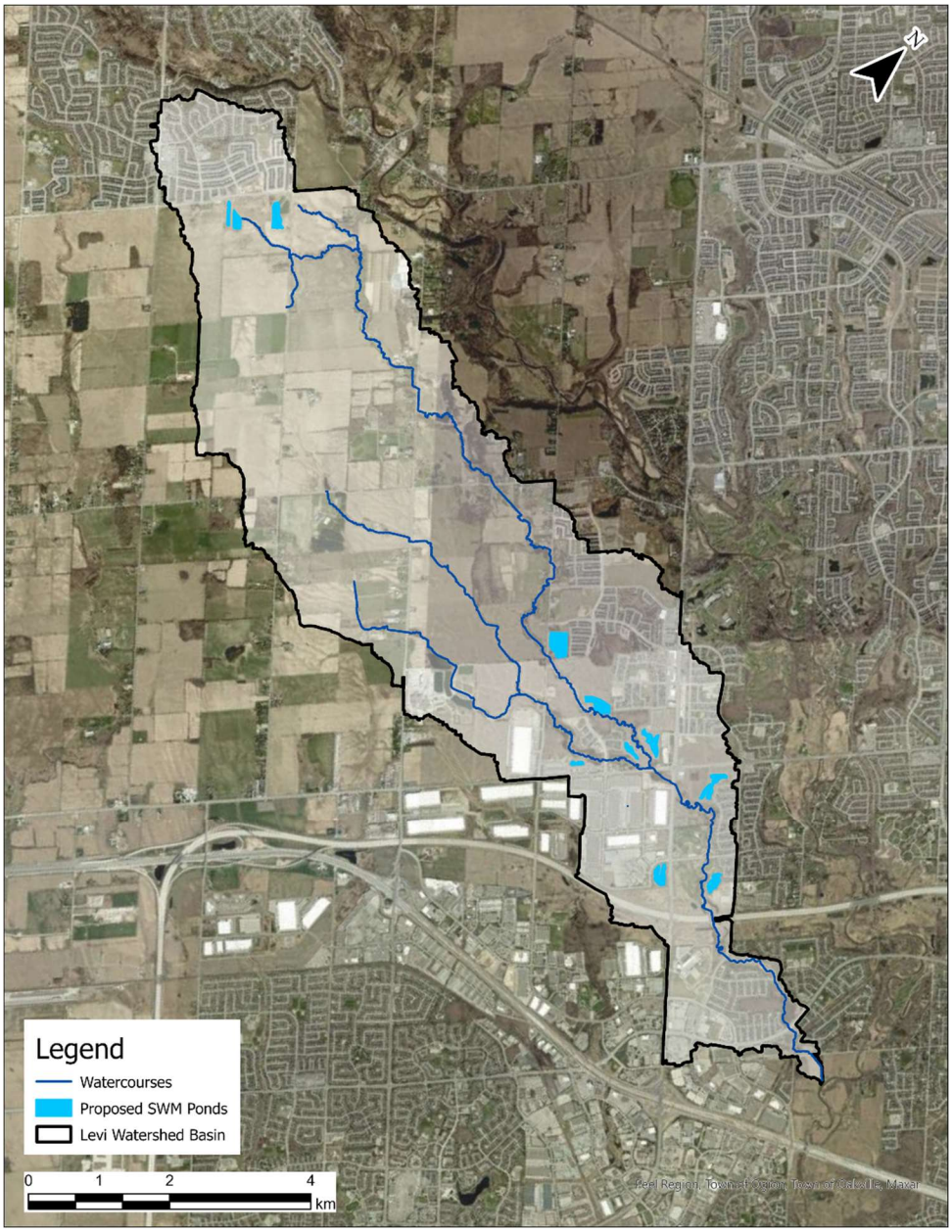


Figure 3-10: Levi Creek Subwatershed Basin and Stormwater Management Ponds

3.2.2.2 Modelled and LU Analysis Inputs

This section will discuss the Levi Creek study subwatershed and its ponds to demonstrate Modelled and LU Analysis using a SW in the ESM Scenario in SPINpy 2.

The LU Analysis inputs required are the same as shown in Figure 3-8. However, the ESM Scenario requires a shapefile with the pond footprint (Figure 3-10). The inputs for the SW Scenario Modelled Analysis in the ESM Scenario are the same as the NSM Scenario (Figure 3-9). However, the detailed hydrologic model outputs must be from a model that contains pond data (Figure 3-10).

3.2.3 NSM Multiple Watersheds within a Boundary (MW)

3.2.3.1 Serson and Applewood Topographic and Land Use Conditions

The Multiple Watershed (MW) is particularly beneficial for users managing multiple watersheds within a defined political or administrative boundary, allowing for comprehensive and coordinated watershed management strategies. In the Modelled Analysis, users must provide specific hydrology model outputs for the selected subwatersheds, including parameters such as precipitation, evapotranspiration, and runoff coefficients. For the LU Analysis, users can indicate headwaters by selecting sample points representing critical inflow areas within the boundary. SPINpy 2 automatically processes the provided data to calculate metrics, including discharge, stream power, channel width, and specific stream power. By consolidating metrics across multiple subwatersheds within a political boundary, the analysis provides an overall view of watershed dynamics, facilitating informed decision-making for resource allocation and environmental protection.

In this thesis, the City of Mississauga's Ward 1 political boundary is used to demonstrate the ability to run the MW analysis in the NSM Scenario. While there are other subwatersheds within this ward, the NSM Modelled and LU Analysis will be executed for Serson and Applewood Creeks because detailed hydrologic model outputs are available to execute the Modelled Analysis (Figure 3-11A). Serson Creek (2.03 km^2) and Applewood Creek (3.47 km^2) are subwatersheds in the City of Mississauga that outlet to Lake Ontario. A review of the Ontario Soil Survey (OMAFRA) indicates that the predominant soils in both Serson and Applewood Creek are poorly draining due to the high built-up areas, contributing to limited infiltration and increased surface runoff. Serson Creek is 80.0% urban, 12.0% natural, and 8.0% agricultural. Applewood Creek is 77.4% urban, 6.8% natural, and 15.9% agricultural (Figure 3-11B). The headwaters of the subwatersheds lie at an elevation of

approximately 95m, descending to 77m at the outlet to Lake Ontario. The benefit of running both subwatersheds simultaneously, particularly the LU Analysis, is capturing the effects of the system of subwatersheds together, not just an individual subwatershed. Chapter 5 will display the inputs and outputs of the analysis.

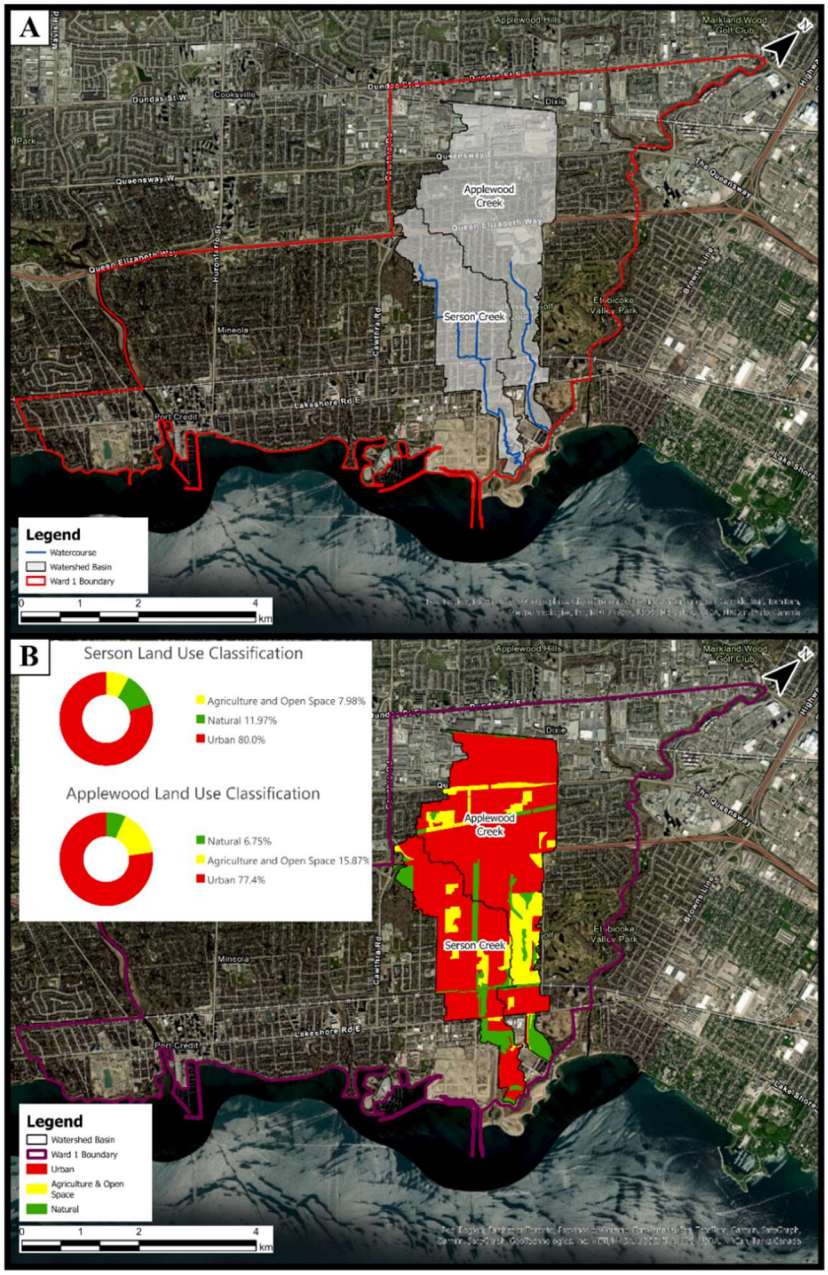


Figure 3-11: Multiple Boundary Analysis A) Serson and Applewood Creeks in the Multiple Boundary Analysis Study Area B) Serson and Applewood Creeks Land Use Classification

3.2.3.2 Modelled and LU Analysis Inputs

The MW Analysis for a SW takes the same inputs as the NSM SW Analysis. However, the user must provide a boundary representing the study area. It is important to note that the DEM provided as input should cover the entirety of the boundary; otherwise, the analysis will only be run for the extent of the DEM.

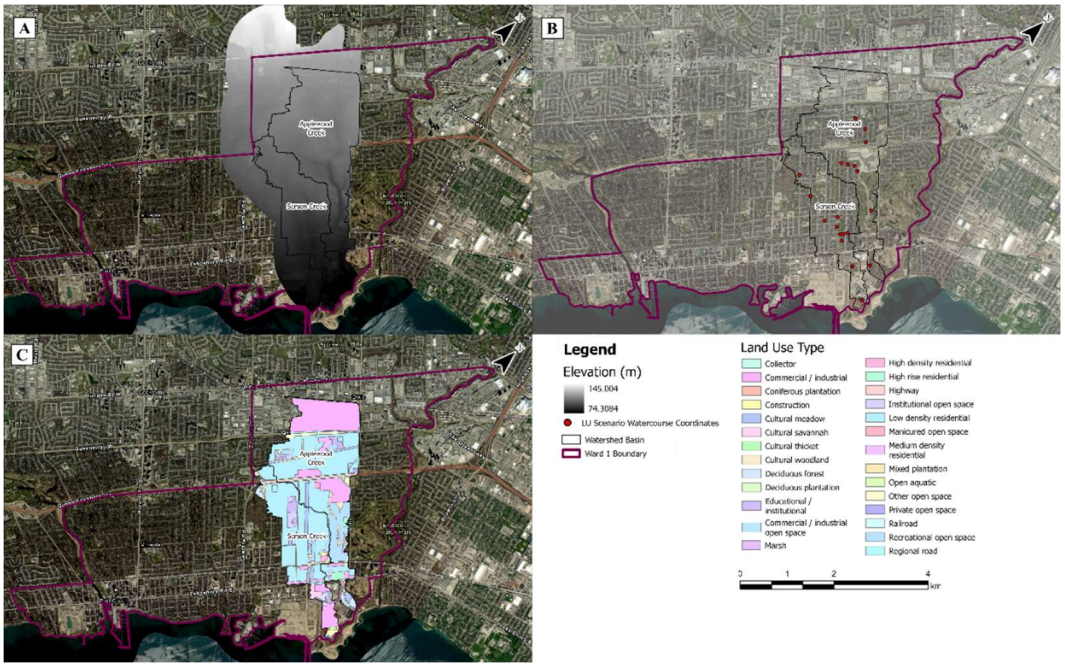


Figure 3-12: Multiple Subwatershed Land Use Analysis with Political Boundary A) Applewood and Serson Creeks DEM, B) LU Points for watercourses of interest, C) LU Shapefile

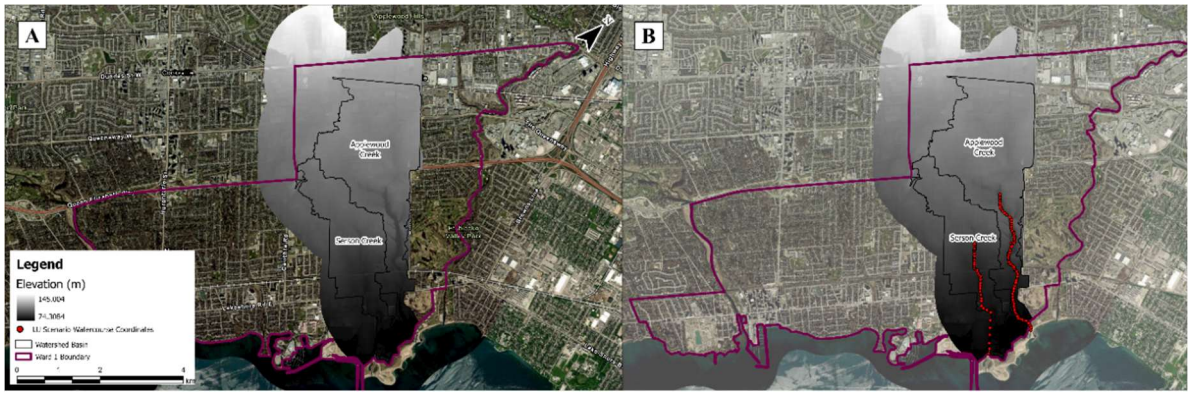


Figure 3-13: Multiple Subwatershed Modelled Analysis with Political Boundary A) DEM, B) Hydrologic Model Stations.

3.2.4 Cooksville Creek Future Scenario ESM Single Watershed Analysis

While SPINpy 2 does not include a specific future scenario analysis, users can leverage existing scenarios within the platform to conduct future analyses using different data inputs. This approach facilitates the prediction of the watershed's response to potential changes. The future scenario analysis is an asset to planners and watershed managers because it enables them to model proposed developments and the optimal configuration of new stormwater management ponds to reduce downstream erosion before construction. Suppose a proposed development, such as an industrial building, is to be considered. In that case, the user must include the new development in the Land Use Shapefiles and categorize it according to the appropriate land use type before conducting the LU analysis to simulate new developments. This process ensures that new developments are accurately classified based on land use types, which is essential for conducting precise LU analyses.

This thesis will demonstrate future scenario analyses by examining two stormwater ponds—Matheson (0.057 km^2) and Eastgate (0.016 km^2)—currently under construction in the Cooksville Creek subwatershed (Figure 3-14). The scenario, which integrates proposed ponds into the ESM LU Analysis, highlights the utility of SPINpy 2 in pre-construction planning. Furthermore, the impacts of these new stormwater management ponds can be simulated in SPINpy 2 using spatial pond data to assess effectiveness in mitigating erosion and managing stormwater runoff. Such analyses are vital for making informed decisions about urban development and environmental management strategies. Incorporating stormwater management ponds into SPINpy 2's Land Use analysis enhances predictive capabilities and supports sustainable watershed management practices, ensuring effective erosion control.

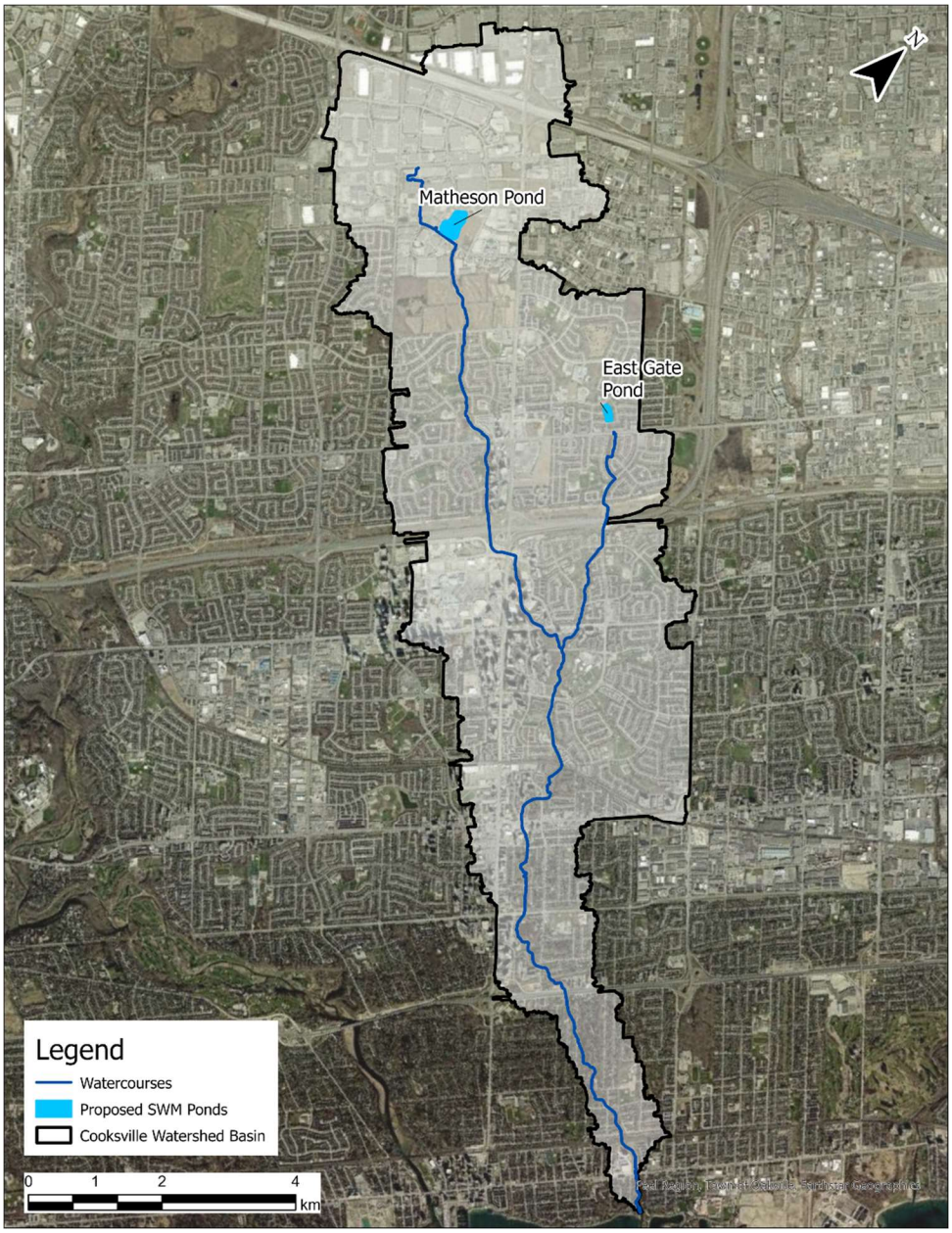


Figure 3-14: Cooksville Creek Future Pond Addition Scenario

3.3 Conclusion

Chapter 3 establishes the study subwatersheds within the CRW where SPINpy 2 will be applied and discussed in this thesis. From Cooksville Creek's highly urbanized landscape to Levi Creek's more balanced land use distribution, these diverse settings provide a framework for displaying SPINpy 2's capabilities to assess erosion risks under different scenarios.

Chapter 4
Code Development

Chapter 4 will present the methods used to consolidate the advantages of the previous versions while addressing specific gaps for modelling urban discharge in the NSM LU Scenario in SPINpy 2. SPINpy 2 introduces a new analysis option, the ESM LU Scenario, to incorporate the benefits of stormwater management ponds. Including NSM and ESM options allows SPINpy 2 to model the hydrological and geomorphic effects of green infrastructure and traditional engineered structures, broadening its utility for urban watershed managers. Table 4-1 presents how the limitations identified in Chapter 2 informed our solutions and guided testing across various watersheds, thereby enriching our understanding of the key results obtained. Sections 4.1 to 4.4 provide the code development methods used to address the identified limitations in Table 4-1.

Table 4-1: Summary of Limitations Addressed in SPINpy 2

Section	Limitation # and Research Gap	Research Objective/Solution	Subwatersheds Applied	Key Results
		Code Development		
4.1.1	(#1) NSM LU Analysis in Python (SPINarc)	Recode the LU Scenario in Python using Whitebox Geospatial Tools.	Cooksville Creek	Raster-based GIS-Hydrological Analysis – cumulative IA, greater accuracy for IA, discharge, faster processing, and opens doors for future improvements.
4.1.1	(#1/#2) Memory limitations and significant processing times (SPINarc)	A new method called the D8 Mass Flux Algorithm in Whitebox Geospatial Tools was used.	Cooksville Creek	
4.1.2	(#3) The inability to depict any benefits from implemented stormwater management solutions	A new scenario called the ESM LU Scenario was created and tested across three CVC subwatersheds.	Loyalist Creek Levi Creek Fletchers Creek	A new empirical model for discharge in urban areas that ensures a continuous transition from rural to urban scenarios. (Equation D)
4.2	(#4) Urban widths below rural empirical widths in the Modelled Analysis (SPINarc)	Revised such that the modelled urban widths did not go below the rural empirical values.	Cooksville Creek	-
4.3	(#5) Inability to run the analysis for multiple subwatersheds within a boundary analysis.	Added new analysis type for user-specified boundaries.	Serson Creek Applewood Creek	Analysis is beneficial to identify areas of interest within Cities, municipalities, wards to focus infrastructure budgets and restoration efforts.
		Equation Calibration and Validation		
4.4	(#6) LU NSM urban discharge equation limitations (SPINarc)	Revisions were made to the LU NSM urban discharge equation and tested across eight subwatersheds in the CVC.	Turtle Creek Serson Creek Birchwood Creek Applewood Creek Loyalist Creek Levi Creek Cooksville Creek Fletchers Creek	New empirical model for discharge in urban areas that ensures a continuous transition from rural to urban scenarios. This equation considers the presence of ponds (Equation G)
Appendix A	(#7) The LU Analysis computes a single-width estimate. Equations 2-11 for both rural and urban scenarios overlook channel enlargement due to increased flows from urbanization and will overestimate SSPR.	Calibrate a new width equation that represents the theoretical channel expansion expected with urbanization.	CVC Wide	Appendix A details the calibration methods used to address Limitation 7 utilizing field data for the Credit River provided by CVC to refine the model of urban width changes in response to urbanization.

4.1 Limitations 1-3: Land Use Analysis Revisions

4.1.1 Limitations 1 & 2: NSM Land Use Scenario Development

The first addition to SPINpy 2 was the ability to execute the NSM LU Scenario directly in Python using a modified algorithm, providing users with enhanced functionality. The first task involved rebuilding the LU Analysis using Python, which allows SPINpy 2 to calculate stream power with land use information. To rebuild the LU Analysis, SPINpy 2 leveraged Whitebox Geospatial Tools to process DEMs across all scenarios, extracting vital data like flow accumulation, flow direction, stream network, slope, and IA outputs faster and without memory restrictions. A critical component of the LU Analysis is obtaining an IA value, which is essential for the discharge equation. The IA value can be extracted using the D8 Mass Flux method in Whitebox Geospatial Tools. The D8 Mass Flux algorithm, based on a D8 Method, calculates flow direction by identifying the lowest elevation among neighbouring cells. Each cell's load value accumulates along the drainage area, representing its imperviousness. In SPINpy 2, the load value represents the imperviousness of each cell, ranging from 0 (completely pervious) to 1 (completely impervious) accumulated along the drainage area. The next step in the SPINpy 2 methodology is to find the ratio between D8 Mass Flux and Flow Accumulation rasters to obtain IA values at each cell.

Outflowing Mass = (Loading - Absorption + Inflowing Mass) × Efficiency (Lindsay, 2016)

(Equation 4-1)

The D8 Mass Flux algorithm calculates flow direction and accumulates outflowing mass values across cells (Equation 4-2). Like the D8 Mass Flux, the Flow Accumulation provides a count of total cells upstream to a specific point. The ratio between the D8 Mass Flux and the Flow Accumulation rasters extracts the IA value at each cell. The subsequent empirical equations in the stream power analysis use the extracted IA to calculate discharge, total stream power, and specific stream power. The D8 Mass Flux algorithm has replaced the polygon method in SPINarc, improving processing speed and eliminating the need for costly ESRI licenses, making SPINpy 2 an efficient open-source tool for processing large, high-resolution watersheds.

4.1.2 Limitation 3: ESM Land Use Scenario Development

The next limitation addressed is incorporating a scenario that reflected the presence of engineered stormwater management solutions, particularly stormwater management ponds. Building upon the

foundation set by SPINpy 2's NSM Scenario, the ESM Scenario introduced the ability to simulate the effects of ponds to reduce discharge rates and erosion risks using empirical discharge equations. This approach aligned with the need for more sophisticated urban hydrological models to model the impacts of urbanization on watershed management, channel stability, and erosion control.

The ESM Scenario uses three (3) subwatersheds within the CVC for modelling and calibration due to the availability of pond data. The modelling process for stormwater management ponds in SPINpy 2 included several detailed steps. First, utilizing Whitebox Tools, SPINpy 2 preprocessed the Digital Elevation Model (DEM) by applying the Flatten Lakes Algorithm, which delineates pond boundaries and adjusts elevations to simulate water storage and flow dynamics. The modelling process included determining the pond outlet by finding the maximum flow accumulation within the pond boundary. Once the pond outlet was determined, the drainage areas entering the pond are calculated using the Whitebox Geospatial Watershed tool (Lindsay, 2016). Like the NSM Scenario, the ESM Scenario's application of the D8 Mass Flux approach calculates the impervious percentage by considering various factors such as land use and topography. The calculation involves using an efficiency raster as input to model the pond's benefits accurately. In the ESM Scenario, setting the efficiency at the pond outlet to 0 results in 0% imperviousness at that specific point. As water flows downstream from the pond outlet, D8 Mass Flux accumulates again from 0, reflecting a reduction in IA due to the presence of the pond and simulating a decrease in discharge. The D8 Mass Flux equation for modelling stormwater management ponds closely resembles the NSM Scenario, however, the efficiency raster was revised for the engineered ponds scenario. The efficiency raster is populated with the outlet's location set to 0 and all other cells set to 1. This represents the discharge efficiency, where 0 represents the presence of a pond and signifies that the pond reduces the discharge quantity. After executing the D8 Mass Flux Algorithm with the revised efficiency raster, the IA is extracted and used to calculate the discharge (Q), total stream power (SP), and specific stream power (SSP).

By reconfiguring the efficiency raster and applying the D8 Mass Flux Algorithm, SPINpy 2 adjusted the IA to reflect the reduced discharge from the pond outlet. This reduction in IA demonstrated a controlled discharge environment downstream from the pond, simulating a decrease in discharge due to stormwater management. The IA output is used in the ESM empirical equations to calculate discharge, total stream power, and specific stream power that displays the benefits of having ponds. This capacity for modelling ESM Scenarios at a high level makes SPINpy 2 a valuable tool for decision-making.

4.2 Limitation 4: Modelled Scenario Revisions

The next limitation was that the Modelled Analysis periodically output urban width values below the rural widths. Therefore, SPINpy 2's Modelled Analysis was adjusted so channel widths did not fall below the rural empirical values.

Modelled data from Mill and Monora Creeks in Orangeville were used in a separate study to evaluate the SPINarc application. The erosion module's results within SPINarc were compared to existing data on these subwatersheds. The study recommended procedural updates for rural conditions, stating that "channel width in urban models should not be allowed to go below the rural empirical width (W_{PD}). Allowing it to do so can lead to predictions of high SSPR values, likely due to local channel fluctuations" (MacVicar, 2022). Revisions were applied to the SPINpy 2 Modelled and LU Analysis to ensure that urban widths do not fall below the rural (W_{PD}) widths. If the modelled width was lower than the W_{PD} rural empirical width (Equation 2-5), the W_{PD} value was used instead. Similarly, when the LU Analysis urban width was lower than the W_{PD}, the larger of the two was used.

4.3 Limitation 5: Multiple Watersheds within a Boundary Analysis

The next limitation addressed in SPINpy 2 was the inability to analyze multiple watersheds. This was necessary when conducting analyses for municipal, political, or regional boundaries. The new Multiple Boundary Analysis allowed users to obtain results for multiple watersheds within a user-specified boundary. SPINpy 2 recognizes a boundary to analyze and execute the algorithm for the number of watersheds within or intersecting the boundary.

When a boundary was defined, the program identified and processed all intersecting watersheds using the same base algorithm as the single watershed scenario. Next, the revised script dissolved the intersecting watersheds into one basin in the LU Analysis, then analyzed as a single watershed. In the Modelled Analysis, SPINpy 2 organizes input data for each watershed into separate folders, then iterates through each watershed, analyzing each watershed individually and combining the results into a single shapefile.

SPINpy 2 automatically calculates critical hydrological and geomorphological metrics such as discharge, stream power, channel width, and specific stream power to identify erosion hotspots. By consolidating these metrics across multiple watersheds, the Multiple Boundary Analysis provides an

overview of watershed dynamics, supporting informed decision-making for resource allocation and environmental protection.

4.4 Limitation 6: Equation Calibration and Validation

4.4.1 Calibration Data Preparation

The final limitation addressed in SPINpy 2 is the calibration of the empirical discharge equations used in the NSM and ESM LU Scenarios. The CVC detailed hydrologic data was used as calibration data. Figure 4-1 and Figure 4-2 illustrate the drainage area and imperviousness distribution by subwatershed for the NSM and ESM LU Scenario study sites, respectively. The NSM data has imperviousness ranging from 0-67%, with most of the data falling within the 27-40% range. Additionally, the drainage areas of the calibration data range from 0-40 km^2, with most of the data falling within the 0-5 km^2 range.

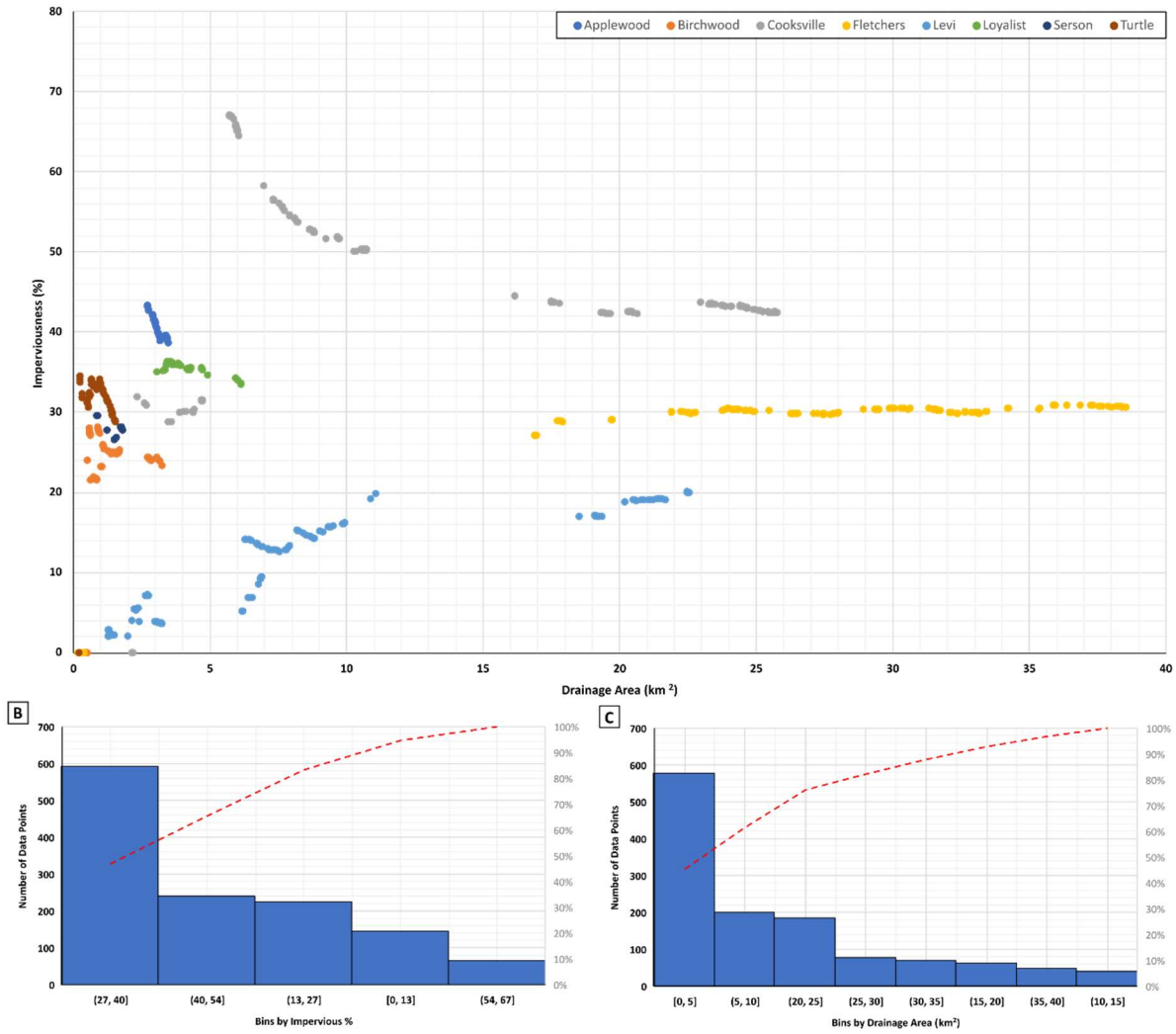


Figure 4-1: NSM Drainage Area and Imperviousness by Subwatershed A) Impervious percent compared to the drainage area for all data points B) Pareto chart depicting the distribution of the impervious percent data C) Pareto chart depicting the distribution of the drainage area bins.

The ESM data has imperviousness ranging from 0-42%, with most of the data falling within the 26-34% range. Additionally, the drainage areas of the calibration data range from 0-40 km^2, with most of the data falling within the 0-5 km^2 range.

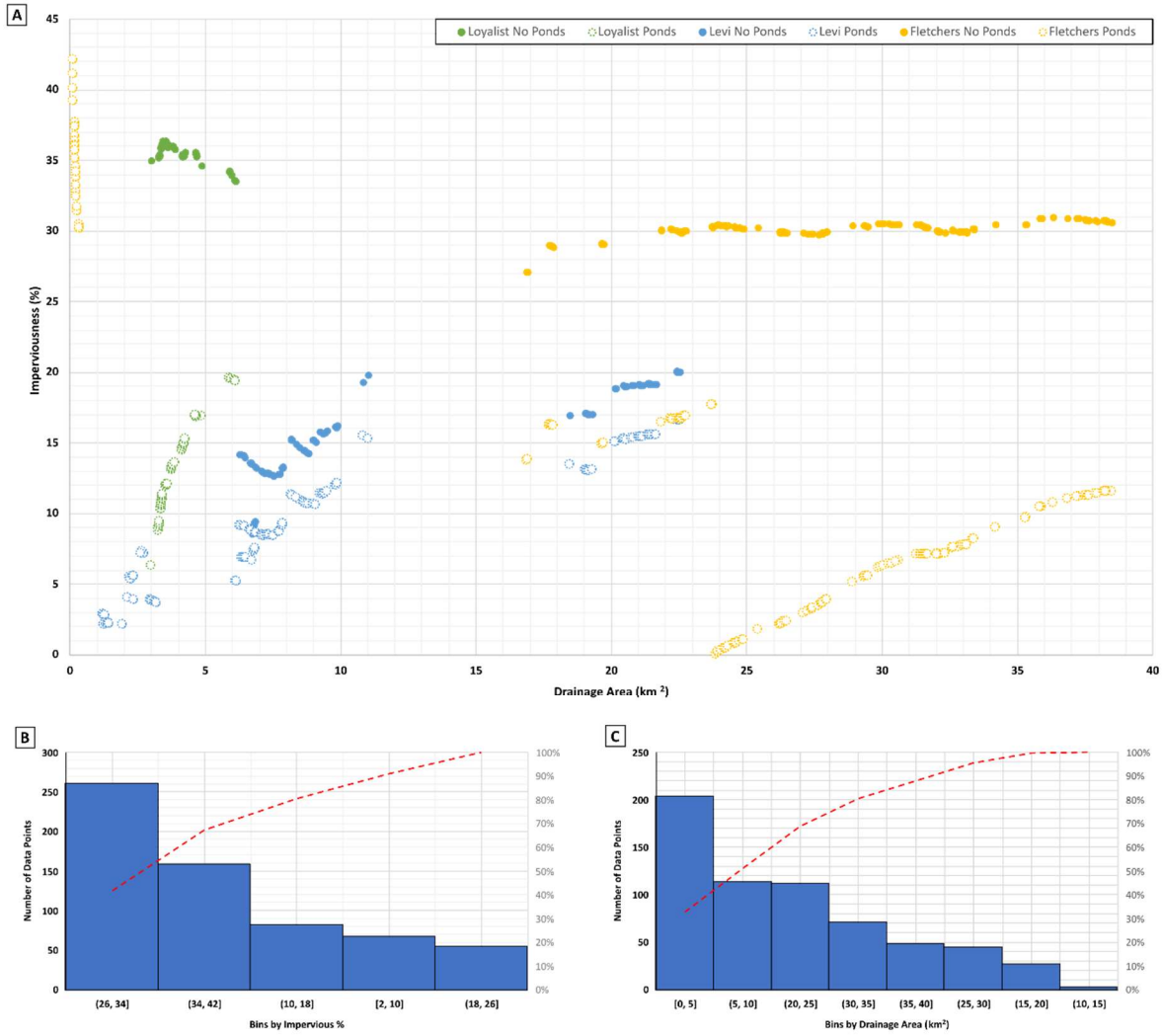


Figure 4-2: ESM Drainage Area and Imperviousness by Subwatershed A) Impervious percent compared to the drainage area for all data points B) Pareto chart depicting the distribution of the impervious percent data C) Pareto chart depicting the distribution of the drainage area bins.

A training dataset was prepared using 20 points from each subwatershed randomly selected using the "slice_sample" function from the "dplyr package" in R Studio to prepare the data for calibration. This training dataset development method ensures that each subwatershed is equally represented and the data is not skewed to a subwatershed with a larger number of data points. An R-script was prepared for a non-linear regression analysis using the training dataset to determine coefficients for each discharge model considered. This R-script facilitates analysis and visualization of the

relationship between variables in the discharge equation. This calibration is crucial for establishing the urban discharge equation relationships within the LU Analysis and provides coefficients for the NSM and ESM discharge equations.

The R-script applies nonlinear regression techniques for optimization to find the best coefficients that minimize the sum of squared residuals (SSR), thereby improving the model's fit to the provided data. The script then extracts variables such as 'Drainage Area (DA)', 'Imperviousness (IA)', 'Discharge (Q)', and coefficients from the Excel detailed hydrologic data for the eight (8) study subwatersheds. Utilizing a non-linear regression model, the script determines the optimal coefficients for the discharge equation model based on the potential coefficients in the inputted Excel data. Following optimization, the script computes predicted discharge using the optimal coefficients and outputs measurement of model fit (SSR, NSE, and PBIAS). In other words, the non-linear regression analysis will determine how well the calibrated models fit the data and the level of uncertainty present in each model.

4.4.2 SPINpy 2 NSM LU Urban Discharge

4.4.2.1 NSM LU Urban Discharge Equation Calibration Methods

Previously. Equation 2-7 was used as the urban discharge relationship. However, Equation 2-7 is untested and uncalibrated, thus it was included as Equation A for testing. The following equations in Table 4-2 were calibrated to suit the CVC watershed conditions. To verify that the calibrated discharge equations in SPINpy 2 are accurate, modelled data provided by the CVC will be used as validation data.

Table 4-2: Equations Considered for NSM Discharge Equation Calibration

	Equation
A	$Q=0.248(DA)^{0.91}(IA)^{0.30}$
B	$Q = c(DA)^{d}(IA)^{e}$
C	$Q = f(DA(1-IA))^{g} + h(DA(IA))^{i}$
D	$Q = c(DA(1-IA))^{d} + j(DA(IA))^{k}$

Equation A, where DA is in km^2, and IA is a percent, is the original equation in the LU Analysis of SPINarc proposed by Ghunowa et al. (2021). The urban scenario is predicted by applying a relative imperviousness factor to the empirical rural relationship between drainage area and discharge (Equation 2-6). The values of c and d are regional coefficients obtained from Phillip and Desloges's (2014) study of specific stream power in rural catchments in the Southern Laurentian Great Lakes. The "e" coefficient in the original equation is not calibrated and is based on an average of typical coefficient values recorded in a study by Bledsoe and Watson (2001). Equation B, where DA is in km^2, and IA is expressed as a percent. Equation B calibrates the "e" coefficient of Equation A to reflect the hydrology of the CRW. Equation C, where DA is in km^2, and IA is a fraction, consider that flows are additive between impervious and pervious areas in a watershed. The coefficients f,g,h, and i are calibrated for the eight (8) study subwatersheds in the CRW. Equation D follows the same additive format as Equation C, however, coefficients c and d are regional coefficients based on Phillip and Desloges (2014), and j and k are calibrated for eight (8) subwatersheds in the CRW. When imperviousness is close to zero (0), this equation behaves more like the rural empirical equation (Phillips & Desloges, 2014). Additionally, the additive nature of flows from impervious and pervious areas (assuming peaks occur simultaneously), improved the predictions for urban watersheds (Deng et al., 2020; Urban Drainage and Flood Control District, 2014).

4.4.2.2 SPINpy 2 NSM LU Urban Discharge Calibration Results

A non-linear regression analysis was executed in R-script to determine the coefficients for each discharge model considered. The calibration dataset was comprised of 160 records (20 from each available subwatershed).

Table 4-3: Discharge Equation Calibration Results

Equation	Calibrated Coefficient
A	$Q=0.248(DA)^{0.91}(IA)^{0.30}$
B	$Q = 0.248(DA)^{0.91}(IA)^{0.64}$
C	$Q = 0.176(DA(1-IA))^{-0.34} + 11.50(DA(IA))^{0.61}$
D	$Q = 0.248(DA(1-IA))^{0.91} + 11.08(DA(IA))^{0.60}$

Each equation was plotted with the calibrated coefficients using the training dataset, which had an average modelled discharge of 15.82 m^3/s. Statistics were calculated based on the outputs. The

statistics presented to determine the best model are the Mean, Standard Error of the Regression (S), Nash Sutcliffe Efficiency (NSE), Percent Bias (PBIAS), Kling Gupta Efficiency (KGE) and Akaike Information Criterion (AIC) (Table 4-4).

Table 4-4: Summary of Statistics for the Calibrated Equation Using Training Dataset

Statistic Method	Equation A	Equation B	Equation C	Equation D
Mean	4.00	12.98	15.98	15.97
S	17.57	9.61	7.54	7.92
NSE	-0.12	0.66	0.80	0.77
PBIAS	-74.74	-17.95	1.01	0.95
KGE	-0.04	0.77	0.84	0.83
AIC	5342.18	5342.18	9104.12	4088.24

In the calibration and statistical comparison of discharge models, Equation D emerged as the most suitable for urban watershed applications due to its balance of statistical accuracy and practical adaptability. With a mean discharge of 15.97 m^3/s, Equation D closely aligns with the training dataset's average modelled discharge of 15.82 m^3/s, unlike Equations A and B, which significantly underpredict the discharge. Although Equation C has the lowest standard error of the regression (S = 7.54) and the highest Nash-Sutcliffe Efficiency (NSE = 0.80), Equation D performs comparably (S = 7.92, NSE = 0.77) while offering critical advantages in practical settings. Equation D's Percent Bias (PBIAS = 0.95) and Akaike Information Criterion (AIC = 4088.24) reflect a minimal bias and optimal model complexity, respectively, affirming its reliability without overfitting. Furthermore, Equation D's KGE (KGE = 0.83) indicates strong performance in capturing both the correlation and variability in discharge, making it adaptable to urban watershed nuances.

Equation D is adaptable when imperviousness approaches zero, where it behaves similarly to the widely accepted rural empirical equation by Phillips & Desloges (2014). This characteristic is beneficial, as it allows Equation D to transition seamlessly to rural-like discharge behaviour in low impervious settings, consistent with the rural empirical model already integrated within SPINpy 2. By aligning with this established model, Equation D provides a cohesive and versatile solution, simplifying the application of a single calibrated model across urban and rural watershed types. This continuity enables SPINpy 2 users to model complex urban-rural gradients accurately while

maintaining consistency with widely recognized rural watershed behaviour. Collectively, these attributes make Equation D the most suitable choice for urban watershed modelling, offering empirical precision, low bias, and robust adaptability across varying imperviousness conditions.

4.4.3 SPINpy 2 ESM LU Urban Discharge

4.4.3.1 SPINpy 2 ESM LU Urban Discharge Calibration Method

Detailed hydrologic models and available gauging stations are used to calibrate and validate the ESM LU Urban Discharge equations. Three (3) subwatersheds within the CVC were used to build the ESM Scenario and calibrate an urban discharge equation that illustrates the characteristics of having stormwater management (engineered or nature-based) controls. The CVC has provided discharge and width outputs from their models and gauge data to be used as calibration data for the urban discharge equation. The following equations were calibrated to better suit the CVC watersheds' conditions.

Table 4-5: Equations Considered for Calibration

Equation	Calibrated Coefficient
G	$Q = c(DA(1\text{-}IA_{SWM}))^d + m(DA(IA_{SWM}))^n$
H	$Q = c[DA((1 - IA_{NOSWM}))]^d - p\left[DA\left(\frac{SWMDA}{DA}\right)\right]^q + j[DA(IA_{NOSWM})]^k$
I	$Q = c(DA(1\text{-}IA_{SWM}))^d + j(DA(IA_{SWM}))^k$

Where IA_{SWM} is the impervious percent extracted from the ESM D8 Mass Flux Algorithm, IA_{NOSWM} is the impervious percent extracted from the NSM D8 Mass Flux Algorithm, DA is the drainage area upstream of the stream segment of interest, and SWMDA is the drainage area contributing to the pond.

The structure of Equation G is the same concept as that used for the NSM discharge equation. However, the ESM discharge equation will have calibrated coefficients m and n to reflect the conditions of the three (3) subwatersheds with detailed hydrologic modelled data with ponds. In Equation G, IA_{SWM} is the output from the D8 Mass Flux that accounts for the ponds by setting the efficiency at the outlet to 0. The strategy used includes identifying the outlet of each pond using raster processing techniques. Equation G offers a refined understanding of surface runoff behaviour on varying surfaces. The equation represents the effect of stormwater management ponds on urban hydrology and uncertainty in the simple equation. Where DA is expressed in km^2, and IA_{SWM} is the

imperviousness value extracted from the LU Analysis D8 Mass Flux Algorithm that has stormwater management ponds. When imperviousness is close to 0, this equation behaves more like the rural empirical equation (Phillips & Desloges, 2014). Equation H is the same as the NSM Scenario discharge equation, however, it has an additional term that subtracts the ratio of the drainage area contributing directly to the pond and the overall drainage area upstream of the outlet. The ratio multiplied by the IA_{NOSWM} provides insight into how much discharge is reduced by flowing into the pond. Equation I is the same coefficients as the discharge in the NSM Scenario.

4.4.3.2 SPINpy 2 ESM LU Urban Discharge Calibration Results

A non-linear regression analysis was executed in R-script to determine each ESM Scenario discharge model's coefficients. All data points recorded in the detailed hydrologic models for three (3) subwatersheds are used for calibration.

Table 4-6: Discharge Equation Calibration Results

Equation	Calibrated Coefficient
G	$Q = 0.248(DA(1\text{-}IA_{SWM}))^{0.91} + 9.39(DA(IA_{SWM}))^{0.25}$
H	$Q = 0.248[DA((1 - IA_{NOSWM}))]^{0.91} - 9.60\left[DA\left(\frac{SWMDA}{DA}\right)\right]^{0.34} + 11.08[DA(IA_{NOSWM})]^{0.60}$
I	$Q = 0.248(DA(1\text{-}IA_{SWM}))^{0.91} + 11.08(DA(IA_{SWM}))^{0.60}$

Each equation was plotted with the calibrated coefficients using the training dataset with an average discharge of 11.84 m^3/s.

Table 4-7: Summary of Statistics for the Calibrated Equation (Training Data)

Statistic Method	Equation G	Equation H	Equation I
Mean	11.83	12.06	15.02
S	4.70	9.33	7.85
NSE	0.52	-0.88	-0.33
PBIAS	-0.12	1.80	26.83
KGE	0.57	0.04	0.39
AIC	341.79	291.08	366.76

Based on the statistical comparison, G is the most suitable for urban watershed discharge simulation in scenarios with stormwater management (SWM) ponds. Its Nash-Sutcliffe Efficiency (NSE = 0.52) and Kling-Gupta Efficiency (KGE = 0.57) indicate moderate performance, while its minimal Percent Bias (PBIAS = -0.12) reflects low deviation from observed values. Furthermore, Equation G's standard error (S = 4.70) is the smallest, ensuring a reliable representation of discharge variability. In contrast, Equation H exhibits poor performance, with a negative NSE (-0.88) and the lowest KGE (0.04). These statistics reveal its inability to replicate observed discharge patterns accurately. This model also displayed instances of negative discharge predictions, highlighting the necessity of revisiting assumptions regarding the contribution of SWM pond drainage areas (SWMDA) in smaller tributaries. While Equation I demonstrates slightly improved correlation metrics (KGE = 0.39) compared to Equation H, its high PBIAS (26.83) suggests significant overprediction of discharge, especially in regions with high imperviousness. Overall, Equation G strikes the best balance among the tested models, offering practical reliability and predictive accuracy under urban watershed conditions with SWM infrastructure. These findings highlight the importance of tailoring discharge equations to account for urbanization patterns and drainage characteristics. Validation using the CVC gauge data for the ESM discharge equation is presented in Appendix A.

4.5 Conclusion

SPINpy 2 can guide infrastructure upgrades by estimating the effectiveness of NbS in managing erosion and flooding, as it integrates hydrological data with erosion models to provide valuable insights for informed urban planning and infrastructure development. The revised SPINpy 2 incorporates improved algorithms and empirical equations tailored for various scenarios, accommodating different watershed contexts. Within a single watershed framework, SPINpy 2 supports NSM and ESM LU Analyses and the Modelled Analysis, which leverages detailed hydrologic data. This flexibility ensures that SPINpy 2 can model erosion risks accurately across diverse land use conditions. Users can apply the same analyses to multiple watersheds in a Multiple Boundary Analysis, simulating cumulative regional impacts. This approach ensures reliable risk assessments in both single- and multiple-watershed applications.

Updating the tool from SPINarc to SPINpy 2 has streamlined the workflow, reducing the process from five tools in succession to one uninterrupted tool. SPINpy 2 can now run a continuous stream power analysis with inputs of a hydro-corrected DEM, LU Shapefile, typical impervious values for

each land use type, and coordinates of the watercourses of interest. The new ESM LU Scenario requires only a shapefile of stormwater management ponds, with an optional boundary input for analyzing multiple watersheds. Additionally, stormwater management infrastructure, particularly ponds, has been integrated into the discharge equations for more accurate modelling of engineered solutions in urban watersheds. Figure 4-3 and Table 4-8 illustrate the different scenarios and associated inputs.

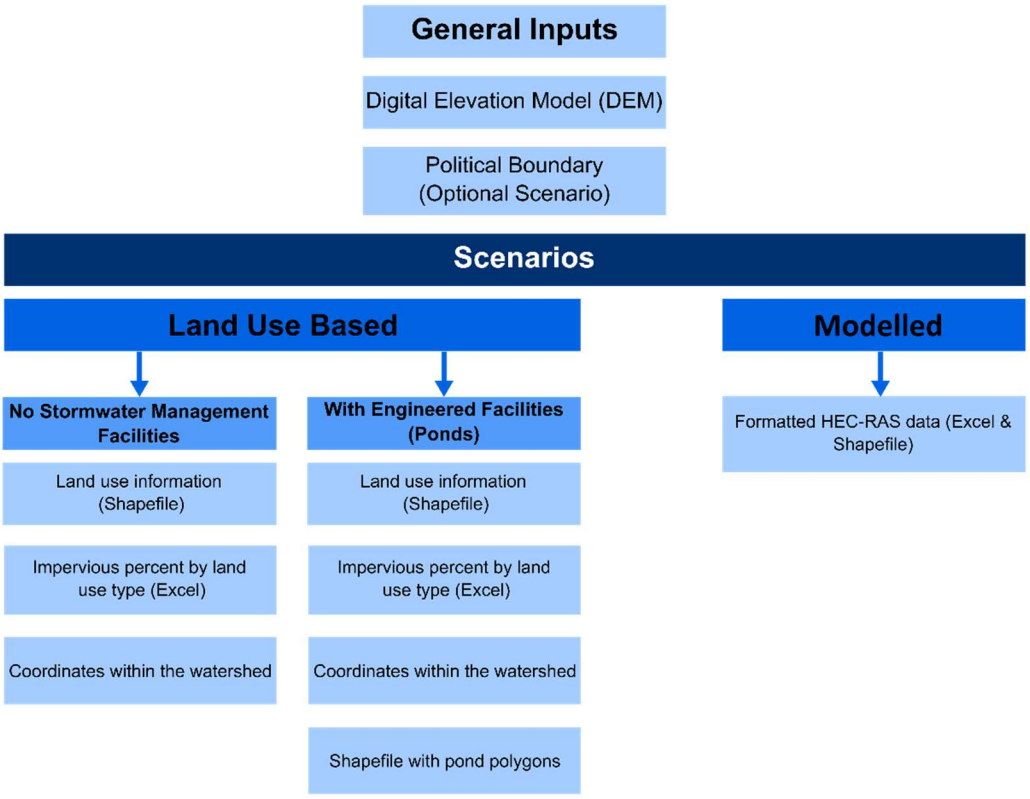


Figure 4-3: SPINpy 2 Possible Scenarios

Table 4-8: SPINarc, SPINpy, and SPINpy 2 program inputs based on type of analysis

SPINarc	SPINpy	SPINpy 2
Modelled Analysis		
Hydro-corrected DEM	Hydro-corrected DEM	Hydro-corrected DEM
Detailed hydrologic model outputs (Excel)	Detailed hydrologic model outputs (Excel)	Detailed hydrologic model outputs (Excel)
Hydrologic model stations (Shapefile)	Hydrologic model stations (Shapefile)	Hydrologic model stations (Shapefile)
Stream Network Delineation	-	-
Stream Segmentation	-	-
Slope Smoothing	-	-
Rural Stream Power Outputs	-	-
Connect Data to Segments	-	-
Data Interpolation	-	-
Urban Stream Power Analysis	-	-
Land Use Analysis		
Hydro-corrected DEM	-	Hydro-corrected DEM
Land Use Shapefile	-	Land Use Shapefile
Stream Network Delineation	-	Coordinates of interested watercourses.
Stream Segmentation	-	Typical impervious values for unique land use types (Excel)
Slope Smoothing	-	Boundary (Optional)
Rural Stream Power Outputs	-	SWM Ponds Shapefile (Optional)
Drainage Area Polygonized	-	-
Urban Stream Power Analysis	-	-

In Chapter 5, SPINpy 2 will be applied to the study sites introduced in Chapter 3, allowing for verification of results and highlighting areas with high erosion risks in urban watersheds. By focusing on the real-world application to select study areas in the CVC, Chapter 5 will demonstrate SPINpy 2's effectiveness through case-specific analysis.

Chapter 5

Results of Applying SPINpy 2 to Cooksville Creek in the CVC Watershed

This chapter presents the quantitative results of implementing SPINpy 2 within select subwatersheds in the CRW. The primary focus will be Cooksville Creek's NSM outputs and hypothetical future ESM Scenarios. Cooksville Creek's outputs showcase SPINpy 2's ability to model discharge, channel morphology, and hydrologic metrics with greater precision. The inputs for each study subwatershed are shown in Chapter 3, and the outputs for Cooksville Creek are shown and discussed below. All other results for select case studies presented in Chapter 3 are in Appendix B. Outputs are compared across four scenarios: a rural baseline, urban conditions with detailed hydrologic modelled data, urban conditions with SPINpy 2 NSM Scenario, and a hypothetical ESM scenario incorporating two additional ponds.

Figure 5-1 highlights discharge variations, showing substantial increases in urban scenarios (Figure 5-1B) compared to the rural baseline (Figure 5-1A). Recalibrated equations in the NSM Scenario (Figure 5-1C) align more closely with observed data. In contrast, the ESM Scenario (Figure 5-1D) demonstrates a reduction in discharge at the outlet by 59%, reflecting the mitigating influence of two additional ponds. Figure 5-2 extends this analysis to total SP, with urbanization amplifying SP outputs (Figure 5-2B) compared to rural conditions (Figure 5-2A). The ESM Scenario (Figure 5-2D) reduces SP by 59% at the outlet, demonstrating the ponds' effectiveness in dissipating stream energy. Figure 5-3 focuses on SSP, emphasizing localized erosion potential per unit area. SSP increases significantly in urbanized scenarios (Figure 5-3B) due to narrower channels, but the ESM Scenario (Figure 5-3D) reduces SSP by 58% at the outlet due to upstream ponds. Finally, Figure 5-4 examines SSPR, showing how recalibrated NSM Scenario discharge equations (Figure 5-4B) better capture urban dynamics, with the ESM Scenario (Figure 5-4C) further reducing SSPR, which indicates less erosive conditions due to the impact of ponds reducing peak flows upstream.

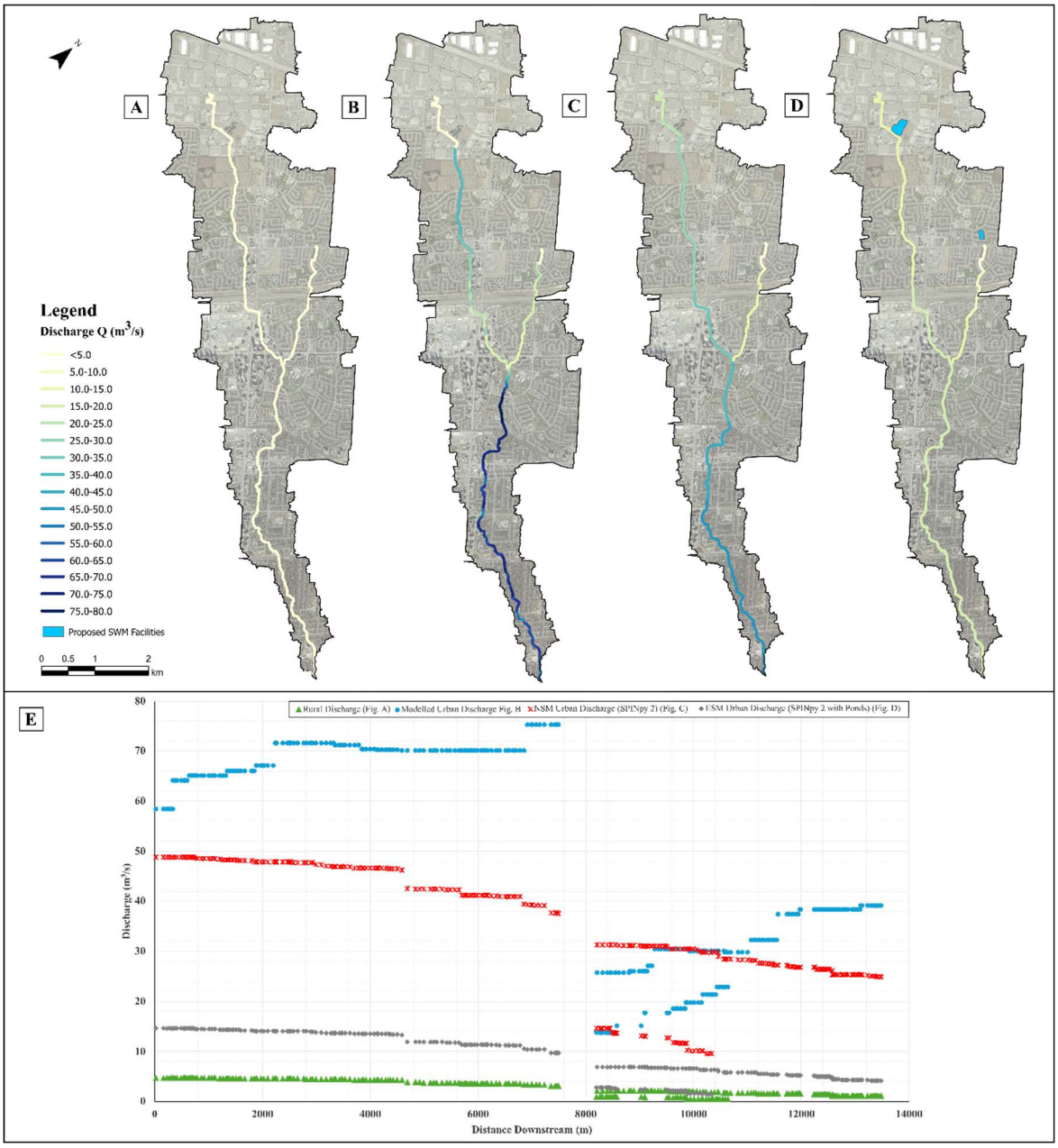


Figure 5-1: Cooksville Creek Rural and Urban NSM and ESM Discharge Outputs A) Rural discharge as a baseline B) Urban modelled discharge using SPINpy 2 Modelled Analysis C) Urban land use discharge with the recalibrated equations for SPINpy 2 in the NSM Scenario D) Urban land use discharge (ESM Scenario) under a hypothetical future scenario of two additional ponds in Cooksville E) Comparison between rural and urban discharges across scenarios A-D in comparison to the distance downstream, where 0 represents the outlet to Lake Ontario. Figure 5-1E highlights the recalibration of the NSM urban discharge equation, having outputs that predict values closer to the modelled values and the potential of the ESM Scenario to model the reduction of discharge due to the presence of upstream ponds.

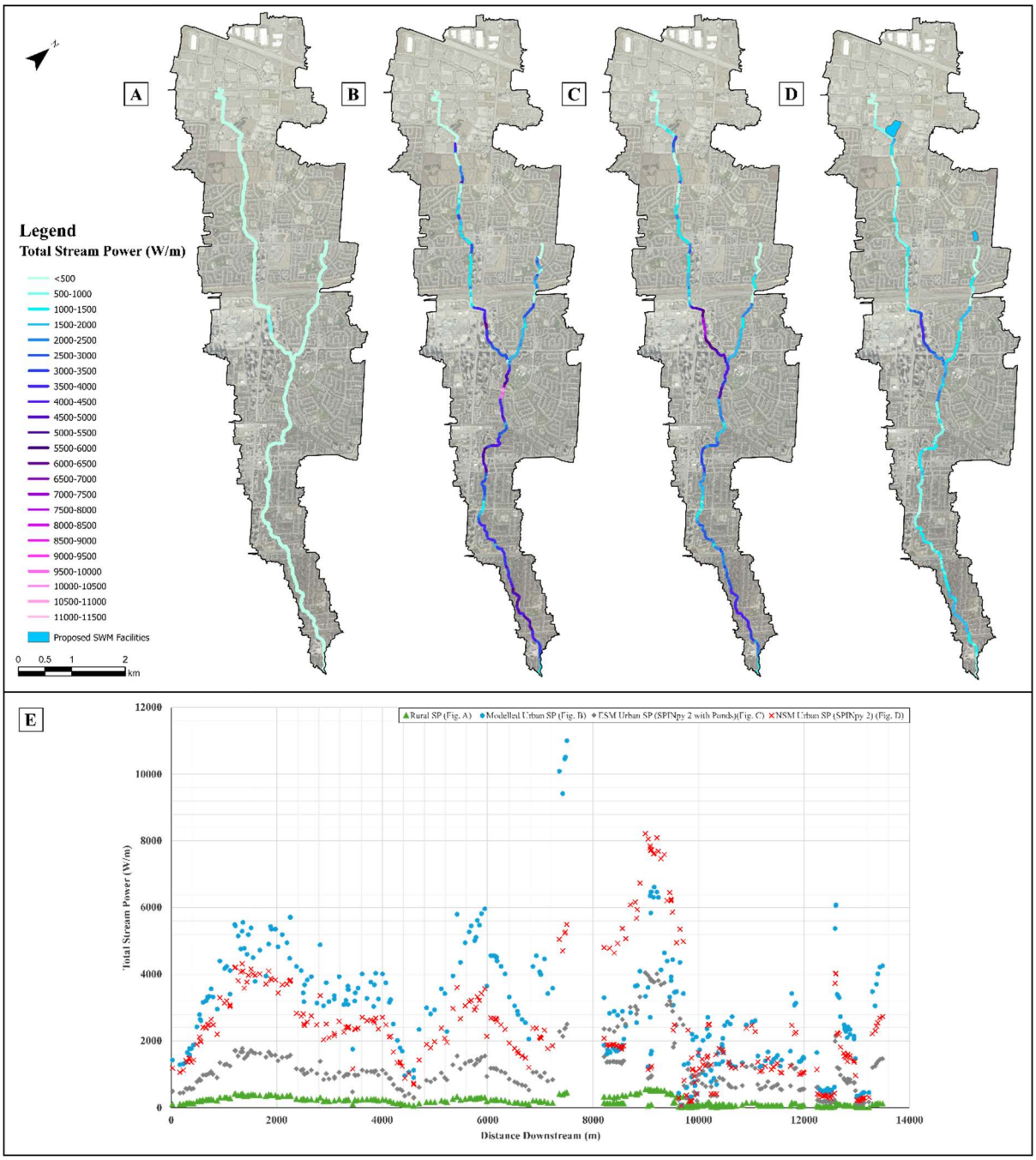


Figure 5-2: Cooksville Creek NSM and ESM SP Outputs A) Rural SP B) Urban modelled SP C) Urban land use SP with the recalibrated equations for SPINpy 2 in the NSM Scenario D) Urban land use SP (ESM Scenario) under a hypothetical future scenario of two additional ponds in Cooksville E) Comparison of the rural, urban modelled, and land use SP outputs. Similar to Figure 5-1E, Figure 5-2E illustrates the new NSM discharge equation's ability to capture SP values similar to the modelled outputs.

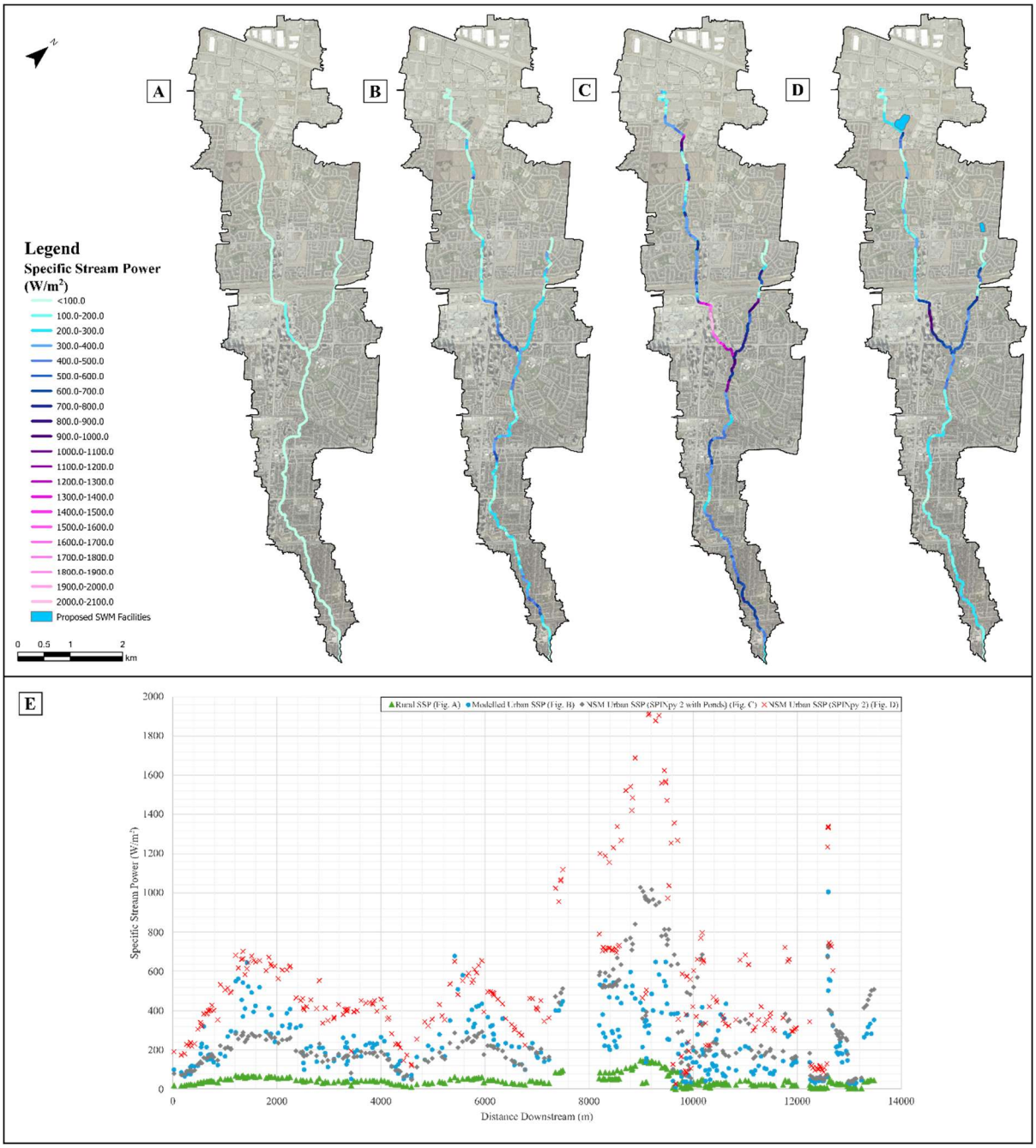


Figure 5-3: Cooksville Creek NSM and ESM SSP Outputs A) Rural SSP B) Urban modelled SSP C) Urban land use SSP with the recalibrated equations for SPINpy 2 in the NSM Scenario D) Urban land use SSP (ESM Scenario) under a hypothetical future scenario of two additional ponds in Cooksville E) Comparison of the rural, urban modelled, and land use SSP outputs. Figure 5-3E's NSM SSP prediction is overpredicting at a distance between 8000-10000 m from the outlet, this is primarily due to using the rural width equation, although NSM Scenario models an urban context.

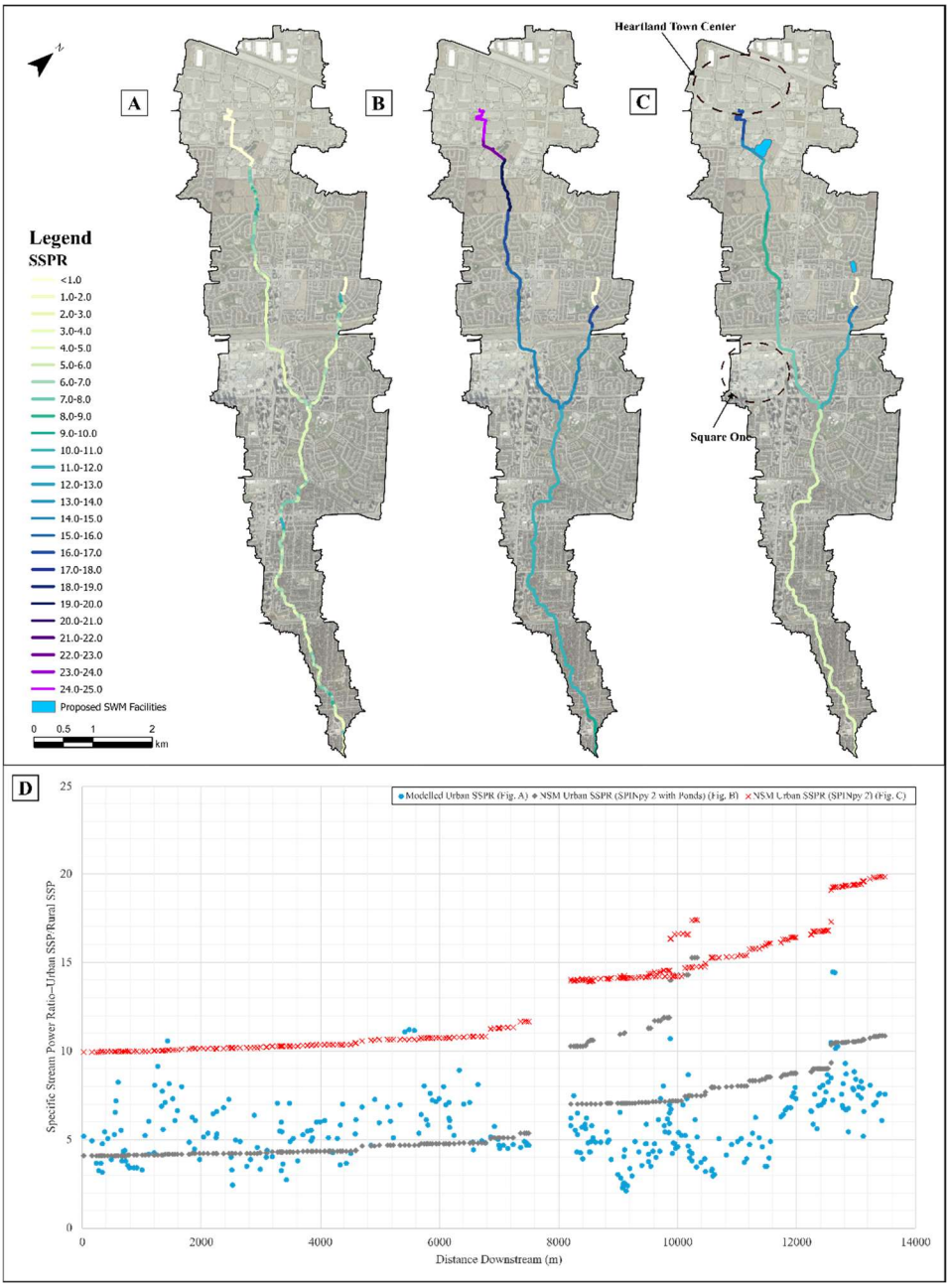


Figure 5-4: Cooksville Creek NSM and ESM SSPR Outputs A) Urban modelled SSPR B) Urban land use SSPR with recalibrated equation for SPINpy 2 in the NSM Scenario C) Urban land use SSPR (ESM Scenario) under a hypothetical future scenario of two additional ponds in Cooksville D) Presents a comparison of the SSPR values across the Modelled, NSM, and ESM Scenarios plotted against the distance downstream, where 0 represents the outlet to Lake Ontario. The SSPR values downstream of heavily urbanized areas such as Heartland Town Center show larger SSPR values; after the future pond is installed downstream, there is a notable decrease in SSPR in the ESM Scenario.

These findings underscore SPINpy 2's utility as a screening tool for urban watershed management, allowing stakeholders to evaluate the impacts of urbanization and test the potential effectiveness of mitigation strategies. The recalibrated equations enhance its predictive accuracy, while scenario modelling demonstrates its ability to inform sustainable erosion management practices efficiently. SPINpy 2 bridges the gap between empirical studies and actionable urban water management solutions by enabling rapid, scenario-based assessments. The results demonstrate that urbanization substantially increases discharge, SP, SSP, and SSPR compared to rural baselines, intensifying erosion risks. The NSM scenario shows that recalibrated equations in SPINpy 2 effectively capture urban dynamics. For example, higher discharge, SP, and SSP are observed after heavily urbanized areas such as Heartland Town Center and Square One. Adding two ponds in the ESM scenario shows the downstream effect of reducing discharge, SP, SSP, and SSPR at the outlet. These findings demonstrate SPINpy 2’s revisions improve its ability as a rapid screening tool in watershed management to identify erosion risk.

Chapter 6
Discussion and Conclusions

This project enhances urban watershed modelling by developing SPINpy 2, which focuses on improving the tool's algorithms and empirical discharge estimates. The improvements account for land use changes driven by urbanization and NbS interventions. The enhanced discharge equations are particularly relevant for addressing the challenges of urbanization, such as increased impervious surfaces and altered flow regimes. Moreover, practitioners often visualize NbS potential to reduce peak flows and restore flow regimes using detailed modelling programs. However, given the need for preliminary planning tools, it is necessary to model NbS in a general way for watershed planning purposes. As a screening tool, SPINpy 2 provides a practical framework for rapidly assessing changes in discharge, total SP, SSP, and SSPR. Its recalibrated equations enable improved accuracy for urban contexts, offering valuable insights into the potential mitigation of erosion risks by adding NbS, such as stormwater ponds.

Applying SPINpy 2 to the select subwatersheds in the CRW demonstrated SPINpy 2's potential as a preliminary screening tool for watershed managers. The results highlight critical areas with high erosion risk that watershed managers should focus on for fieldwork and detailed studies. Additionally, potential configurations of stormwater management ponds and its impact on the whole watershed can be spatially visualized through a high-level modelling program such as SPINpy 2. These findings are crucial for urban watershed management, offering guidance for balancing hydrological stability with sustainability goals.

The following sections will discuss the application of SPINpy 2 to Cooksville Creek and the tool's limitations. Additionally, this chapter will discuss SPINpy 2's potential to contribute to urban watershed management, particularly in urbanized watersheds like the Cooksville Creek subwatershed. Finally, future work to enhance SPINpy 2 will be discussed.

6.1 SPINpy 2 Application

SPINpy 2's empirical equations effectively estimated urban metrics (discharge and SP, SSP, and SSPR) similar to the modelled data, and rural metrics aligned with the study completed by Phillips and Desloges (2014). Figures 5-1 to 5-4 quantitatively and visually depict the results, where Figure 5-1 explicitly captures the urban discharge empirical equation changes. Refining the urban discharge equations in SPINpy 2 played a critical role in capturing the hydrological dynamics of urban watersheds. The previously used discharge equation failed to accurately predict runoff patterns and streamflow responses in urbanized areas due to its inability to fully account for impervious surfaces' effects. By revising the discharge equations, this study improves model accuracy under urban conditions. The new NSM and ESM discharge equations adapt to the increased runoff and peak flows driven by urbanization, better accounting for land-use changes and imperviousness.

The current SPINpy 2 NSM urban discharge equation (Figure 5-1C) demonstrates improved performance compared to previous discharge equations, highlighting significant improvements in predicting streamflow and flood risks under urban land-use scenarios (Appendix B). Cooksville Creek is specifically heavily urbanized, and two major industrial/commercial areas are within the subwatershed (Heartland Town Center and Square One). SPINpy 2's results effectively display increased discharge at both locations (Figure 5-1C). The results of NSM total and specific stream power (Figures 5-2C and 5-3C) display high erosion-risk areas. A specific stream power of approximately 1300-1450 W/m^2 is particularly observed near the Square One development, which is highly urbanized and has existing erosion and flooding concerns (Credit Valley Conservation Authority et al., 2021).

SPINpy 2's ESM urban discharge equation is helpful in modelling NbS interventions to identify the effects of placing interventions such as stormwater management ponds. Two future ponds have been added to Cooksville Creek to demonstrate this ability. This addition changes the subwatershed's discharge, SP, SSP, and SSPR distributions, notably south of the pond on the east branch (Matheson Pond). Doing an urban pre-development to post-development simulation is a beneficial way for users to assess the downstream impact of any proposed NbS. Similarly, proposed developments can be simulated in SPINpy 2 by running the tool with different LU configurations. This is very useful for watershed managers to decide where within the watershed to build new developments and make a

sustainable decision based on a combination of screening tools such as SPINpy 2 and field visits based on the SPINpy 2 results.

6.2 Tool for Urban Watershed Management

SPINpy 2 is an open-source screening tool for watershed-scale decision support. The open-source nature of the SPINpy 2 tool allows for a widespread application and allows users to make modifications as needed. SPINpy 2 is best suited for identifying spatial and temporal patterns rather than providing precise quantitative discharge predictions. This distinction underscores the importance of using SPINpy 2 as a screening tool to guide fieldwork and infrastructure planning rather than a definitive solution for watershed modelling. This approach streamlines decision-making processes by prioritizing areas for infrastructure upgrades or additional data collection, ensuring that resources are allocated efficiently and effectively.

One of SPINpy 2's key innovations is its integration of NbS, allowing watershed managers to evaluate the potential of stormwater management ponds to mitigate the effects of urbanization. By modelling the effects of NbS interventions, SPINpy 2 provides critical insights into water retention, infiltration, and overall hydrological stability. This capability promotes sustainable urban watershed management and supports strategies aligned with long-term environmental and ecological goals (Booth et al., 2002; Poff, 2002; Poff et al., 2006).

SPINpy 2 also supports proactive planning and collaboration by modelling erosion sensitivity at the watershed scale. This feature enables managers to anticipate future hotspots of instability and prioritize interventions accordingly. Additionally, its ability to visualize areas at risk of instability aids in guiding stream management efforts, such as controlling runoff, sediment transport, and channel degradation (Vocal Ferencevic & Ashmore, 2012). By considering broader network-scale impacts of urbanization, SPINpy 2 extends its relevance beyond localized issues, helping planners evaluate the consequences of land-use changes. This holistic perspective supports sustainable urban growth by balancing economic development with environmental protection.

Furthermore, the tool's outputs provide critical data for designing infrastructure projects and stormwater management policies addressing the challenges of urbanization and climate change. SPINpy 2's ability to refine traditional modelling approaches, integrate NbS, and accurately predict geomorphological changes in urban watersheds makes it an essential tool for sustainable watershed management.

6.3 Limitations of this Study/Future Research

While SPINpy 2 offers significant advancements over traditional models, it has limitations. One limitation is the sensitivity of the model to input data, such as land-use classifications and impervious percentages. These factors can introduce uncertainties in predictions, particularly when applying the model to regions outside the calibration dataset. The study's geographic focus on the CRW limits the broader applicability of the results. SPINpy 2's performance in other urbanized watersheds may vary, and further validation is necessary to assess its generalizability. Additionally, the model's ability to account for the full range of land-use scenarios could be further improved by expanding the database of urbanized regions used for calibration.

The calibration dataset fundamentally drives SPINpy 2's discharge predictions. However, reliance on empirical equations calibrated with other modelled data introduces limitations. The assumptions and accuracy of the underlying calibration data often constrain these equations. While the new discharge equations help model the disconnection of impervious areas, the equations do not account for detailed rural and urban watershed characteristics (surficial geology, soil types, vegetation, drainage density, and density of urban development). A more robust approach would integrate geomorphic conditions, such as soil type, and utilize methods like the Curve Number (CN), which considers climate factors more comprehensively. This shift would enhance SPINpy 2's reliability for discharge estimations and update the model's calibration to account for regional differences in urbanization patterns. Future work could focus on expanding the geographic applicability of the model by applying it to a broader range of urbanized watersheds. This would help validate the model's performance in different urban environments and ensure its robustness across diverse conditions.

Furthermore, while SPINpy 2 effectively models the impacts of stormwater management ponds on stream power and erosion risks, more data on specific NbS types, such as rain gardens, is needed to refine the discharge equation for a broader range of NbS types. This allows watershed managers to identify erosion hotspots reliably and design effective mitigation strategies. Additionally, future work of incorporating more detailed data on stormwater management infrastructure and refining the modelling of NbS effects could enhance the model's utility. For instance, specific NbS types—such as infiltration galleries—affect runoff and erosion differently depending on site conditions and urban land-use patterns (Fletcher et al., 2015). Refining algorithms to incorporate these interactions would provide a more detailed understanding of how NbS alter watershed dynamics, contributing to more

sustainable watershed planning. Additionally, the current ESM Scenario assumes ponds are 100% effective at reducing erosive forces due to increased runoff. However, in practice, SWM ponds have been designed and constructed with varying performance and criteria. It may be possible to justify a nuanced approach to adjust the efficiency factor that could reflect the varying performance of different pond and NbS solution designs using design parameters or monitoring data (which may be limited).

6.4 Conclusions

This research advances urban watershed management through developing and refining SPINpy 2, addressing the challenges of predicting erosion sensitivity in urbanized environments. The study's contributions include:

- Improved algorithms and discharge empirical equations to enhance the model's ability to predict runoff and streamflow dynamics in urban areas by incorporating the effects of impervious surfaces and land-use changes.
- By modelling the impacts of NbS, such as stormwater management ponds, SPINpy 2 evaluates how these interventions can mitigate erosion risks, stabilize channels, and reduce peak flows. This capability aligns with the broader goals of sustainable watershed management by promoting resilience in urban systems while addressing the environmental challenges posed by increased imperviousness.
- SPINpy 2 is an open-source platform, making it a valuable decision-support tool for watershed managers. SPINpy 2 helps identify high-priority areas for intervention and assess the long-term benefits of sustainable practices.

In summary, this study highlights the importance of open-source screening tools like SPINpy 2 for sustainable watershed management in the context of rapid urbanization and climate variability. These contributions provide a foundation for further research and practical applications, supporting the integration of NbS into urban planning and offering a pathway toward more resilient urban watersheds. SPINpy 2's ability to incorporate dynamic land-use changes into its algorithms ensures it remains a relevant tool for managing the evolving risks posed by urbanization, especially when combined with scenario modelling for NbS.

References

Adamowski, K., Liang, G.-C., & Patry, G. G. (1998). Annual maxima and partial duration flood series analysis by parametric and non-parametric methods. *Hydrological Processes*, *12*(10–11), 1685–1699. https://doi.org/10.1002/(SICI)1099-1085(199808/09)12:10/11<1685::AID-HYP689>3.0.CO;2-7

Alberti, M. (2005). The Effects of Urban Patterns on Ecosystem Function. *International Regional Science Review*, *28*(2), 168–192. https://doi.org/10.1177/0160017605275160

Althoff, D., & Neiva Rodrigues, L. (2021). Goodness-of-fit criteria for hydrological models: Model calibration and performance assessment. *Journal of Hydrology*, *600*, 126674. https://doi.org/10.1016/j.jhydrol.2021.126674

Anderson, D. G. (1970). Effects of urban development on floods in northern Virginia. In *Water Supply Paper* (Nos. 2001–C). U.S. Govt. Print. Off.,. https://doi.org/10.3133/wsp01C

Aravena, J. E., & Dussaillant, A. (n.d.). *Storm-Water Infiltration and Focused Recharge Modeling with Finite-Volume Two-Dimensional Richards Equation: Application to an Experimental Rain Garden*. https://doi.org/10.1061/ASCEHY.1943-7900.0000111

Arnold, J. G., Srinivasan, R., Muttiah, R. S., & Williams, J. R. (1998). Large Area Hydrologic Modeling and Assessment Part I: Model Development. *JAWRA Journal of the American Water Resources Association*, *34*(1), 73–89. https://doi.org/10.1111/j.1752-1688.1998.tb05961.x

Arnold, J. G., & Williams, J. R. (1987). Validation of SWRRB—Simulator for Water Resources in Rural Basins. *Journal of Water Resources Planning and Management*, *113*(2), 243–256. https://doi.org/10.1061/(ASCE)0733-9496(1987)113:2(243)

Ashmore, P., McDonald, J., & Barlow, V. (2023). Multi-decadal changes in river morphology in an urbanizing watershed: Highland Creek, Toronto, Canada. *Geomorphology*, *433*, 108710. https://doi.org/10.1016/j.geomorph.2023.108710

Bagnold, R. A. (1968). DEPOSITION IN THE PROCESS OF HYDRAULIC TRANSPORT. *Sedimentology*, *10*(1), 45–56. https://doi.org/10.1111/j.1365-3091.1968.tb01910.x

Bagnold, R. A. (1980). EMPIRICAL CORRELATION OF BEDLOAD TRANSPORT RATES IN FLUMES AND NATURAL RIVERS. *Proceedings of The Royal Society of London, Series A: Mathematical and Physical Sciences*, *372*(1751), 453–473.

Banai, R., & DePriest, T. (2014). *Urban Sprawl: Definitions, Data, Methods of Measurement, and Environmental Consequences*. *7*.

Bernhardt, E. S., & Palmer, M. A. (2007). Restoring streams in an urbanizing world. *Freshwater Biology*, *52*, 738–751. https://doi.org/10.1111/j.1365-2427.2006.01718.x

Bevan, V., MacVicar, B., Chapuis, M., Ghunowa, K., Papangelakis, E., Parish, J., & Snodgrass, W. (2018). Enlargement and evolution of a semi-alluvial creek in response to urbanization. *Earth Surface Processes and Landforms*, *43*(11), 2295–2312. https://doi.org/10.1002/esp.4391

Bhatt, A., Bradford, A., & Abbassi, B. E. (2019). Cradle-to-grave life cycle assessment (LCA) of low-impact-development (LID) technologies in southern Ontario. *Journal of Environmental Management*, *231*, 98–109. https://doi.org/10.1016/j.jenvman.2018.10.033

Bizzi, S., & Lerner, D. N. (2015). The Use of Stream Power as an Indicator of Channel Sensitivity to Erosion and Deposition Processes. *River Research and Applications*, *31*(1), 16–27. https://doi.org/10.1002/rra.2717

Bledsoe, B. P. (2002). Stream Erosion Potential and Stormwater Management Strategies. *Journal of Water Resources Planning and Management*, *128*(6), 451–455. https://doi.org/10.1061/ASCE0733-94962002128:6451

Bledsoe, B. P., & Watson, C. C. (2001). Effects of urbanization on channel instability. *Journal of the American Water Resources Association*, *37*(2), 255–270. https://doi.org/10.1111/j.1752-1688.2001.tb00966.x

Booth, D. B., Hartley, D., & Jackson, R. (2002). *FOREST COVER, IMPERVIOUS-SURFACE AREA, AND THE MITIGATION OF STORMWATER IMPACTS*.

Booth, D. B., Roy, A. H., Smith, B., & Capps, K. A. (2016). *Global perspectives on the urban stream syndrome*. *35*(1), 412–420. https://doi.org/10.2307/26540157

Bourne, L. (2001). The Urban Sprawl Debate: Myths, Realities and Hidden Agendas. *Plan Canada*, *41*(4). https://doi.org/10.25316/IR-181

Brawley-Chesworth, A. (2023). Professional differences in green infrastructure implementation: A case study of integrating engineering and ecological knowledge systems in the water sector. *Environmental Science & Policy*, *146*, 203–211. https://doi.org/10.1016/j.envsci.2023.05.013

Burns, M. J., Fletcher, T. D., Walsh, C. J., Ladson, A. R., & Hatt, B. E. (2012). Hydrologic shortcomings of conventional urban stormwater management and opportunities for reform. *Landscape and Urban Planning*, *105*, 230–240. https://doi.org/10.1016/j.landurbplan.2011.12.012

Cahill, T. H., Hoekstra, J., Horner, W., Pagano, S., Adams, M., & Cahill, P. E. *. (2004). *Application of a Water Balance Model and GIS for Sustainable Watershed Management*.

Charlton, R. (2007). *Fundamentals of Fluvial Geomorphology*. Routledge. https://doi.org/10.4324/9780203371084

Cheng, H., Yeon, C., Cho, M., & Park, C. (2023). Water requirement of Urban Green Infrastructure under climate change. *Science of The Total Environment*, 164887. https://doi.org/10.1016/j.scitotenv.2023.164887

Chin, A., & Gregory, K. J. (2005). Managing urban river channel adjustments. *Geomorphology*, *69*(1–4), 28–45. https://doi.org/10.1016/j.geomorph.2004.10.009

Clark, J., Clark, J. J., & Hamilton, J. (2008). *Effects of Land Use, Stormwater Management, and Channel Materials on the Channel Morphology of Apple Creek, WI*. *2*(1). https://www.researchgate.net/publication/228464759

Cohen-Shacham, E., Andrade, A., Dalton, J., Dudley, N., Jones, M., Kumar, C., Maginnis, S., Maynard, S., Nelson, C. R., Renaud, F. G., Welling, R., & Walters, G. (2019). Core principles for successfully implementing and upscaling Nature-based Solutions. *Environmental Science and Policy*, *98*, 20–29. https://doi.org/10.1016/j.envsci.2019.04.014

Cohen-Shacham, E., Janzen, C., Maginnis, S., & Walters, G. (2016). *Nature-based solutions to address global societal challenges*. https://doi.org/10.2305/IUCN.CH.2016.13.en

Credit Valley Conservation Authority, Risk Sciences International, & University of Waterloo. (2021). *Risk and Return on Investment Tool (RROIT) Technical Guide: Version 2.0 Risk and Return on Investment Tool Technical Guide Risk and Return on Investment Tool (RROIT) Technical Guide: Version 2.0*.

Decker, E. H., Elliott, S., & Smith, F. A. (2002). Megacities and the Environment. *The Scientific World JOURNAL*, *2*, 374–386. https://doi.org/10.1100/tsw.2002.103

Deng, J., Yin, H., Kong, F., Chen, J., Dronova, I., & Pu, Y. (2020). Determination of runoff response to variation in overland flow area by flow routes using UAV imagery. *Journal of Environmental Management*, *265*, 109868. https://doi.org/10.1016/j.jenvman.2019.109868

DHI. (2024). *MIKE+ UserGuide SWMM*.

Dussaillant, A. R., Asce, M., Chin, ;, Wu, H., Asce, A. M., & Potter, K. W. (2004). *Richards Equation Model of a Rain Garden*. https://doi.org/10.1061/ASCE1084-069920049:3219

Ferguson, R. I. (2005). Estimating critical stream power for bedload transport calculations in gravel-bed rivers. *Geomorphology*, *70*(1–2), 33–41. https://doi.org/10.1016/j.geomorph.2005.03.009

Fletcher, T. D., Andrieu, H., & Hamel, P. (2013). Understanding, management and modelling of urban hydrology and its consequences for receiving waters: A state of the art. *Advances in Water Resources*, *51*, 261–279. https://doi.org/10.1016/j.advwatres.2012.09.001

Fletcher, T. D., Shuster, W., Hunt, W. F., Ashley, R., Butler, D., Arthur, S., Trowsdale, S., Barraud, S., Semadeni-Davies, A., Bertrand-Krajewski, J. L., Mikkelsen, P. S., Rivard, G., Uhl, M., Dagenais, D., & Viklander, M. (2015). SUDS, LID, BMPs, WSUD and more – The evolution and application of terminology surrounding urban drainage. *Urban Water Journal*, *12*(7), 525–542. https://doi.org/10.1080/1573062X.2014.916314

Forte, A. M., & Whipple, K. X. (2019). Short communication: The Topographic Analysis Kit (TAK) for TopoToolbox. *Earth Surface Dynamics*, *7*(1), 87–95. https://doi.org/10.5194/esurf-7-87-2019

Ghunowa, K., MacVicar, B. J., & Ashmore, P. (2021). Stream power index for networks (SPIN) toolbox for decision support in urbanizing watersheds. *Environmental Modelling and Software*, *144*. https://doi.org/10.1016/j.envsoft.2021.105185

Giese, E., Rockler, A., Shirmohammadi, A., & Pavao-Zuckerman, M. A. (2019). Assessing Watershed-Scale Stormwater Green Infrastructure Response to Climate Change in Clarksburg, Maryland. *Journal of Water Resources Planning and Management*, *145*(10). https://doi.org/10.1061/(ASCE)WR.1943-5452.0001099

Gregory, K. J., Davis, R. J., & Downs, P. W. (1992). Identification of river channel change to due to urbanization. *Applied Geography*, *12*(4), 299–318. https://doi.org/10.1016/0143-6228(92)90011-B

Gupta, H. V., & Kling, H. (2011). On typical range, sensitivity, and normalization of Mean Squared Error and Nash-Sutcliffe Efficiency type metrics. *Water Resources Research*, *47*(10). https://doi.org/10.1029/2011WR010962

Gupta, H. V., Kling, H., Yilmaz, K. K., Martinez, G. F., & Kling, H. (2009). Decomposition of the mean squared error and NSE performance criteria: Implications for improving hydrological modelling. *Journal of Hydrology*, *377*, 80–91. https://doi.org/10.1016/j.jhydrol.2009.08.003

Guswa, A. J., Hall, B., Cheng, C., & Thompson, J. R. (2020). Co-designed Land-use Scenarios and their Implications for Storm Runoff and Streamflow in New England. *Environmental Management*, *66*(5), 785–800. https://doi.org/10.1007/s00267-020-01342-0

Guthrie, Rick & Deniseger, John. (2001). *Impervious Surfaces in French Creek*. Ministry of Water, Land and Air Protection.

Hammer, T. R. (1972). Stream channel enlargement due to urbanization. *Water Resources Research*, *8*(6), 1530–1540. https://doi.org/10.1029/WR008i006p01530

Han, M., Shen, H., Tolson, B. A., Craig, J. R., Mai, J., Lin, S. G. M., Basu, N. B., & Awol, F. S. (2023). BasinMaker 3.0: A GIS toolbox for distributed watershed delineation of complex lake-river routing networks. *Environmental Modelling & Software*, *164*, 105688. https://doi.org/10.1016/j.envsoft.2023.105688

Harman, W. A., Jennings, G. D., Tweedy, K. R., Buck, J. A., & Taylor, D. L. (2001). Lessons Learned from Designing and Constructing In-Stream Structures. In *Wetlands Engineering & River Restoration 2001* (pp. 1–9). https://doi.org/10.1061/40581(2001)71

Huang, S. yang, Cheng, S. jen, Wen, J. chau, & Lee, J. huang. (2008). Identifying peak-imperviousness-recurrence relationships on a growing-impervious watershed, Taiwan. *Journal of Hydrology*, *362*(3–4), 320–336. https://doi.org/10.1016/j.jhydrol.2008.09.002

Hussain, M., & Imitiyaz, I. (2018). *URBANIZATION CONCEPTS, DIMENSIONS AND FACTORS*. *9*, 23513–23523.

Jefferson, A. J., Bhaskar, A. S., Hopkins, K. G., Fanelli, R., Avellaneda, P. M., & McMillan, S. K. (2017). Stormwater management network effectiveness and implications for urban watershed function: A critical review. *Hydrological Processes*, *31*(23), 4056–4080. https://doi.org/10.1002/hyp.11347

Johnson, J. P., & Hunt, W. F. (2020). Field Assessment of the Hydrologic Mitigation Performance of Three Aging Bioretention Cells. *Journal of Sustainable Water in the Built Environment*, *6*(4). https://doi.org/10.1061/jswbay.0000925

Jordan, D. C., & Fonstad, M. A. (2005). Two Dimensional Mapping of River Bathymetry and Power Using Aerial Photography and GIS on the Brazos River, Texas. *Geocarto International*, *20*(3), 13–20. https://doi.org/10.1080/10106040508542351

Julien, P. Y. (1995). Erosion and sedimentation. In *Erosion and sedimentation*.

Korgaonkar, Y., Meles, M. B., Guertin, D. P., Goodrich, D. C., & Unkrich, C. (2020). Global sensitivity analysis of KINEROS2 hydrologic model parameters representing green infrastructure using the STAR-VARS framework. *Environmental Modelling and Software*, *132*. https://doi.org/10.1016/j.envsoft.2020.104814

Kumar, P., Debele, S. E., Sahani, J., Rawat, N., Marti-Cardona, B., Alfieri, S. M., Basu, B., Basu, A. S., Bowyer, P., Charizopoulos, N., Gallotti, G., Jaakko, J., Leo, L. S., Loupis, M., Menenti, M., Mickovski, S. B., Mun, S. J., Gonzalez-Ollauri, A., Pfeiffer, J., … Zieher, T. (2021). Nature-based solutions efficiency evaluation against natural hazards: Modelling methods, advantages and limitations. *Science of the Total Environment*, *784*. https://doi.org/10.1016/j.scitotenv.2021.147058

Lafortezza, R., Chen, J., van den Bosch, C. K., & Randrup, T. B. (2018). Nature-based solutions for resilient landscapes and cities. *Environmental Research*, *165*, 431–441. https://doi.org/10.1016/j.envres.2017.11.038

Lee, H., Kim, H., Kim, J., Jun, S. M., Hwang, S., Song, J. H., & Kang, M. S. (2023). Analysis of the effects of low impact development practices on hydrological components using HSPF. *Journal of Hydro-Environment Research*, *46*, 72–85. https://doi.org/10.1016/j.jher.2023.01.001

Leopold, L. B. (1968). Hydrology for urban land planning—A guidebook on the hydrologic effects of urban land use. In *Circular* (No. 554). U.S. Geological Survey. https://doi.org/10.3133/cir554

Leopold, L. B., & Maddock, T. (1953). *The Hydraulic Geometry of Stream Channels and Some Physiographic Implications*.

Li, C., Sun, G., Cohen, E., Zhang, Y., Xiao, J., McNulty, S. G., & Meentemeyer, R. K. (2020). Modeling the impacts of urbanization on watershed-scale gross primary productivity and tradeoffs with water yield across the conterminous United States. *Journal of Hydrology*, *583*. https://doi.org/10.1016/j.jhydrol.2020.124581

Lindsay, J. B. (2016). Whitebox GAT: A case study in geomorphometric analysis. *Computers and Geosciences*, *95*, 75–84. https://doi.org/10.1016/j.cageo.2016.07.003

Marcinkowski, P., Kiczko, A., & Kardel, I. (2022). Large-scale assessment of spatial variability in stream power distribution: An indicator of channel erosion and deposition potential. *Journal of Hydrology*, *605*. https://doi.org/10.1016/j.jhydrol.2021.127319

Meyer, J. L., Paul, M. J., & Taulbee, W. K. (2005). Stream ecosystem function in urbanizing landscapes. *Journal of the North American Benthological Society*, *24*(3), 602–612. https://doi.org/10.1899/04-021.1

Mitchell, V. G., Mein, R. G., & McMahon, T. A. (2001). Modelling the urban water cycle. *Environmental Modelling & Software*, *16*(7), 615–629. https://doi.org/10.1016/S1364-8152(01)00029-9

Molnar, P., Anderson, R. S., Kier, G., & Rose, J. (2006). Relationships among probability distributions of stream discharges in floods, climate, bed load transport, and river incision. *Journal of Geophysical Research: Earth Surface*, *111*(F2). https://doi.org/10.1029/2005JF000310

Moriasi, D. N., Gitau, M. W., Pai, N., & Daggupati, P. (2015). Hydrologic and water quality models: Performance measures and evaluation criteria. *Transactions of the ASABE*, *58*(6), 1763–1785. https://doi.org/10.13031/TRANS.58.10715

Nash, J. E., & Sutcliffe, J. V. (1970). Not to be reproduced by photoprint or microfilm without written permission from the publisher RIVER FLOW FORECASTING THROUGH CONCEPTUAL MODELS PART I-A DISCUSSION OF PRINCIPLES*. *Journal of Hydrology*, *10*, 282–290.

Nelson, P. A., Smith, J. A., & Miller, A. J. (2006). Evolution of channel morphology and hydrologic response in an urbanizing drainage basin. *Earth Surface Processes and Landforms*, *31*(9), 1063–1079. https://doi.org/10.1002/esp.1308

Olding, D. D., Steele, T. S., & Nemeth, J. C. (2004). *Operational Monitoring of Urban Stormwater Management Facilities and Receiving Subwatersheds in Richmond Hill, Ontario*. *39*(4), 392–405.

Papangelakis, E., MacVicar, B., Ashmore, P., Gingerich, D., & Bright, C. (2022). Testing a Watershed-Scale Stream Power Index Tool for Erosion Risk Assessment in an Urban River. *Journal of Sustainable Water in the Built Environment*, *8*(3). https://doi.org/10.1061/JSWBAY.0000989

Parker, C., Thorne, C. R., & Clifford, N. J. (2015). Development of ST: REAM: A reach-based stream power balance approach for predicting alluvial river channel adjustment. *Earth Surface Processes and Landforms*, *40*(3), 403–413. https://doi.org/10.1002/esp.3641

Paul, M. J., & Meyer, J. L. (2001). Streams in the urban landscape. *Annual Review of Ecology and Systematics*, *32*, 333–365. https://doi.org/10.1146/annurev.ecolsys.32.081501.114040

Phillips, R. T. J., & Desloges, J. R. (2014). Glacially conditioned specific stream powers in low-relief river catchments of the southern Laurentian Great Lakes. *Geomorphology*, *206*, 271–287. https://doi.org/10.1016/j.geomorph.2013.09.030

Poff, N. L. (2002). Ecological Response to and Management of Increased Flooding Caused by Climate Change. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, *360*(1796), 1497–1510.

Poff, N. L., Allan, J. D., Bain, M. B., Karr, J. R., Prestegaard, K. L., Richter, B. D., Sparks, R. E., & Stromberg, J. C. (1997). The Natural Flow Regime. *BioScience*, *47*(11), 769–784. https://doi.org/10.2307/1313099

Poff, N. L., Bledsoe, B. P., & Cuhaciyan, C. O. (2006). Hydrologic variation with land use across the contiguous United States: Geomorphic and ecological consequences for stream ecosystems. *Geomorphology*, *79*(3), 264–285. https://doi.org/10.1016/j.geomorph.2006.06.032

Qin, H.-P., Li, Z.-X., & Fu, G. (2013). The effects of low impact development on urban flooding under different rainfall characteristics. *Journal of Environmental Management*, *129*, 577–585. https://doi.org/10.1016/j.jenvman.2013.08.026

Rhoads, B. L. (1987). Stream power terminology. *Professional Geographer*, *39*(2), 189–195. https://doi.org/10.1111/j.0033-0124.1987.00189.x

Rodrigues, B. N., Molina, V. E., & Canteras, F. B. (2023). Green Infrastructure as a solution to mitigate the effects of climate change in a coastal area of social vulnerability in Fortaleza (Brazil). *Environmental Advances*, 100398. https://doi.org/10.1016/j.envadv.2023.100398

Rossman, Lewis A. & Simon, Michelle A. (2022). *Storm Water Management Model User's Manual Version 5.2*.

Roux, C., Alber, A., Bertrand, M., Vaudor, L., & Piégay, H. (2015). "FluvialCorridor": A new ArcGIS toolbox package for multiscale riverscape exploration. *Geomorphology*, *242*, 29–37. https://doi.org/10.1016/j.geomorph.2014.04.018

Schwanghart, W., & Kuhn, N. J. (2010). TopoToolbox: A set of Matlab functions for topographic analysis. *Environmental Modelling and Software*, *25*(6), 770–781. https://doi.org/10.1016/j.envsoft.2009.12.002

Sharpe, DR, Barnett, PJ, Brennand, TA, Finley, D, Gorrell, G, Russell, HAJ, & Stacey, P. (1997). *Surficial Geology of the Greater Toronto and Oak Ridges Moraine Area. Canada Geological Society: Southern Ontario*. https://www.sfu.ca/geog/paleoglaciology/pubs_files/_SurfGeol_gscof_3062_e_1997_mn01.pdf

Sikder, M. S., Wang, J., Allen, G. H., Sheng, Y., Yamazaki, D., Song, C., Ding, M., Crétaux, J.-F., & Pavelsky, T. M. (2023). Lake-TopoCat: A global lake drainage topology and catchment database. *Earth System Science Data*, *15*(8), 3483–3511. https://doi.org/10.5194/essd-15-3483-2023

Spiess, A.-N., & Neumeyer, N. (2010). An evaluation of R2 as an inadequate measure for nonlinear models in pharmacological and biochemical research: A Monte Carlo approach. *BMC Pharmacology*, *10*, 6. https://doi.org/10.1186/1471-2210-10-6

Suits, K., Annus, I., Kändler, N., Karlsson, T., Maris, A. V., Kaseva, A., Kotoviča, N., & Rajarao, G. K. (2023). Overview of the (Smart) Stormwater Management around the Baltic Sea. *Water (Switzerland)*, *15*(8). https://doi.org/10.3390/w15081623

Tarboton, D. G., Bras, R. L., & Rodriguez-Iturbe, I. (1991). On the extraction of channel networks from digital elevation data. *Hydrological Processes*, *5*(1), 81–100. https://doi.org/10.1002/hyp.3360050107

Tetford, P. E. (2024). *A Critical Analysis of the Adjustment of River Channel Geometry in Southern Ontario and the Role of Land Use, Climate Variability, and Boundary Conditions*.

Tilley, Janet & Slonecker, E. Terrence. (2006). *Quantifying the Components of Impervious Surfaces* (Open-File Report) [Open-File Report].

Tillinghast, E. D., Hunt, W. F., & Jennings, G. D. (2011). *Relating Stormwater Control Measure (SCM) Discharge Design Standards to Stream Erosion in Piedmont North Carolina: Case Studies in Raleigh and Chapel Hill, North Carolina*.

Tillinghast, E., Hunt, W., Jennings, G., & D'Arconte, P. (2012). Increasing Stream Geomorphic Stability Using Storm Water Control Measures in a Densely Urbanized Watershed. *Journal of Hydrologic Engineering*, *17*, 1381–1388. https://doi.org/10.1061/(ASCE)HE.1943-5584.0000577

Tisdale, H. (1941). The Process of Urbanization. *Social Forces*, *20*(3), 311–316.

Tong, P., Yin, H., Wang, Z., & Trivers, I. (2022). Combining Stormwater Management and Park Services to Mitigate Climate Change and Improve Human Well-Being: A Case Study of Sponge City Parks in Shanghai. *Land*, *11*(9), Article 9. https://doi.org/10.3390/land11091589

United States Department of the Interior Geological Survey Water Resources Branch. (1947). *Instructions for Flood Frequency Compilations* [Handbook].

Urban Drainage and Flood Control District. (2014, September 30). *CUHP User Manual*. Mile High Flood District. https://mhfd.org/resources/software/

Vietz, G. J., Walsh, C. J., & Fletcher, T. D. (2016). Urban hydrogeomorphology and the urban stream syndrome: Treating the symptoms and causes of geomorphic change. *Progress in Physical Geography*, *40*(3), 480–492. https://doi.org/10.1177/0309133315605048

Vocal Ferencevic, M., & Ashmore, P. (2012). Creating and evaluating digital elevation model-based stream-power map as a stream assessment tool. *River Research and Applications*, *28*(9), 1394–1416. https://doi.org/10.1002/rra.1523

Walsh, C. J., Fletcher, T. D., & Burns, M. J. (2012). Urban Stormwater Runoff: A New Class of Environmental Flow Problem. *PLoS ONE*, *7*(9). https://doi.org/10.1371/journal.pone.0045814

Walsh, C. J., Roy, A. H., Feminella, J. W., Cottingham, P. D., Groffman, P. M., & Morgan, R. P. (2005). The urban stream syndrome: Current knowledge and the search for a cure. *Journal of the North American Benthological Society*, *24*(3), 706–723. https://doi.org/10.1899/04-028.1

Williams, J. R., & Arnold, J. G. (1997). A system of erosion—Sediment yield models. *Soil Technology*, *11*(1), 43–55. https://doi.org/10.1016/S0933-3630(96)00114-6

Winston, R. J., Dorsey, J. D., & Hunt, W. F. (2016). Quantifying volume reduction and peak flow mitigation for three bioretention cells in clay soils in northeast Ohio. *Science of the Total Environment*, *553*, 83–95. https://doi.org/10.1016/j.scitotenv.2016.02.081

Winterbottom, S. J., & Gilvear, D. J. (1997). Quantification of channel bed morphology in gravel-bed rivers using airborne multispectral imagery and aerial photography. *Regulated Rivers: Research & Management*, *13*(6), 489–499. https://doi.org/10.1002/(SICI)1099-1646(199711/12)13:6<489::AID-RRR471>3.0.CO;2-X

Wohl, E., Angermeier, P. L., Bledsoe, B., Kondolf, G. M., MacDonnell, L., Merritt, D. M., Palmer, M. A., Poff, N. L., & Tarboton, D. (2005). River restoration. *Water Resources Research*, *41*(10). https://doi.org/10.1029/2005WR003985

Wolman, M. G., & Schick, A. P. (1967). Effects of construction on fluvial sediment, urban and suburban areas of Maryland. *Water Resources Research*, *3*(2), 451–464. https://doi.org/10.1029/WR003i002p00451

Woods Ballard, B. (Bridget) & Construction Industry Research and Information Association. (2015). *The SuDS manual*. CIRIA.

Xu, H., Zhong, T., Chen, Y., & Zhang, J. (2023). How to simulate future scenarios of urban stormwater management? A novel framework coupling climate change, urbanization, and green stormwater infrastructure development. *Science of The Total Environment*, *874*, 162399. https://doi.org/10.1016/J.SCITOTENV.2023.162399

Appendix A

Urban Width and Discharge Methods

A.1 Urban Width & Discharge Equation Detailed Statistics

A.1.1 Statistics Considered for Discharge & Width Equation Calibration

The coefficients used in the discharge and width equations are based on a nonlinear regression model calibrated with the training dataset, which is then applied to all the data to identify how well it performs. The statistical analysis will determine how well the calibrated models fit the data and the level of uncertainty present in each model. Table 1**Error! Reference source not found.** states that the statistical analysis is completed and that it will represent the variable.

The R2 value is typically used to express the goodness of fit for a linear regression model. The value obtained is indicative of the strength of the relationship between the model and the dependent variable. R^2 is a statistical measure representing the proportion of variance in the dependent variable that independent variables can explain a regression model (Althoff & Neiva Rodrigues, 2021; Moriasi et al., 2015). However, for a non-linear model, R^2 is not a reliable statistic to measure the reliability of the model and the observed values. Speiess and Neumeyer (2010) found that the R^2 value leads to incorrect conclusions and does not differentiate between good and bad nonlinear models. To assess the calibrated models in this study, the following statistics were calculated: Standard Error of Regression (S), Error Sum of Squares (SSE), Mean Squared Error (MSE), Nash Sutcliffe Efficiency (NSE), Percent Bias (PBIAS), Kling Gupta Efficiency (KGE) and Akaike Information Criterion (AIC).

Table 1: Statistical Analysis Method and Variables for Model Goodness of Fit Testing

Method	Equation	Variables	Purpose and Interpretation
Coefficient of Determination (Pearson) (R^2)	$\boldsymbol{R^2 = \frac{[\sum_{i=1}^{n}(O_i - \bar{O})(S_i - \bar{S})]^2}{\sum_{i=1}^{n}(O_i - \bar{O})^2 \sum_{i=1}^{n}(S_i - \bar{S})^2}}$	Where O_i is the observed value, $\bar{O}$ is the mean of the observed values, $S_i$is the modelled data, and $\bar{S}$ is the mean of the modelled values.	An R^2 value of 1 indicates that the independent variables can explain the variability of the dependent variable (Spiess & Neumeyer, 2010).
Standard Error of the Regression (S)	$\boldsymbol{S = \sqrt{\frac{\sum(y_i - \hat{y}_i)^2}{n - p - 1}}}$	Where y_i is the value for the observed data, $\check{y}_i$is the model predicted value, n is the number of observations, and p is the number of coefficients in the model not including constants.	The S statistic is a goodness of fit measure that is appropriate to use for both linear and nonlinear model. S measures the absolute measure of the distance the data falls from the regression line.
Error Sum of Squares (SSE)	$\boldsymbol{SSE = \sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$	Where y_i is the value for the observed data, and $\hat{y}_i$is the model predicted value.	SSE evaluates the variability of the model predicted values around the mean observed data. If the SSE value is lower it indicates a closer fit to the regression line, with zero representing a perfect fit.
Mean Square Error (MSE)	$\boldsymbol{MSE = \frac{\sum(y_i - \hat{y}_i)^2}{n}}$	Where y_i is the value for the observed data, $\check{y}_i$is the model predicted value, and n is the number of observations.	MSE measures the amount of error in each model, by assessing the squared difference between the observed and predicted values. A low MSE indicates good fit.
Nash Sutcliffe Efficiency (NSE)	$\boldsymbol{NSE = 1 - \frac{\sum_{i=1}^{n}(O_i - S_i)^2}{\sum_{i=1}^{n}(O_i - \bar{O})^2}}$	Where O_i is the observed value, $\bar{O}$ is the mean of the observed values, and $S_i$is the modelled data.	Nash Sutcliffe Efficiency is another statistical metric that is commonly used for assessing the accuracy of hydrological models (Nash & Sutcliffe, 1970). NSE values greater than 0.75 means that the model is performing well, NSE values less than 0.50 indicates a poor model performance.
Percent Bias (PBIAS)	$\boldsymbol{BIAS = \frac{\sum_{i=1}^{n}(O_i - S_i)}{\sum_{i=1}^{n}(O_i)} \times 100}$	Where O_i is the observed value and $S_i$is the modelled data.	PBIAS measures the tendency of the predicted data to be larger or smaller than the observed data (Althoff & Neiva Rodrigues, 2021). PBIAS values under 10% are considered good and indicates the model is accurate. A negative PBIAS value indicates an underestimation of the model, and a positive value indicates overestimation.
Kling Gupta Efficiency (KGE)	$\boldsymbol{KGE = 1 - \sqrt{(r-1)^2 + (\beta - 1)^2 + (\gamma - 1)^2}}$	Where r is the Pearson correlation coefficient of observed and modelled values, β is the ratio of the standard deviation of modelled values to the observed values, γ is the ratio of the modelled values to the observed values.	KGE measures the hydrologic model performance by considering both the correlation and reliability (Gupta et al., 2009; Gupta & Kling, 2011). A KGE of 1 indicates perfect fit between simulations and observations, 0 indicates a poor fit, and a value < 0 indicates the mean of the observations provides better estimates.
Akaike Information Criterion (AIC)	$\boldsymbol{AIC = 2K - 2\ln(L)}$	Where K is the number of independent variables used, and L is the log likelihood estimate.	AIC is a goodness of fit measure that allows for the comparison between models to select the best model. The AIC is calculated using Equation 9 in R, and the

ideal model is chosen based on the lowest AIC value .

A.1.2 NSM LU Urban Discharge Equation Calibration/Validation Using Modelled Data

For the validation process, Equation D again demonstrated superior performance as a generalized model despite appearing second in some rankings. With 1274 data points from eight subwatersheds, this validation phase confirms that Equation D captures the complexity of discharge behaviour across varying degrees of imperviousness and drainage areas and provides a reliable framework for general use, particularly in subwatersheds lacking stormwater management controls. In calculating discharge across the dataset, each calibrated equation's performance was evaluated by categorizing results by percent imperviousness and drainage area into bins, allowing for a comprehensive understanding of how each equation responds to different environmental settings. For each bin, the MSE, the root mean square error (RMSE), and the percentage error is calculated (Table 2**Error! Reference source not found.**).

Table 2: Equations A,B,C,D RMSE & Percentage Error for Imperviousness Bins

Bins	**Equation A**		**Equation B**		**Equation C**		**Equation D**	
	RMSE	*Error %*	*RMSE*	*Error %*	*RMSE*	*Error %*	*RMSE*	*Error %*
< 1	10.39	148.99	10.39	148.99	10.27	147.20	0.49	7.04
1-2	-	-	-	-	-	-	-	-
2-5	1.94	76.87	1.62	64.23	0.49	19.32	0.40	15.95
5-10	2.78	61.40	1.16	25.56	1.75	38.67	2.45	54.15
10-20	1.74	23.93	11.69	160.47	13.06	179.37	14.31	196.53
20-30	21.93	81.59	20.19	75.10	16.77	62.36	16.96	63.07
30-40	17.98	77.66	14.29	61.73	9.48	40.94	9.80	42.33
40-50	50.54	87.05	21.16	36.44	21.06	36.28	21.88	37.69
50-60	26.02	82.29	10.64	33.64	6.34	20.06	6.67	21.09
60-70	34.25	88.68	20.52	53.12	12.10	31.32	13.33	34.50

Next, the modelled discharge and calculated discharges for the equations were binned by the drainage area to assess the performance of each equation for varying drainage areas. For each bin MSE, RMSE, and the percentage error is calculated (Table 3**Error! Reference source not found.**).

Table 3: Equations A,B,C,D RMSE & Percentage Error for Drainage Area Bins

Bins	Equation A		Equation B		Equation C		Equation D	
	RMSE	*Error %*	*RMSE*	*Error %*	*RMSE*	*Error %*	*RMSE*	*Error %*
<1	3.04	97.35	2.50	80.14	2.27	72.68	2.31	74.01
1-2	2.95	89.72	5.24	159.31	5.13	155.88	5.17	157.11
2-5	15.95	98.29	11.46	70.60	7.72	47.55	7.77	47.88
5-10	22.62	104.07	12.97	59.67	8.77	40.34	9.52	43.82
10-20	39.02	95.70	20.16	49.44	19.30	47.33	19.88	48.76
20-30	36.78	87.01	15.85	37.50	16.24	38.42	17.20	40.69
30-40	20.80	55.36	18.71	49.81	12.20	32.46	12.89	34.31

Each equation was ranked based on the percent error for the impervious percent and the drainage area (Table 4 and 5). Then, the rankings for each equation across the bins are summed. The minimum ranking illustrates which equation best represents the specific range of imperviousness and drainage areas.

Table 4: Equation Performance Rankings (Imperviousness)

Impervious %	Equation A	Equation B	Equation C	Equation D
<1	4	3	2	1
1-2	-	-	-	-
2-5	4	3	2	1
5-10	4	1	2	3
10-20	1	2	3	4
20-30	4	3	1	2
30-40	4	3	1	2
40-50	4	2	1	3
50-60	4	3	1	2
60-70	4	3	1	2
Sum	**33**	**23**	**14**	**20**

Table 5: Equation Performance Rankings (Drainage Area)

DA (km^2)	Equation A	Equation B	Equation C	Equation D
<1	4	3	1	2
1-2	1	4	2	3
2-5	4	3	1	2
5-10	4	3	1	2
10-20	4	3	1	2
20-30	4	1	2	3
30-40	4	3	1	2
Sum	**25**	**20**	**9**	**16**

While Equation C had the lowest summed rankings across imperviousness and drainage area bins (Tables 4 and 5), Equation D remains the more practical choice due to its unique advantages. Equation D's relatively low RMSE and error percentages across imperviousness bins (such as 15.95% error in the 2-5% impervious range and 42.33% in the 30-40% range) underscore its adaptability to urban conditions. Furthermore, as imperviousness approaches zero, Equation D mirrors the rural empirical equation by Phillips & Desloges (2014), allowing for a seamless transition to rural discharge behaviour in undeveloped areas. This alignment with the rural model enhances Equation D's suitability for general use, especially when applying SPINpy 2 to a landscape with urban and rural subwatershed characteristics.

Therefore, Equation D is the most suitable 2-year discharge model to represent the relationship between imperviousness, drainage area, and discharge in the LU Analysis which accurately reflects a scenario without stormwater management controls. Equation D's integration into SPINpy 2 enables a unified approach for calculating urban discharge, ensuring consistency across varying subwatershed types and providing a reliable baseline for hydrologic predictions within urbanized subwatersheds without stormwater management methods.

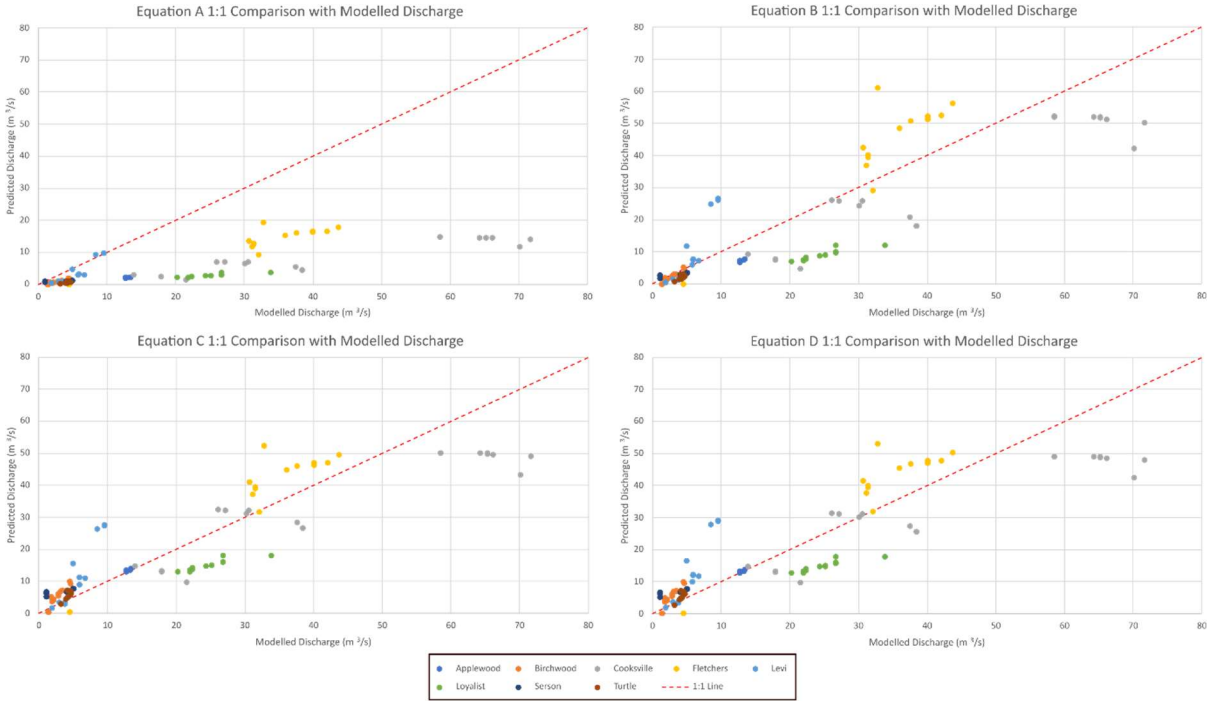


Figure 1: Comparison of predicted discharge to the modelled discharge for the eight NSM Scenario Subwatersheds in the training dataset

A.1.3 ESM LU Urban Discharge Equation Calibration/Validation Using Modelled Data

Table 6: Equations Root Mean Square Error & Percentage Error for Imperviousness Bins

Bins	Equation G		Equation H		Equation I	
	RMSE	*Error %*	*RMSE*	*Error %*	*RMSE*	*Error %*
< 1	14.56	73.12	5.42	27.21	16.72	83.94
1-2	13.61	72.79	5.49	29.36	15.28	81.69
2-5	9.95	99.30	4.46	44.51	9.16	91.42
5-10	10.45	98.48	4.11	38.70	5.92	55.76
10-20	8.14	59.52	2.54	18.58	4.98	36.42
20-30	-	-	-	-	-	-
30-40	4.87	110.62	19.31	438.78	2.38	54.04
40-50	4.42	100.46	21.19	481.64	1.86	42.20

Table 7: Equations G, H, I RMSE & Percentage Error for Drainage Area Bins

Bins	Equation G		Equation H		Equation I	
	RMSE	*Error %*	*RMSE*	*Error %*	*RMSE*	*Error %*
<1	4.81	109.35	19.56	444.60	2.32	52.65
1-2	2.91	142.03	2.04	99.62	0.34	16.56
2-5	4.56	47.41	3.34	34.75	10.45	108.66
5-10	3.02	35.38	3.49	40.93	9.13	106.90
10-20	3.47	25.44	1.00	7.31	5.09	37.27
20-30	10.24	67.19	9.96	65.35	26.22	172.02
30-40	16.38	88.27	13.73	73.98	16.64	89.67

Table 8: Equation Performance Rankings (Impervious)

Impervious %	Equation G	Equation H	Equation I
<1	2	1	3
1-2	2	1	3
2-5	3	1	2
5-10	3	1	2
10-20	3	1	2
20-30	-	-	-
30-40	2	3	1
40-50	2	3	1
Sum	**17**	**11**	**14**

Table 9: Equation Performance Rankings (Drainage Area)

DA (km^2)	Equation G	Equation H	Equation I
<1	2	3	1
1-2	3	2	1
2-5	2	1	3
5-10	1	2	3
10-20	2	1	3
20-30	2	1	3
30-40	2	1	3
Sum	**14**	**11**	**17**

The calibration results demonstrate that while Equation H performs the best statistically, Equation G is the most suitable model for simulating discharge in urban watersheds with stormwater management ponds. Equation G displayed a well-balanced performance across different imperviousness and drainage area bins, particularly for moderately urbanized regions with imperviousness levels between 5% and 50% and drainage areas ranging from 5 to 40 km^2. Despite a moderate NSE value of 0.52, Equation G's other statistical metrics, such as the lower Akaike Information Criterion (AIC) and acceptable PBIAS, underscore its strength in fitting the observed data. This indicates that Equation G effectively captures discharge trends while offering a better fit to the dataset than the other equations. However, its performance in highly urbanized areas (imperviousness >50%) may be less reliable due to limited calibration data, suggesting the need for additional refinement to enhance its applicability in extremely urban contexts.

Conversely, Equation H faces significant limitations in practical use, although it performs adequately in low imperviousness and small drainage areas (both under 2%), Equation H's reliability diminishes in larger drainage areas and higher imperviousness levels. A critical issue with Equation H is its tendency to produce negative discharge values. This issue occurs because the equation improperly assigns stormwater management drainage area (SWMDA) benefits to smaller tributaries that lack stormwater management (SWM) ponds, particularly where small tributaries flow into a main channel that benefits from upstream SWM structures. The model inaccurately applies SWMDA benefits across the tributary system, resulting in non-viable negative discharge calculations, particularly in headwater areas without SWM support. Given these discrepancies, Equation H is considered invalid for discharge modelling in subwatersheds where SWM ponds do not influence smaller tributaries.

Equation I, while somewhat competitive in certain mid-size drainage areas (10-20 km^2), has the poorest overall fit among the equations with respect to NSE and PBIAS. With a negative NSE (-0.33) and a high PBIAS (26.83), Equation I appears to overestimate discharge volumes, as reflected in its mean discharge (15.02 m^3/s), significantly higher than the observed average. These metrics suggest that Equation I may not fully capture the discharge-mitigating effects of SWM ponds, particularly in urban areas. The calibration results for Equation I highlight potential misalignment in response to impervious area, indicating further refinement may be required for accurate discharge representation.

Therefore, Equation G emerges as the preferred model for urban watershed discharge simulation when SWM ponds are present, demonstrating a balanced performance across relevant urban scenarios. The negative discharge issues in Equation H underscore a need for careful re-evaluation of SWMDA benefits in smaller tributaries, while Equation I's higher error metrics suggest further calibration is required to optimize its response to impervious area. These insights affirm the need for tailored discharge equations based on urbanization patterns and drainage characteristics, with Equation G showing particular promise for broader applications in urban watershed management. The next section will detail the validation methods for Equation G, assessing its performance against the provided gauge data further to establish its accuracy and applicability across diverse watershed scenarios.

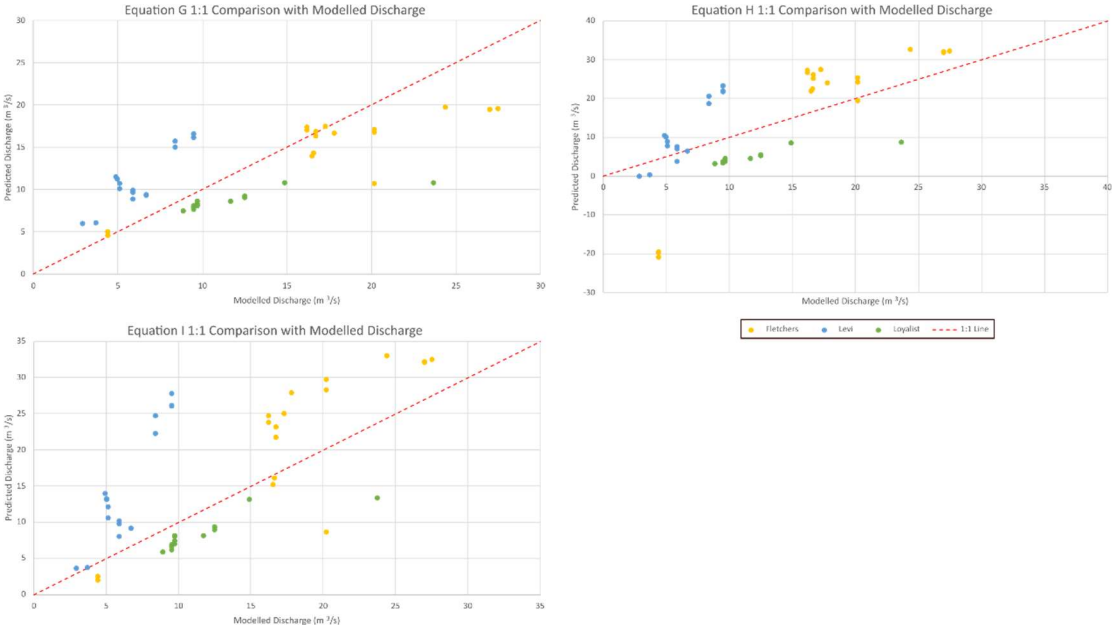


Figure 2: Comparison of predicted discharge to the modelled discharge for the three ESM Scenario subwatersheds in the training dataset

A.1.4 ESM LU Urban Discharge Equation Validation Using Gauge Data

Few gauging stations exist for the relatively small urban basins being investigated, but the analysis will allow a more accurate representation of urban hydrology and uncertainty in the simple equation. Accurate prediction of flood recurrence is crucial for effective management of urban watersheds, especially when evaluating the 2-year return period typically used in geomorphic analysis. Two main approaches for constructing flood datasets exist: Partial Duration Series (PDS) and Annual Maxima Series (AMS). The AMS method involves selecting the highest annual discharge from mean daily values at a gauge which ensures statistical independence between years (Molnar et al., 2006; Tetford, 2024). The PDS method includes all flood events exceeding a base threshold and considers all 365 days of observation (United States Department of the Interior Geological Survey Water Resources Branch, 1947). The PDS method can capture multiple floods within a year, providing a more comprehensive understanding of flood dynamics, particularly for frequent events. PDS datasets often exhibit higher discharge values than AMS datasets for such events (Adamowski et al., 1998; Tetford, 2024).

In order to validate the outputs of the 2-year urban discharge equation calibrated in the previous section (Equation G), the provided gauge data (Figure 3-2 and Table 3-1) and the AMS method were used to calculate a 2-year discharge at the gauges. First, a frequency analysis was conducted to determine the discharges for the gauges for the 2-100-year return periods. The next step after a frequency analysis was plotting the 2-year discharge estimates for each gauge (Table 4-18 and Figure 3A). Then the calculated 2-year discharge of the gauges was compared to the discharges calculated from Equation G and the modelled data provided by the CVC. The aim of the validation exercise is to see how accurate the predictions are compared to the observed gauge data. Figure 3A graphically shows the outcome of the frequency analysis for the three (3) gauges of interest. Then, Bernoulli statistics are used to verify how often the modelled and Equation G outputs exceed the 2-year discharge calculated using the observed gauges. The data presented in Figure 3 shows the probabilities of exceeding the 2-year observed discharge threshold for three subwatersheds: Fletchers, Loyalist, and Levi. Additionally Figures 3B to 3D display a comparison of the calculated 2-year gauge discharge, Modelled discharge and the predicted discharge from Equation G. The results provide key insights into the performance of the modelled data and the predictions from Equation G.

Table 10: 2-100 Year Discharge Thresholds Calculated Using A Frequency Analysis

Frequency	Fletchers Creek at 2nd Line	Levi Creek at Financial Drive	Loyalist Creek at 5th Line
2	10.46	3.66	10.67
5	13.91	6.71	14.04
10	16.47	8.93	15.29
25	20.06	11.81	16.21
50	22.98	13.99	16.60
100	26.14	16.15	16.85

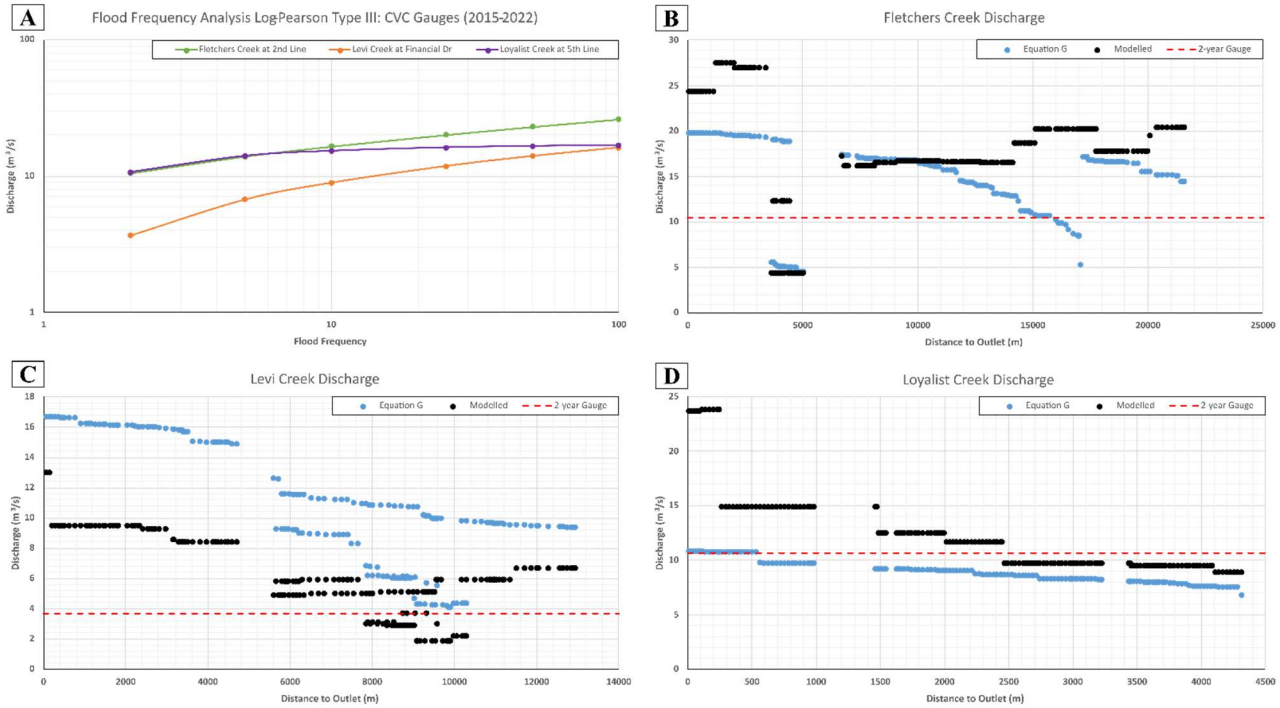


Figure 3: A) Flood Frequency Analysis Log-Pearson Type III CVC Gauges (2015-2022) showing the selected gauges' 2-100 year discharge. The calculated 2-year discharge from Figure A is used in Figures 3B to D to identify whether the provided Modelled data and the predicted discharge using Equation G are overpredicting or underpredicting with reference to the calculated 2-year gauge discharge. B) Fletchers Creek 2-year Discharge Comparison C) Loyalist Creek 2-year Discharge Comparison D) Levi Creek 2-year Discharge Comparison.

Table 11: Model's Probability Of Exceeding The 2-Year Observed Threshold

Watershed	2-Year Observed Discharge Threshold (m^3/s)	Probability of Exceedance of Threshold (%)	
		Modelled	Equation G
Fletchers	10.45	85.53	85.53
Loyalist	10.67	49.69	14.56
Levi	3.66	79.90	100

The results indicate Fletchers Creek outputs for both modelled data and Equation G exceed the 2-year discharge observed threshold 85.53% of the time. For Fletchers Creek, both the modelled data and Equation G exceed the 2-year observed discharge threshold 85.53% of the time. This consistent exceedance suggests that SPINpy 2, combined with Equation G, effectively captures the flood dynamics in these subwatersheds. Furthermore, the CVC has provided geomorphology assessment data for Fletchers Creek which can be overlayed with the SPINpy 2 and modelled results to highlight whether the field data is comparable to the patterns that are predicted (Figure 4).

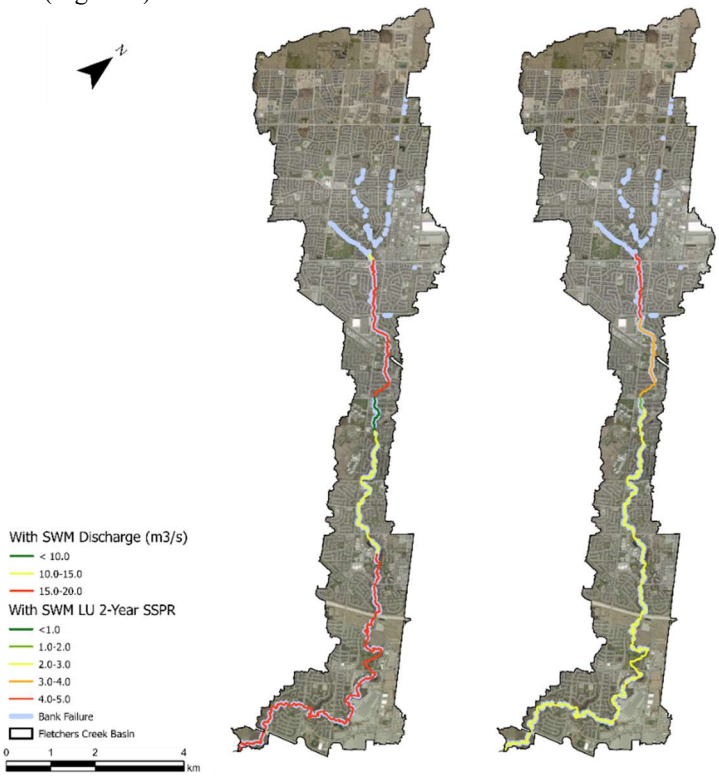


Figure 4: Comparison of Fletchers Creek SPINpy 2 and Modelled to Field Observed Erosion

In contrast, Loyalist Creek presents more complex results. Equation G predicts a relatively low exceedance probability of 14.56%, while the modelled data shows a significantly higher exceedance of 49.69%. The higher exceedance in the modelled data, particularly at the outlet, is reasonable given that the gauge is not near the outlet where higher discharges are typical. However, this discrepancy suggests other spatial or hydrological influences, such as localized runoff patterns or urbanization effects, that are not fully captured in the current

model configuration. This highlights the need for further exploration into these urban hydrological nuances. Expanding the calibration dataset to include additional subwatersheds or finer temporal datasets would allow SPINpy 2 to better account for these variations and improve its predictive accuracy in urban settings.

For Levi Creek, the modelled data shows an exceedance probability of 79.90%, while Equation G predicts 100% exceedance. While both results show high exceedance, the modelled data indicates that SPINpy 2 and the Modelled discharges are overestimating. These results indicate that SPINpy 2 can be further refined to better align Equation G's predictions with the modelled data, particularly in accounting for spatial variations at the outlet.

Overall, the frequency analysis of the three subwatersheds provides insights into SPINpy 2's performance in urbanized environments. Fletchers and Levi Creek show a substantially higher exceedance probability, suggesting that calibrated Equation G may not fully capture the complexities of urban hydrology in these subwatersheds under a single equation. This is due to urban areas being characterized by complex responses to precipitation events, which are often exacerbated by impervious surfaces, altered drainage networks, and channel modifications. These factors can introduce variability that may not be fully accounted for in the current model configuration. Expanding the calibration dataset to include more subwatersheds or finer temporal datasets could help the model better account for these urban-specific nuances, improving the overall predictive accuracy of SPINpy 2. By including additional data points, especially from areas with more complex hydrological responses, the model's robustness can be enhanced, allowing it to more accurately reflect the variability in discharge and channel morphology across urbanized watersheds.

When identifying a general discharge equation, the results for Fletchers and Levi Creeks suggest that SPINpy 2, in combination with Equation G, performs well in capturing flood dynamics. However, the discrepancies observed in Loyalist Creek highlight the need for further calibration. This underscores the importance of refining the model to better capture local variations and improve its applicability in urbanized watersheds. With more data and continued calibration, SPINpy 2 can be further developed to provide more accurate and reliable predictions of discharge exceedances, enabling better management of stormwater and erosion risks in urban environments. Overall, while using three (3) subwatersheds was beneficial, the results highlight the need for more data to calibrate and validate the SWM discharge model. This validation process has provided a critical assessment of the SPINpy 2 model's strengths and areas for improvement. The AMS approach, while appropriate for annual peak analysis, could be complemented with PDS for more frequent flood dynamics, particularly if the objective shifts to understanding urban stormwater management impacts. For now, the chosen methods were effective in verifying the model's reliability in most watersheds, with Fletcher's and Levi Creek meeting expectations. Loyalist Creek's divergence underscores the need for further refinement, which will involve detailed calibration with an expanded dataset, thereby ensuring SPINpy 2 is robustly equipped to support urban watershed management.

A.1.5 LU Urban Width Equation Calibration/Validation Using CVC Field Data

The CVC provided detailed field data for 98 cross-sections and rapidly measured sites for 56 cross-sections across the CVC (Figure 5).

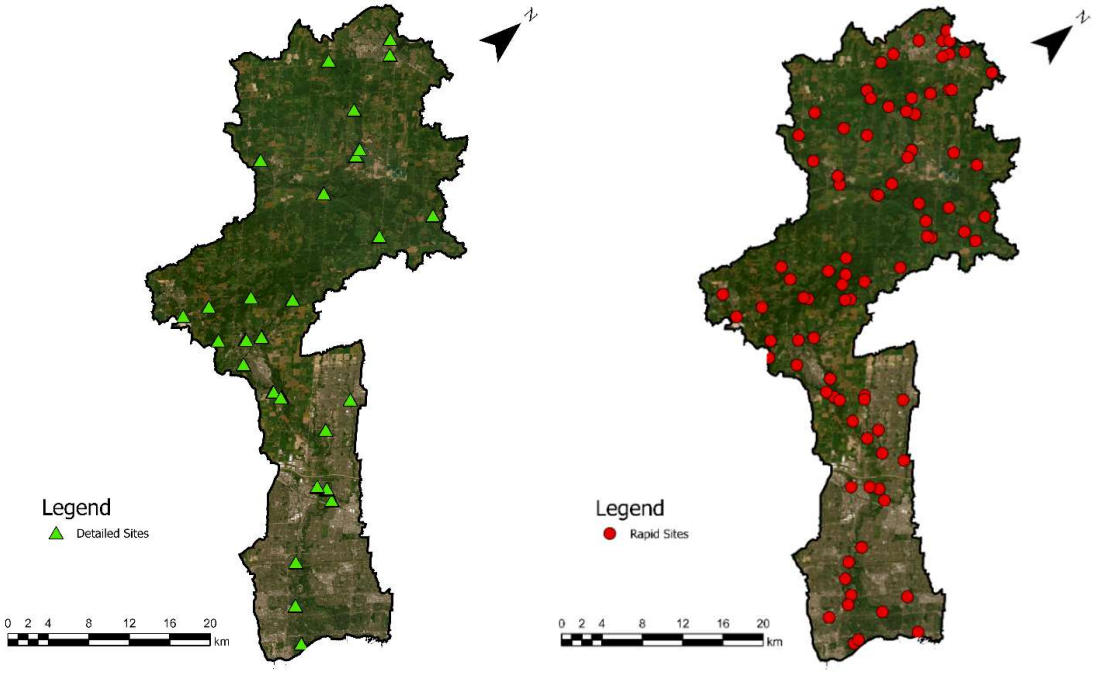


Figure 5: CVC Detailed and Rapid Width Measurement Field Data Sites

Once SPINpy 2 was run for the entire watershed to obtain predicted discharge and stream power metrics. The SPINpy 2 outputs were joined to the field data to narrow down streams in third order and up that can be used for calibration and validation. In total, 98 records were used from the detailed sites for calibration data, and 56 records were used from the rapid sites for validation. SPINarc uses a rural width equation (Equation 3) based on the study of rural streams completed by Phillips and Desloges (2014). The rural channel width (Equation 3) was used as a fixed assumption for the urban scenario in SPINarc, resulting in no change in width. However, this approach does not consider the sensitivity of the channel width to urbanization. In SPINpy 2, we must depict the changing widths due to urbanization. The following section will compare the Phillips-Desloges (Equation 3) and the detailed hydrologic model widths to identify the limitations of Equation 2-6 (rural width equation). Then, the methods of calibrating and validating a new urban width equation using measured field data will be presented.

Figure 6 and 7 show the prediction patterns of the Phillips-Desloges compared to the modelled widths for Cooksville Creek. A comparison of the widths indicates that the Phillips-Desloges prediction significantly underpredicts compared to the CVC's detailed hydrology modelling.

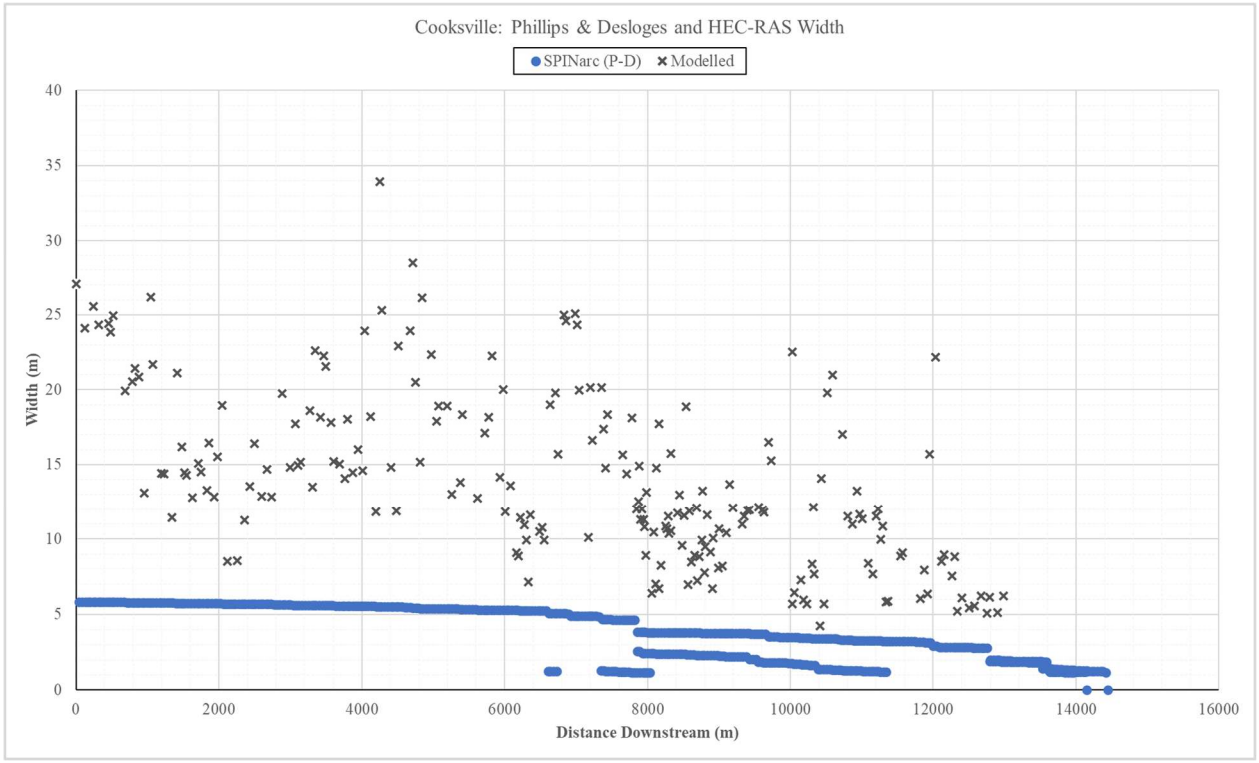


Figure 6: 2-yr Cooksville Creek SPINpy 2 Phillip-Desloges, Barlow-Ashmore Width, and Modelled Comparison

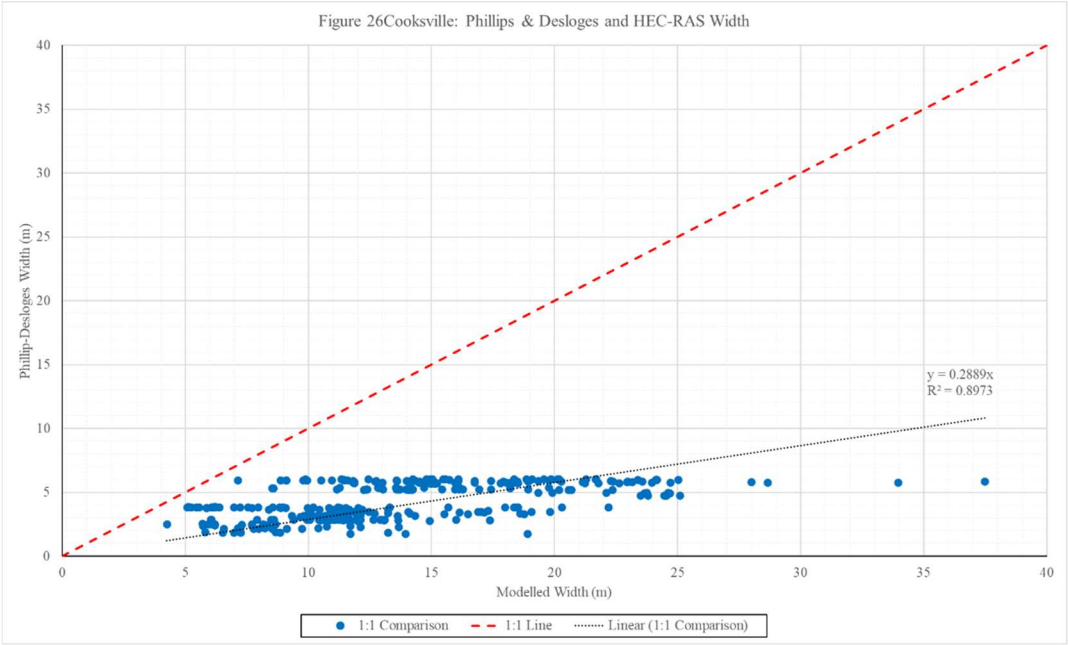


Figure 7: SPINpy 2 Phillip-Desloges and Measured Width 1:1 Comparison

Figure 7 shows a 1:1 comparison between the Modelled data and Phillips-Desloges. The outputs from Equation 3 are compared against the modelled data, and the statistical comparisons are presented below (Table 12).

Table 12: Statistical Metrics of PD Equation (Equation 2-6) Compared to Cooksville Modelled Width

Statistical Measure	Equation 2-6
Mean	4.30
Standard Deviation	1.37
Variance	1.87
S	10.70
PBIAS	-68.70
RMSE	10.63
Pearson Correlation (r)	0.55

While traditional empirical equations provide a foundation, their simplicity may prove inadequate in urban contexts. Therefore, it is observed that there is a need to revise the existing urban width equation to consider the impact of urbanization on channel width. Thus, Barlow-Ashmore's (2023) and Charlton's (2007) width relationships are used as a starting point as they consider stream power and discharge. Which is more sensitive to the effects of urbanization and can better predict changes in channel width. A new modified equation inspired by Barlow & Ashmore (2023) and Charlton (2007) that considers stream power will be calibrated using field data provided by the CVC.

While the detailed hydrologic model was used at the initial stages to compare the equation performances, field data was obtained for equation calibration and validation. The CVC has provided detailed field data for 98 cross-sections and 56 rapidly measured cross-sections across the CVC. The width measurements from the detailed sites will be used for calibration, and the rapid sites will be used for validation. However, we will calibrate the width equation using the measured widths, calculated discharge equation, and stream power.

A.1.6 SPINpy 2 Urban Width Calibration Method

The coefficients of Equation 7 must be calibrated and validated against available field measurements for the Credit River Watershed. An R-script was prepared for a non-linear regression analysis to determine coefficients for each width model considered. The two equations were considered for the calibration, as shown in Table 13**Error! Reference source not found.**.

Table 13: Equations Considered for Calibration

	Equation
J	$W_{urban}=1.16(DA)^{0.508}$
K	$W_{urban} = W_{rural}\,(1 + a(\frac{Q_{urban} - Q_{rural}}{Q_{rural}})^{0.5})$

Equation J is the original W_{PD} equation that was used in SPINarc. Equation J was developed for rural watersheds and uses drainage areas to predict the channel width. Equation K is the new method of predicting urban width in SPINpy 2. Equation K uses a change in discharge (Q) and adds the change to the rural width, where W_{PD} is the rural width, Q_{Urban} is the urban discharge, Q_{Rural} is the rural discharge, and a is the coefficient to be calibrated. For analysis purposes within this thesis, a value of 0.01 will be used for '*a*'. It should be noted

that the discharge is the new calibrated (Equation D). We use a change in Q instead of SP because SPINpy 2 assumes the slopes in rural and urban contexts are the same. Therefore, slope changes are not relevant for calculations in SPINpy 2. If the change in discharge is negative ($Q_{Urban} - Q_{Rural} < 0$), then the equation will be set up within SPINpy 2 to default to W_{Rural}. These three equations are analyzed to identify the most suitable for our field data.

While the urban width equation calibration remains ongoing, a new urban width equation based on stream power has been implemented using the general format shown in Equation K. The urban width equation refinement addresses key limitations related to channel widening and erosion sensitivity in urbanized streams. The updated width model in SPINpy 2 has the potential to improve urban width predictions by reliably predicting channel responses to increased stream power in urban settings. The refinements are especially significant in urban watersheds, where rapid changes in land use and stormwater management necessitate better tools for predicting geomorphological responses.

Appendix B

Results Figure Sets

B.1 NSM Single Watershed Modelled and LU Analysis Outputs

B.1.1 Cooksville Creek Modelled and LU Analysis Outputs

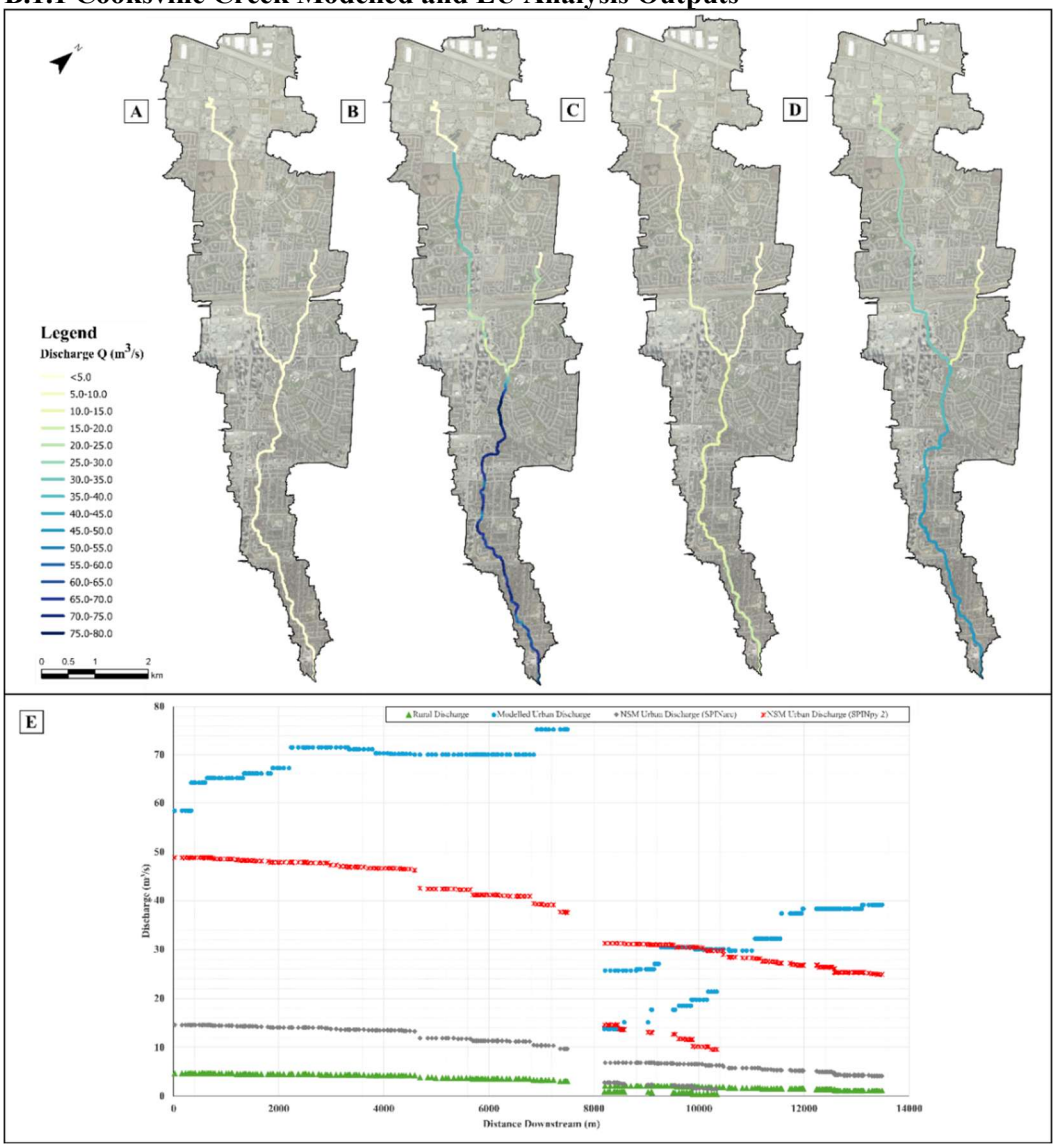


Figure 8: Cooksville Creek Discharge Comparison Under Different Scenarios A) Rural discharge as a baseline; B) Urban modelled discharge using SPINpy 2 Modelled Scenario; C) SPINarc urban land use discharge; D) Revised SPINpy 2 urban land use discharge (NSM Scenario); and E) Comparison between rural and urban discharges across scenarios A-D**.** Figure 8E highlights the increased urban land use discharge precision achieved through the SPINpy 2 refinements.

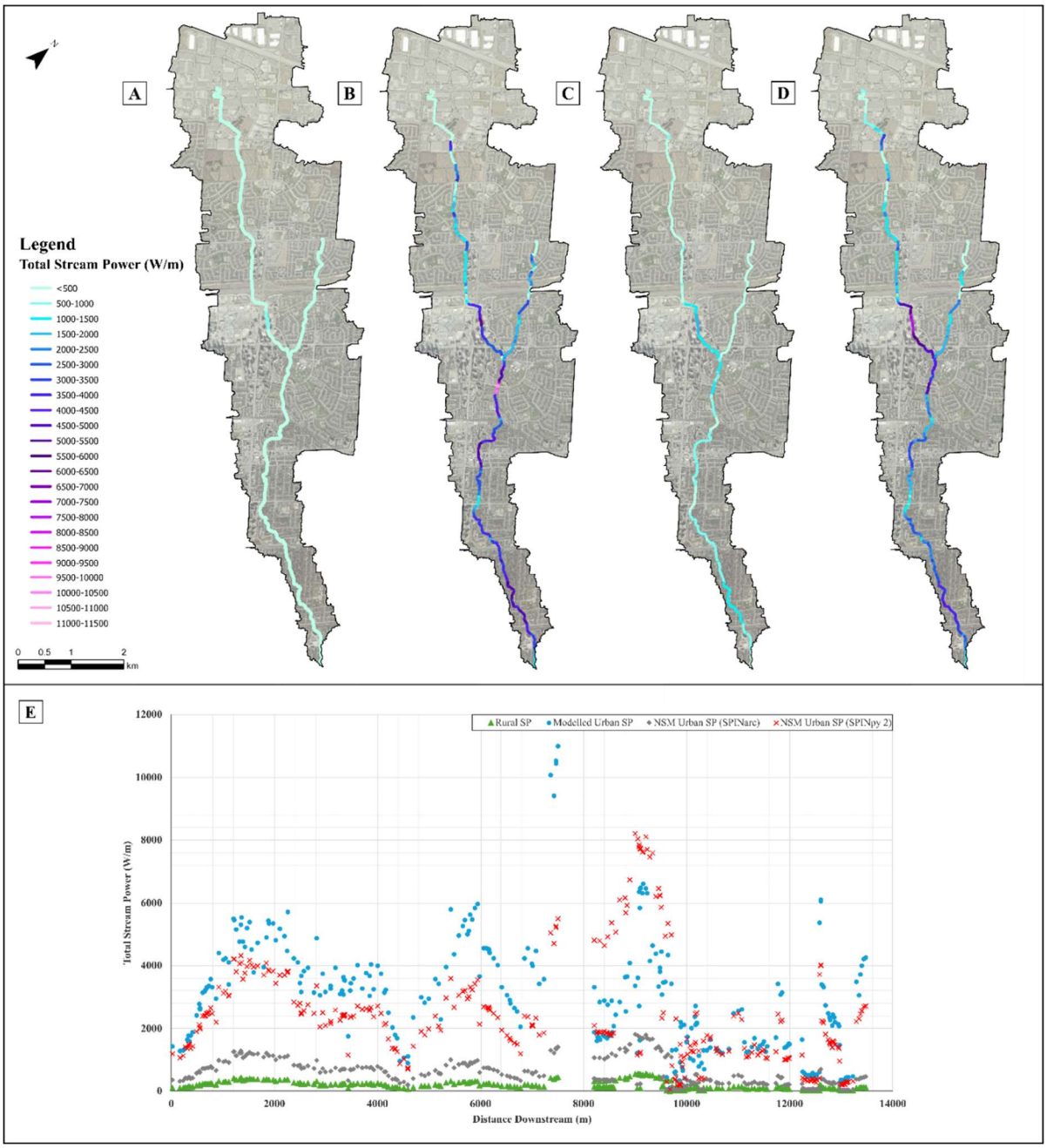


Figure 9: Total Stream Power Comparison for Cooksville Creek Under Different Scenarios A) Rural Total SP; B) Urban Total SP in SPINpy 2 Modelled Analysis; C) SPINarc urban total SP using the LU Scenario; D) Recalibrated urban total SP in SPINpy 2's LU Analysis; E) Displays that the NSM Urban SP can predict total SP closer to the modelled outputs.

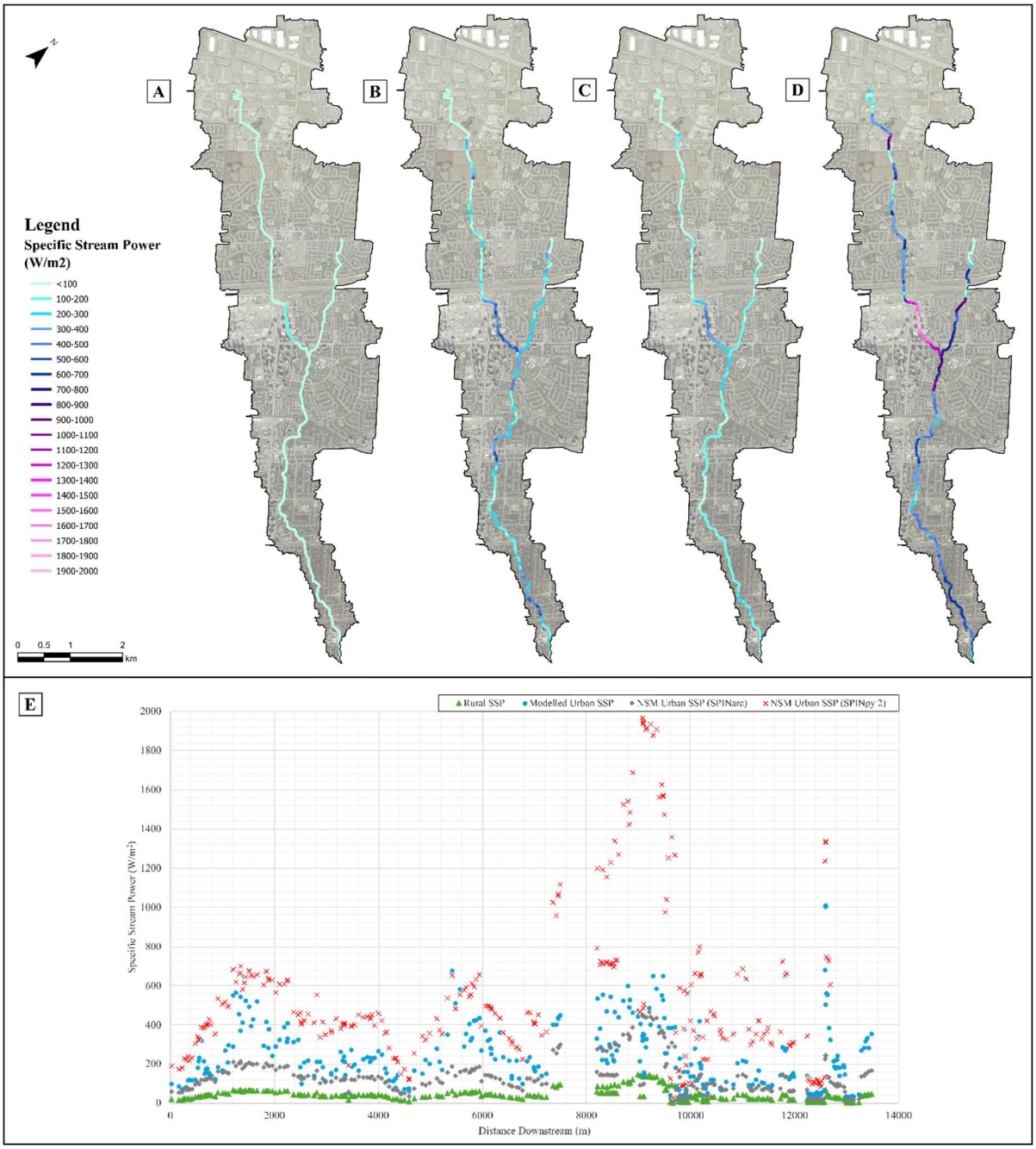


Figure 10: SSP for Cooksville Creek Under Different Scenario A) Rural SSP; B) Urban Total SSP in SPINpy 2 Modelled Analysis; C) SPINarc urban SSP; D) Recalibrated urban SSP for SPINpy 2 in NSM Scenario; E) Comparison of the rural, urban modelled, and land use SSP outputs demonstrates SPINpy 1's refined SSP estimates for the urban Land Use Analysis.

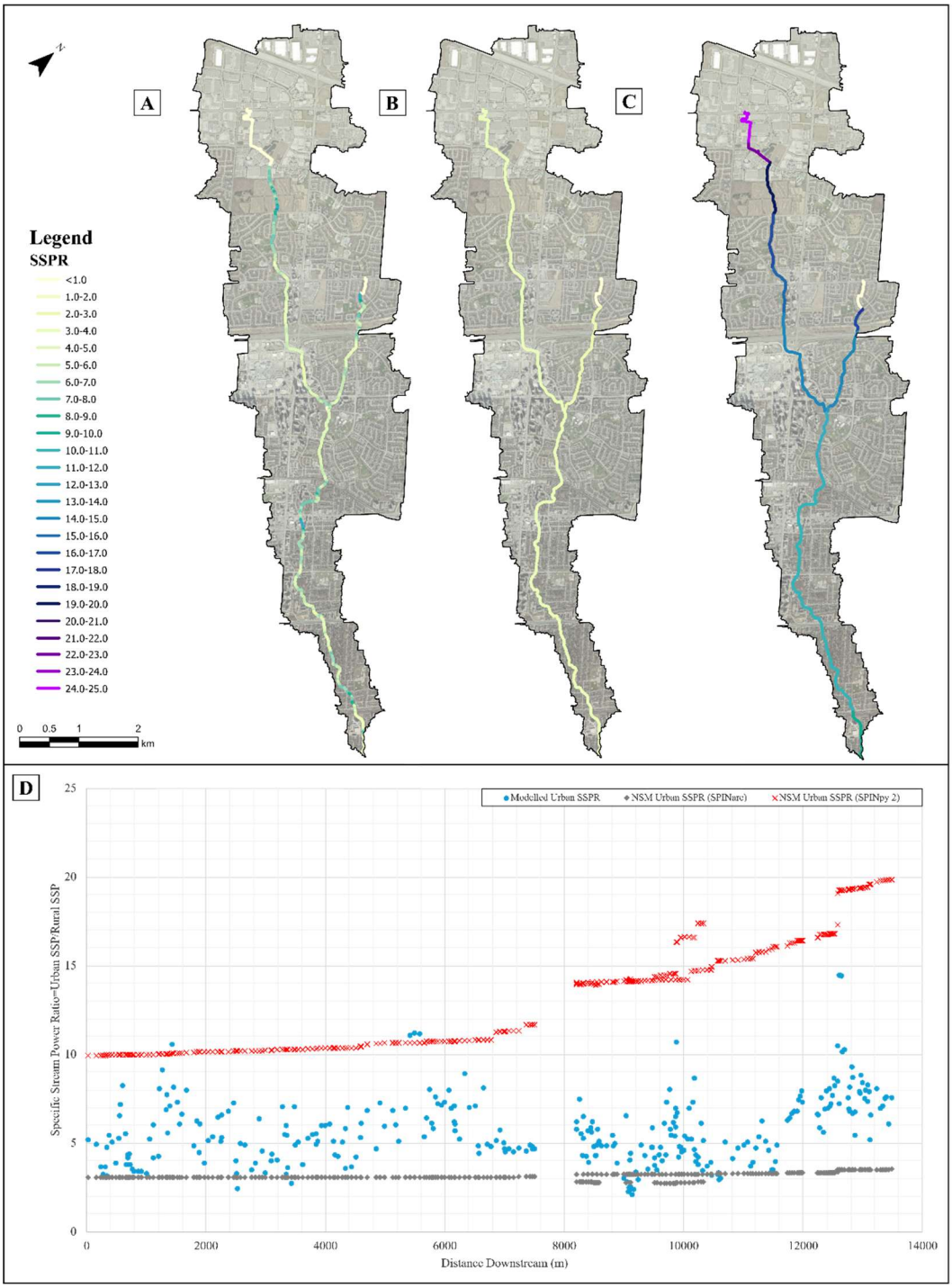


Figure 11: Cooksville Creek SSPR Output Comparison Under Different Scenarios A) Urban modelled SSPR; B) SSPR from SPINarc equations; and C) SPINpy 2 recalibrated SSPR in the NSM Scenario. SPINpy 2's SSPR calculations align better with urban dynamics than prior approaches, aiding in erosion risk assessment.

B.1.2 Levi Creek Modelled and LU Analysis Outputs

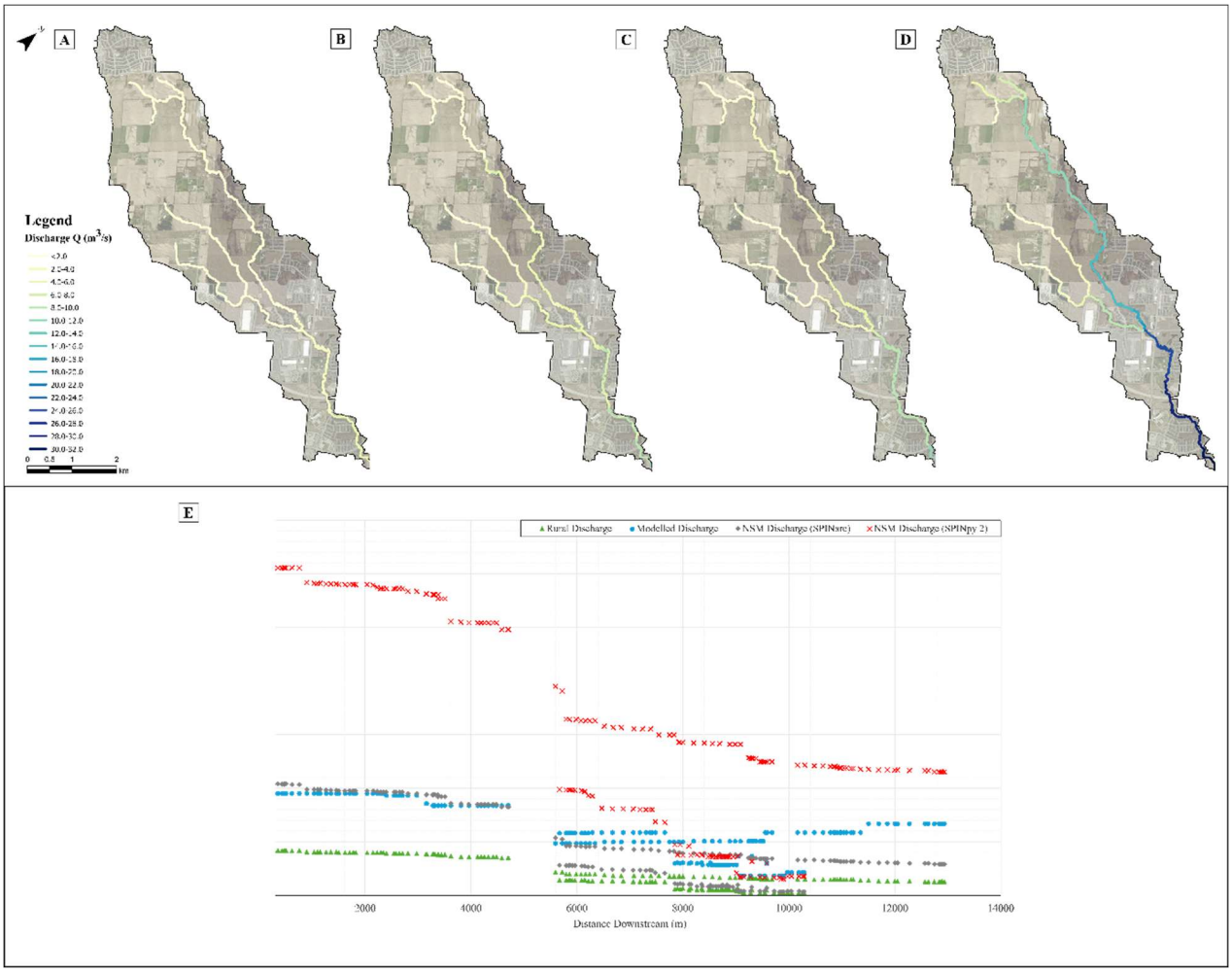


Figure 12: Levi Creek Discharge Comparison Across Scenarios A) Rural discharge baseline; B) Urban discharge modelled with SPINpy 2 Modelled Analysis; C) SPINarc urban discharge; D) Revised SPINpy 2 urban discharge in the LU Analysis; E) Comparison of the rural, urban modelled, and land use discharge outputs. SPINpy 2 shows that the LU Analysis urban discharge predicts larger values than the modelled ones. This is due to the generalizations made in the equation and calibrated across eight different subwatersheds with varying imperviousness.

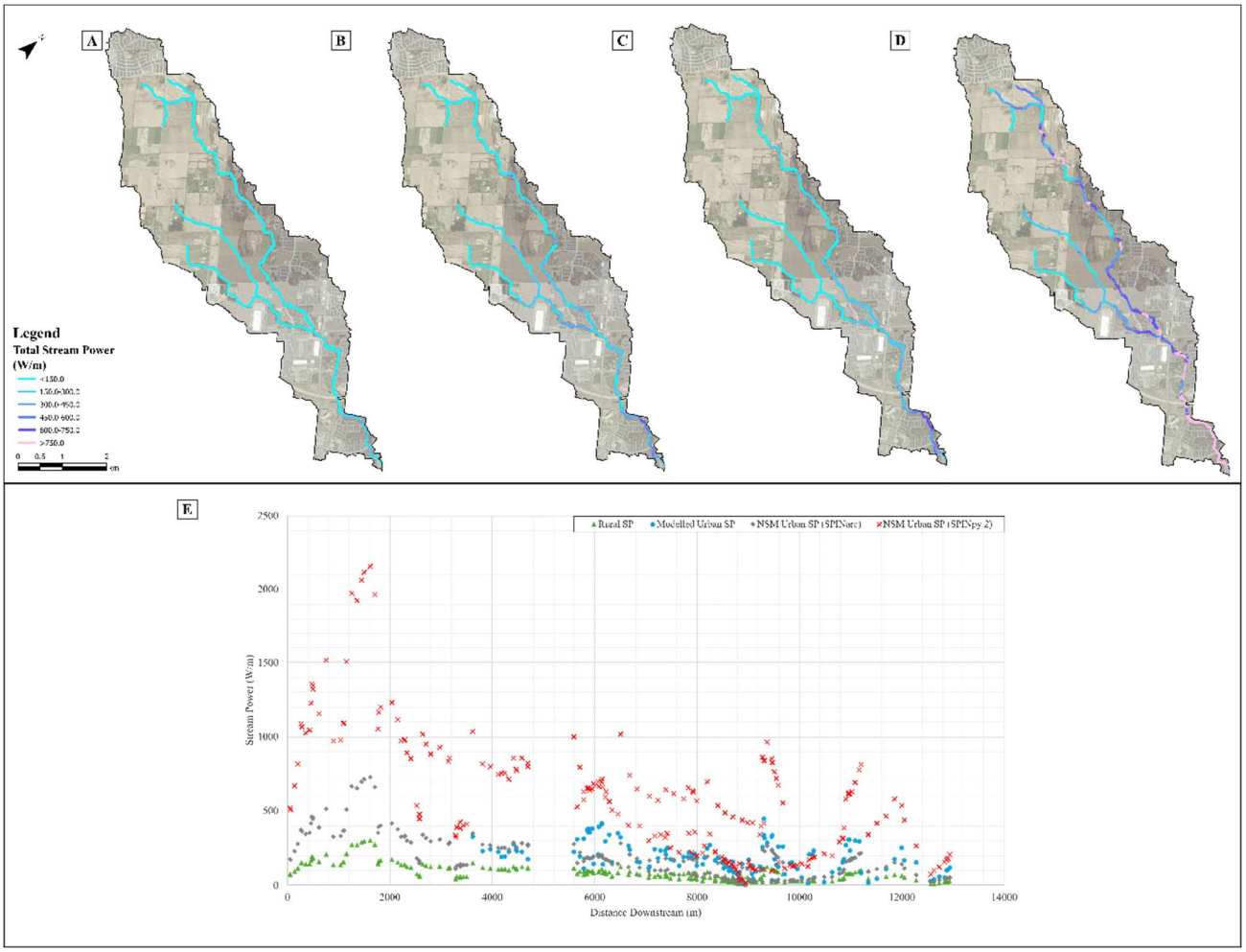


Figure 13: Total SP in Levi Creek Across Different Scenarios A) Rural Total SP; B) Urban Total SP in SPINpy 2 Modelled Analysis; C) SPINarc urban total SP using the LU Analysis; D) Recalibrated urban total SP in SPINpy 2's LU Analysis; E) Displays that the NSM Scenario's urban SP can predict total SP closer to the modelled outputs although it is overpredicting.

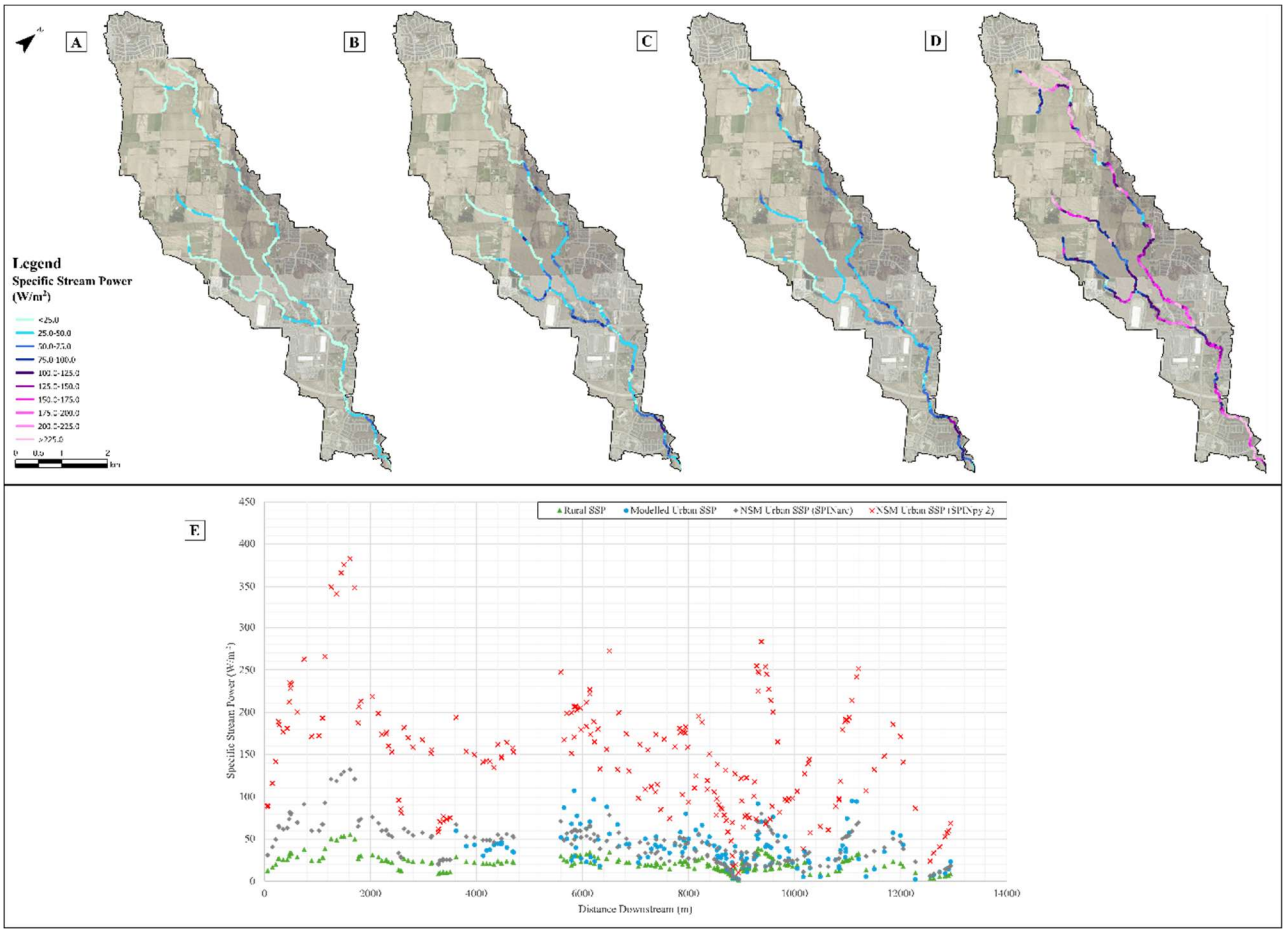


Figure 14: SSP for Levi Creek Under Different Scenarios A) Rural SSP; B) Urban Total SSP in SPINpy 2 Modelled Analysis; C) SPINarc urban SSP; D) Recalibrated urban SSP for SPINpy 2 in NSM Scenario; E) Comparison of the rural, urban modelled, and land use SSP outputs demonstrates SPINpy 1's refined SSP estimates for the urban Land Use Analysis.

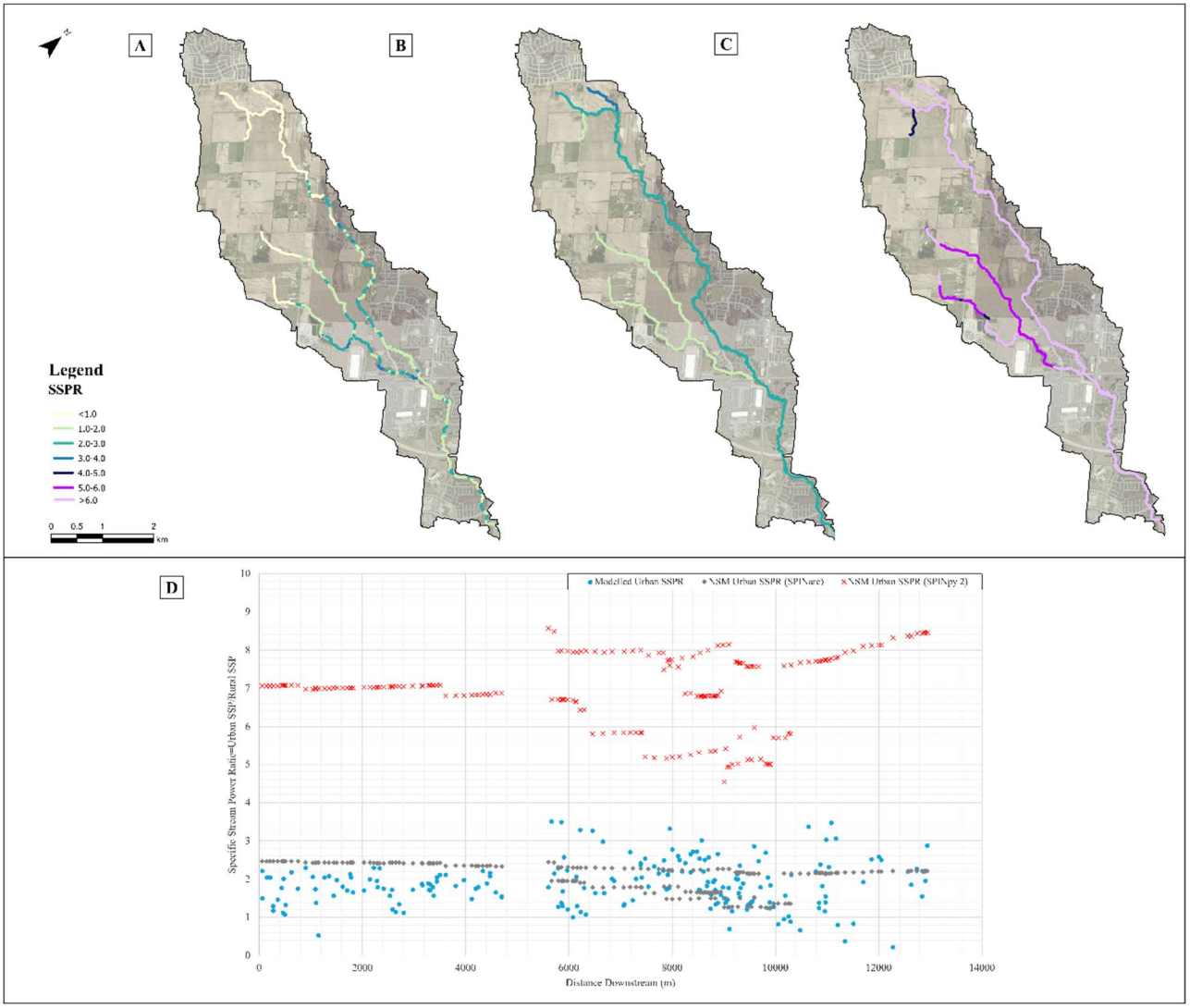


Figure 15: Levi Creek SSPR Output Comparison Under Different Scenarios A) Urban modelled SSPR; B) SSPR from SPINarc equations; and C) SPINpy 2 recalibrated SSPR in the NSM Scenario. SPINpy 2's SSPR calculations align better with urban dynamics than prior approaches, aiding in erosion risk assessment.

B.2 ESM Single Watershed Analysis Modelled and LU Analysis Outputs: Levi Creek

The single watershed ESM scenario was run for the Levi Creek subwatershed. The inputs and outputs for the scenario are shown and discussed below. Using the modified D8 Mass Flux method mentioned above, the presence of the pond reduces the discharge.

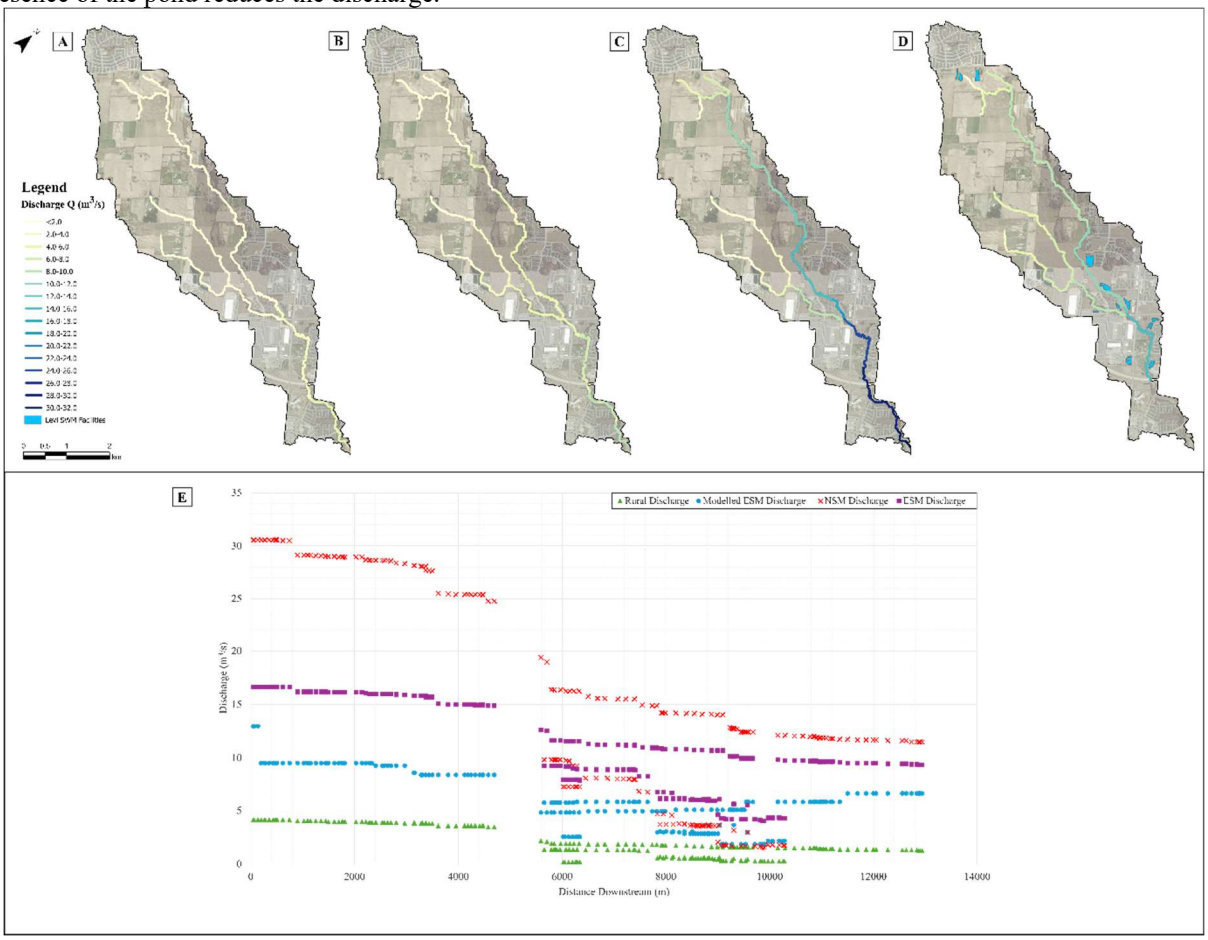


Figure 16: A) Rural discharge B) ESM urban modelled discharge C) Urban land use discharge without the presence of ponds (NSM Scenario) D) Urban land use discharge with consideration of ponds (ESM Scenario) E) Comparison of the rural, ESM urban modelled, and land use discharge outputs.

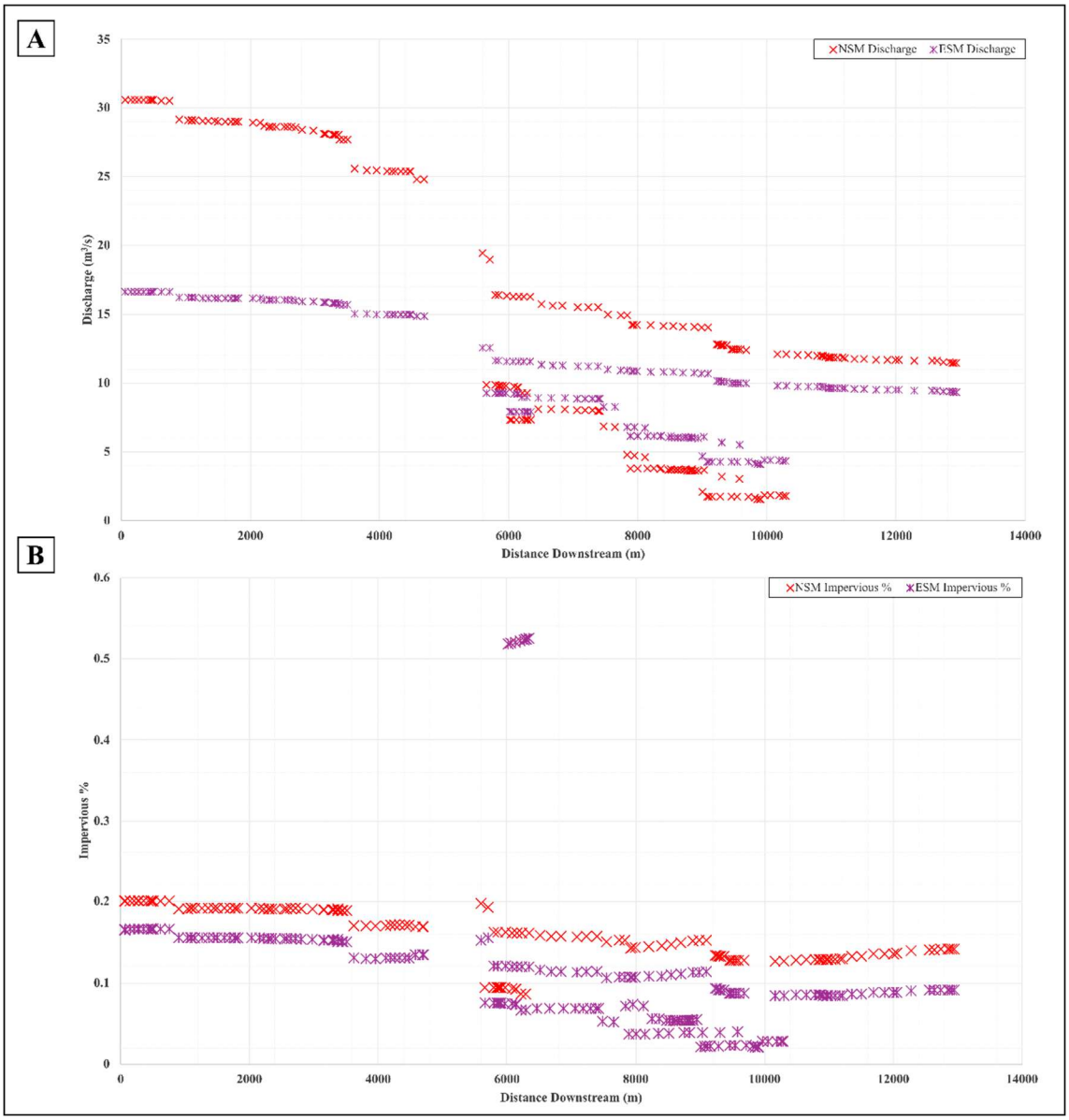


Figure 17: Comparison of the NSM and ESM discharge and impervious percent. SPINpy 2 is most suitable for predicting patterns of high erosion risk. The above comparison shows the effect of the reduced IA on the discharge predictions.

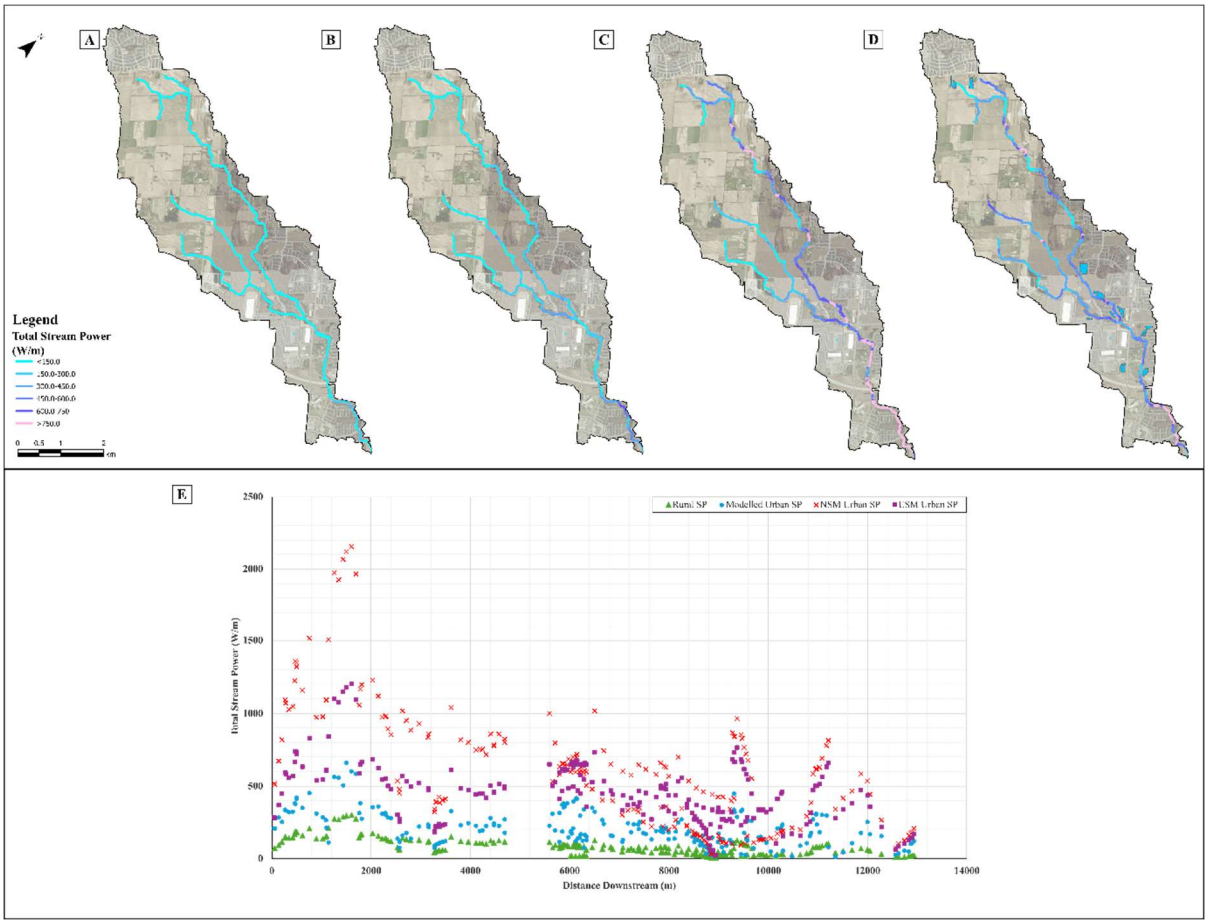


Figure 18: A) Rural total SP B) Urban ESM modelled total SP C) Urban land use total SP in the NSM Scenario D) Urban land use total SP in the ESM Scenario E) Comparison of the rural, ESM urban modelled, and land use total SP outputs.

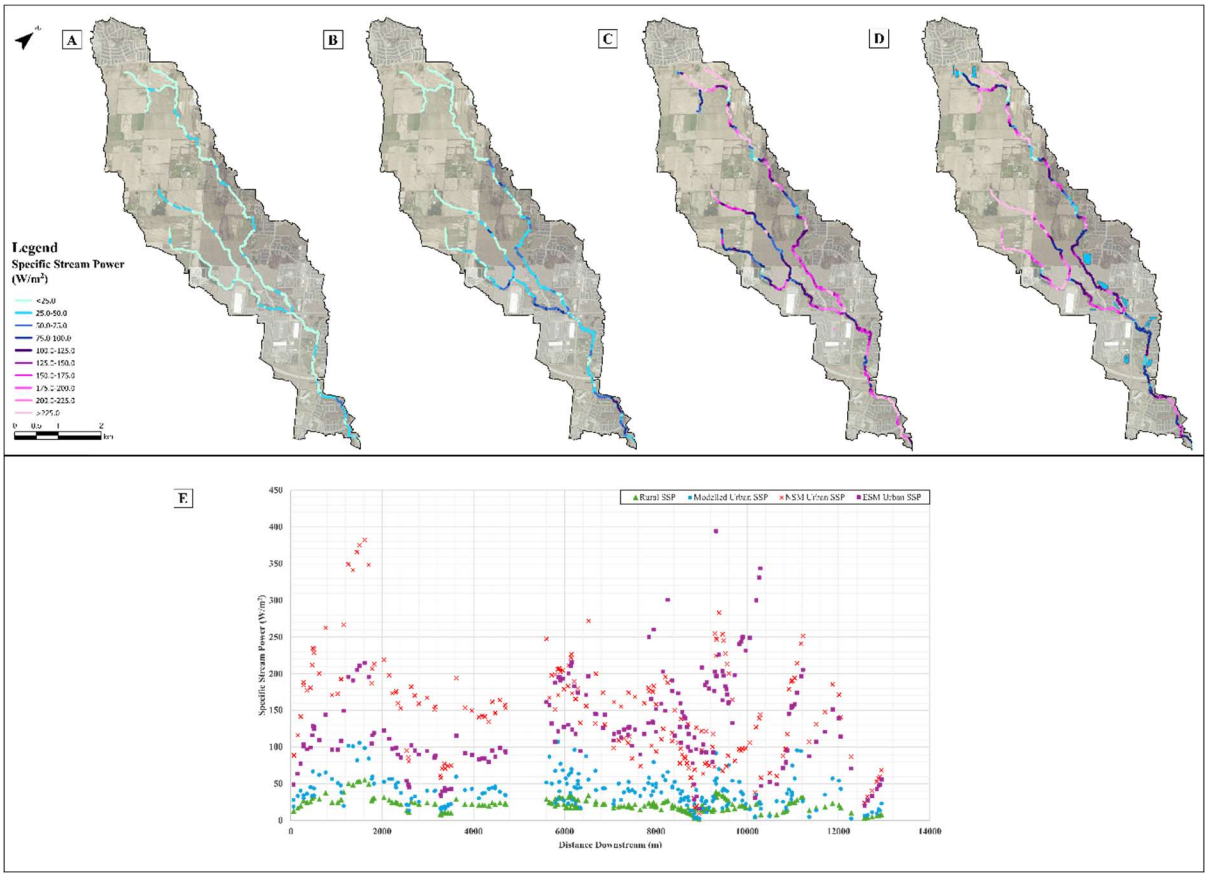


Figure 19: A) Rural SSP B) Urban ESM modelled SSP C) Urban land use SSP in the NSM Scenario D) Urban land use SSP in the ESM Scenario E) Comparison of the rural, ESM urban modelled, and land use SSP outputs.

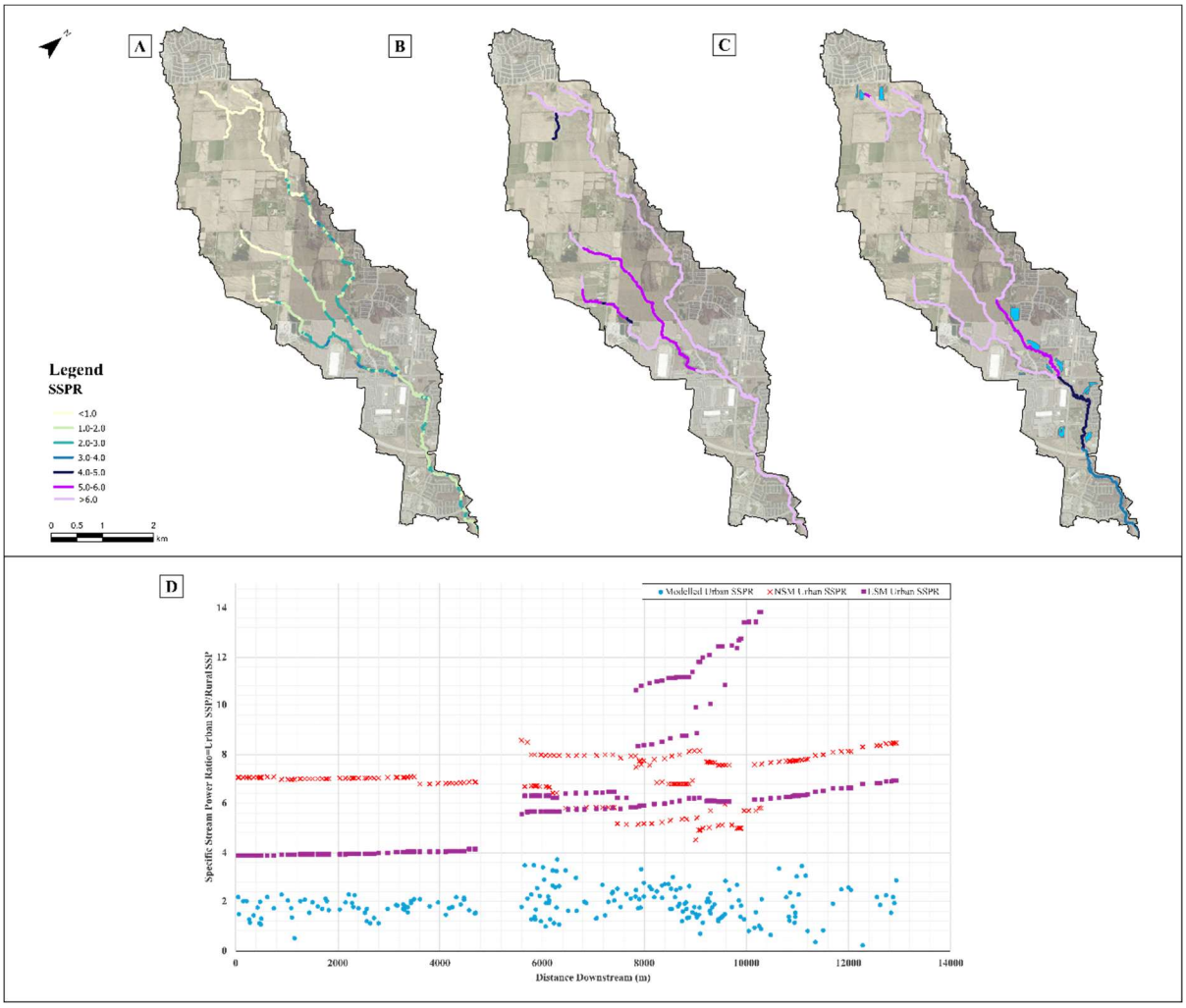


Figure 20: A) Urban ESM modelled SSPR B) Urban land use SSPR in the NSM Scenario C) Urban land use SSPR ESM Scenario

B.3 NSM Multiple Watersheds Scenario within a Boundary: Serson and Applewood Creeks

The multiple watershed NSM Scenario was run for subwatersheds within the Mississauga Ward 1 boundary. The inputs and outputs for the analysis are shown and discussed below. Refer to the study area descriptions in Chapter 3.

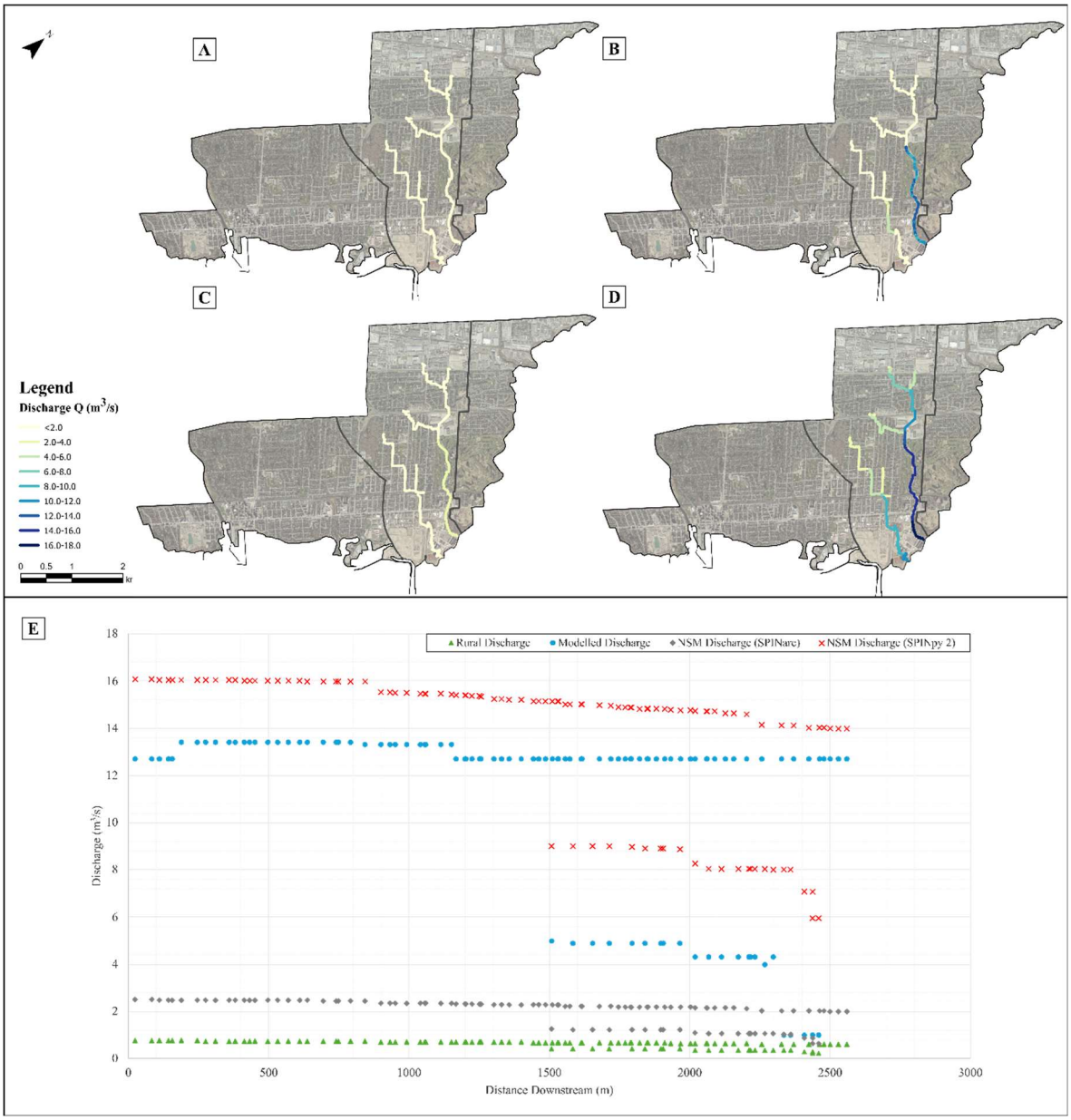


Figure 21: A) Rural discharge B) Urban modelled discharge C) Urban land use discharge - previous version (SPINarc) D) Urban land use discharge (NSM) (revised in SPINpy 2) E) Comparison of the rural, urban modelled, and land use discharge outputs.

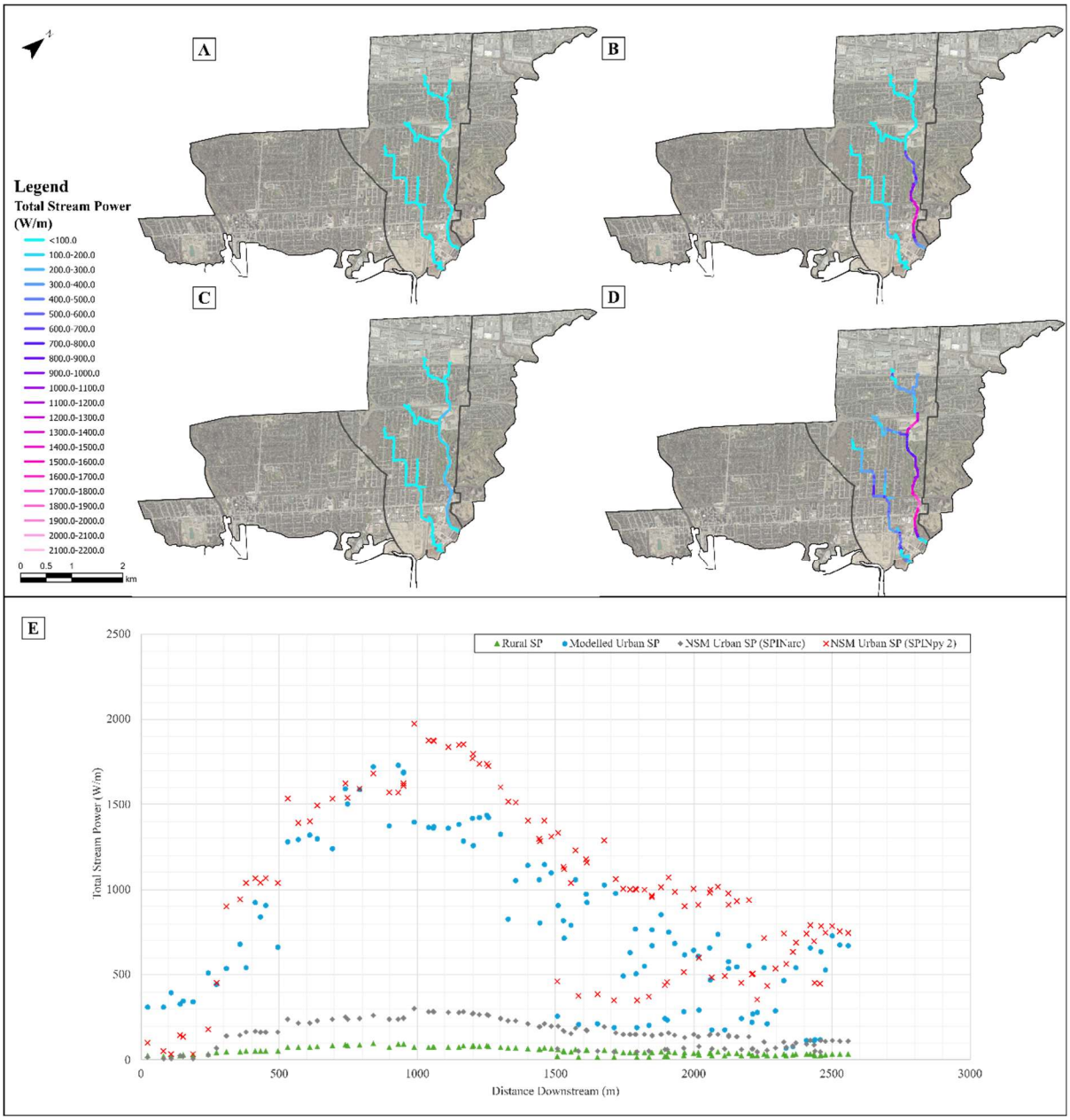


Figure 22: A) Rural total SP B) Urban modelled total SP C) Urban land use total SP with the SPINarc equations D) Urban land use total SP using the recalibrated equation for SPINpy 2 in the NSM Scenario E) Comparison of the rural, urban modelled, and land use total SP outputs.

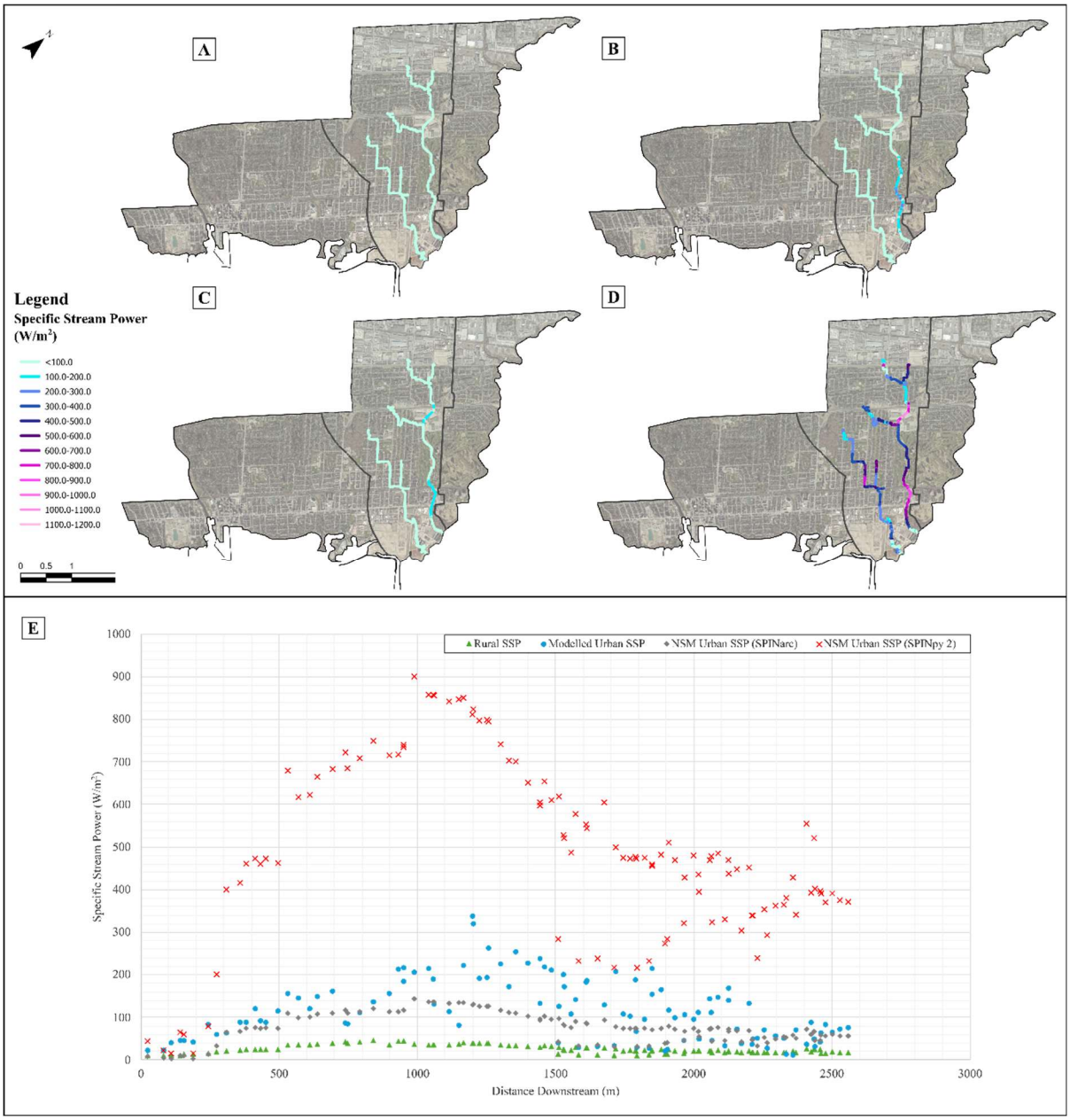


Figure 23: A) Rural SSP B) Urban modelled SSP C) Urban land use SSP with the SPINarc equations D) Urban land use SSP using the recalibrated equation for SPINpy 2 in the NSM Scenario E) Comparison of the rural, urban modelled, and land use SSP outputs.

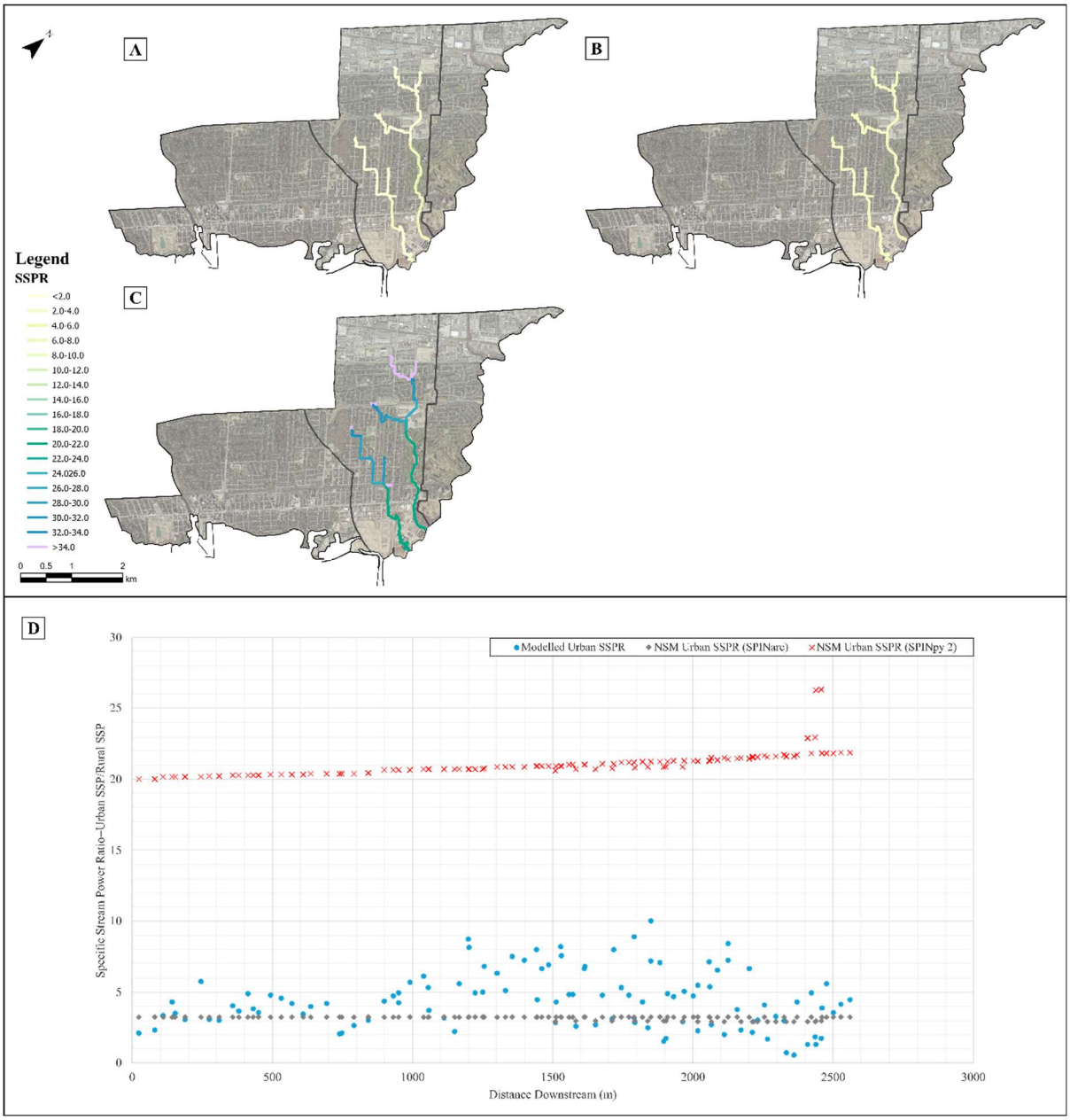


Figure 24: A) Urban modelled SSPR B) Urban land use SSPR with the SPINarc equations C) Urban land use SSPR using the recalibrated equation for SPINpy 2 in the NSM Scenario

Appendix C
SPINpy 2 Instruction Manual

SPINpy 2 Instruction Manual

Thiruni Thirimanne (E.I.T), Kimisha Ghunowa (MASc), Dr. Bruce MacVicar (Ph.D., P.Eng)
SPINpy 2 (October 11, 2024)

SPINpy 2 is the updated version of the SPIN Toolbox (https://github.com/macvicab/SPIN). The updates to the SPIN tool address the memory constraints that were faced when using the SPIN Toolbox in ArcMap.

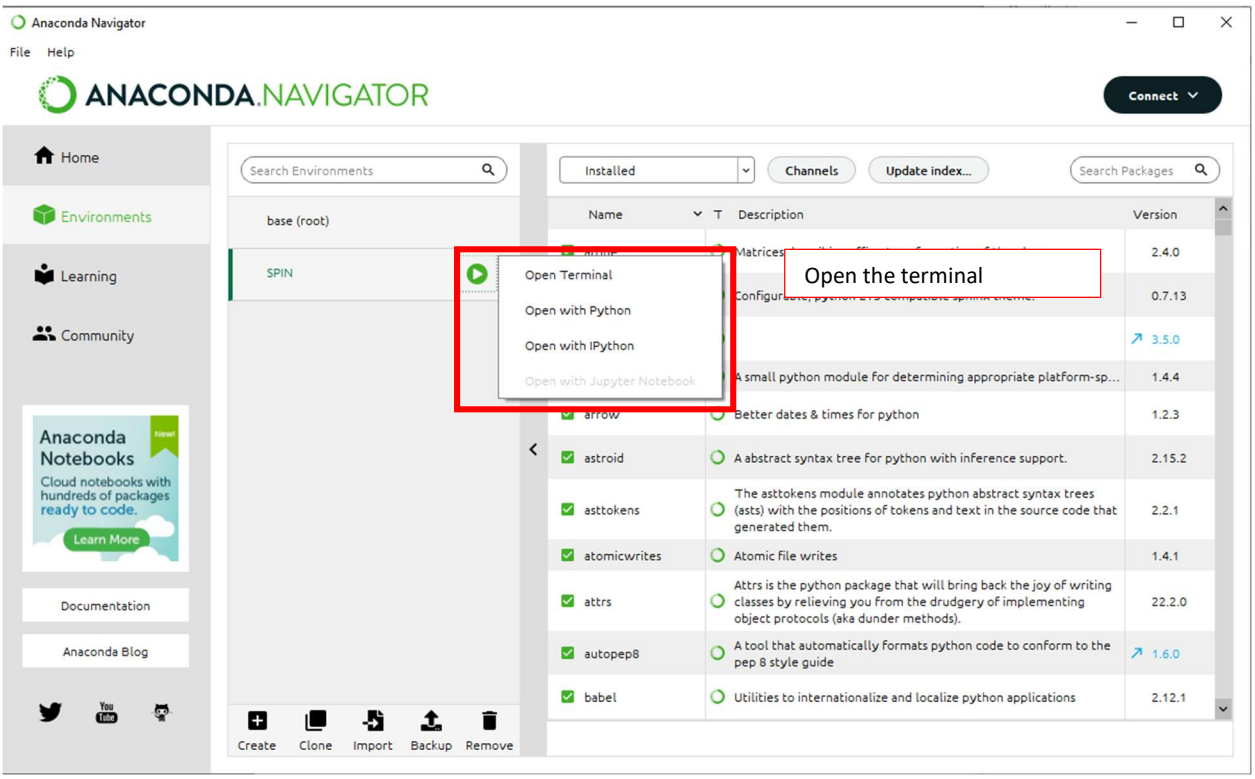
SPINpy 2 is an open-source tool that allows users to run stream power analyses using two input data types (detailed hydrologic model or Land Use data or both). The tool is built using Python and takes inputs using an Excel interface.

When running analyses using SPINpy 2 it is important to remember that the processing time is dependent on the size/detail of the provided data, the computer hardware and software.

Setting Up Python Environment

To run the downloaded script, it is necessary to replicate the Anaconda (Python) environment used. Provided within this download is the environment file. Follow these steps to recreate the environment:

- Ensure that Anaconda is installed on the machine https://www.anaconda.com/products/individual
- Unzip the folder downloaded from GitHub, and ensure that the "environment.yml" file is present.
- Open the Anaconda Prompt (terminal window)

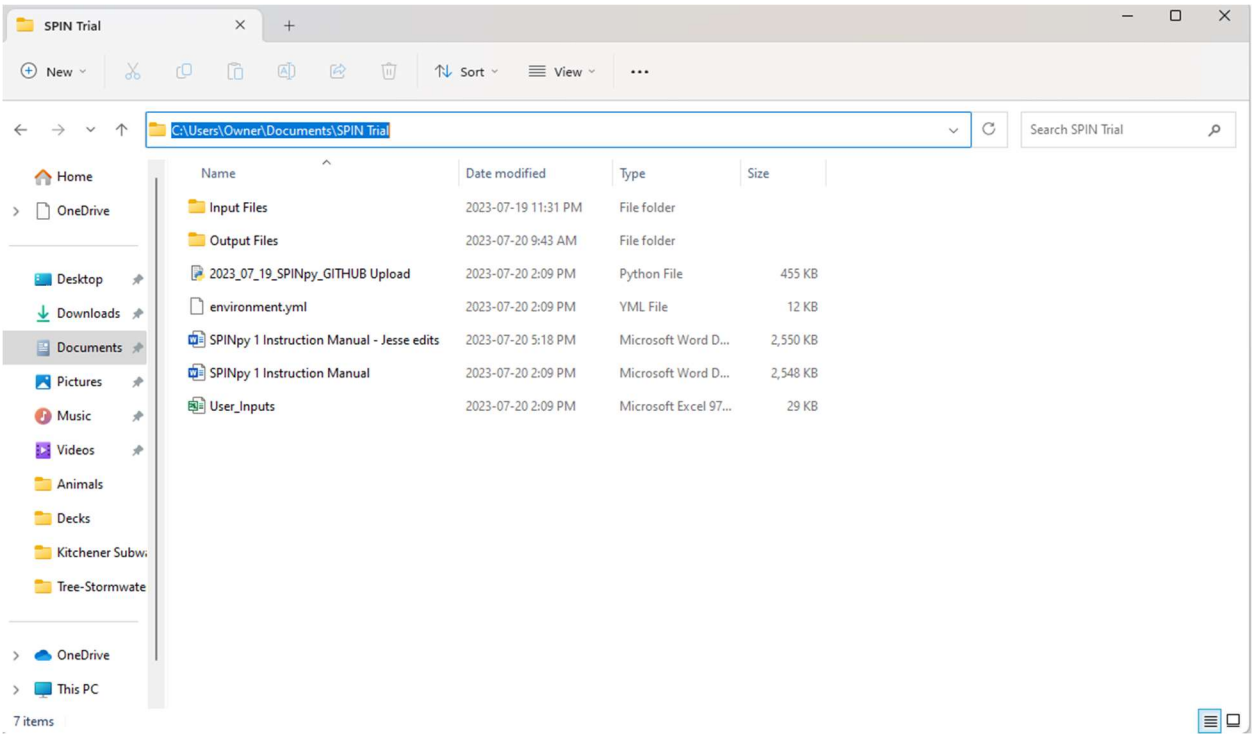


- Once the terminal window opens, type the following:

  "cd /path/to/directory"

Replace the path to the directory with where your unzipped folder is saved.

For example: cd C:\Users\Owner\Documents\SPIN Trial



- Now you can create a new environment based on the file by typing in the following:

  conda env create -f environment.yml

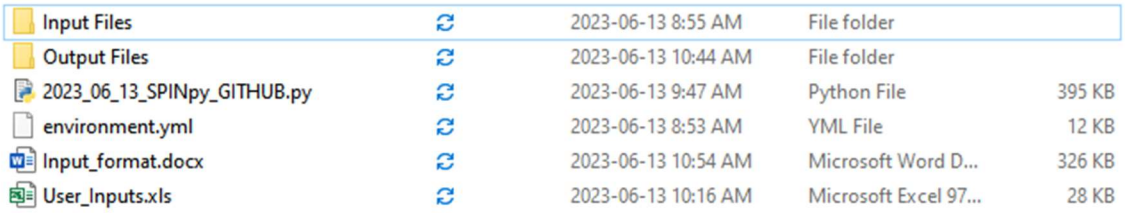
  This creates a new environment with the same packages and version that are specified in the environment.yml file. The environment.yml file will have a section with the name of the environment.

- To activate the environment, type the following:

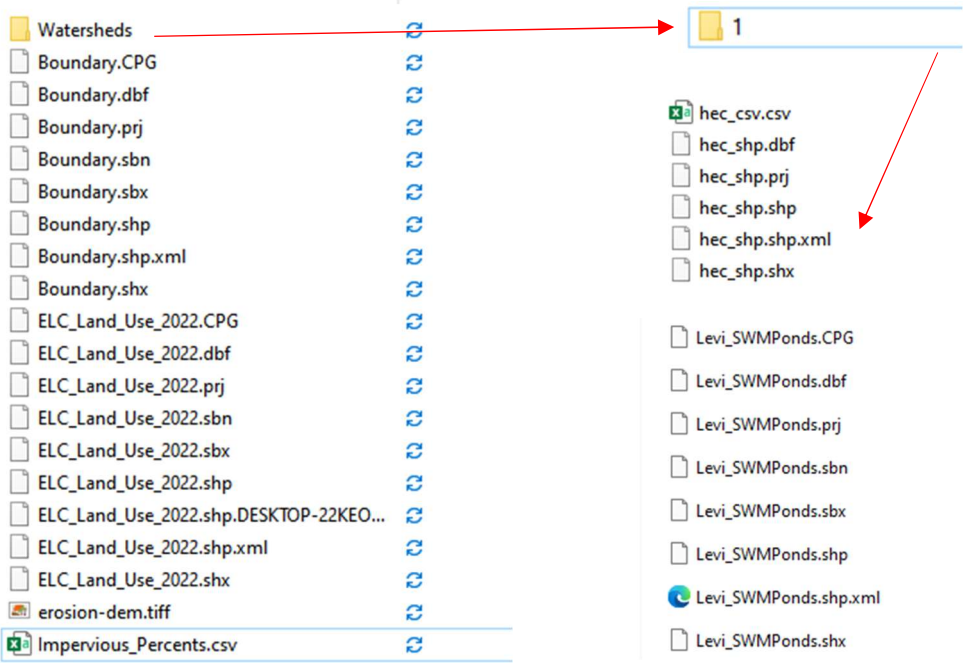
  conda activate SPIN

Folder Organization

General Folder Layout:



Input Files:



- Modelled Stream Power Analysis:
  - If running a modelled analysis (using outputs from a program such as HEC-RAS), please save the formatted files for each watershed in folders within the “Watersheds” folder labelled from 1 onwards (refer to the figure above).
  - If running it for multiple watersheds within a boundary, multiple watersheds can be saved under separate folders.
  - If a pond analysis is to be done in the modelled analysis in SPINpy 2, ensure the provided modelled data outputs have an analysis where ponds are in the model.
- Land Use Stream Power Analysis:
  - A shapefile that contains land use information and a digital elevation model raster (DEM) is necessary. Please ensure that both the Land Use information and the DEM cover the area of interest.
  - An Excel file of the impervious percent associated with the Land Use Types in the Land Use Layer is also necessary. There should only be two columns, “TYPE” and “Imperviousness”.

- "Levi_SWM_Ponds.shp" -If pond footprints are available and the user would like to analyze the effect of ponds, please upload a shapefile with polygons of the pond.
- "Sample Points.csv" - A Excel file that contains X Y coordinates of key watercourses to capture. This will help the tool select its stream-pruning threshold.

The user can save a boundary shapefile if they would like to run the program to capture information within a given boundary which will result in multiple watersheds being analyzed.

Output Files:

It should be set up as an empty folder; this is the working directory and where everything will be written.

2024_10_04_SPINpy_GITHUB.py:

This is the Python file that contains the SPINpy 2 script. You can open this using the Spyder IDE program within Anaconda. Once this program is run, the outputs will be written to the "Output Files" folder.

User_Inputs.xls:

	A	B	C	D
1	**Required Parameters**	**Type**	**Default**	**Input**
2	Working Directory	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Output Files
3	Enforced DEM (.tiff)	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Input Files\\erosion-dem.tif
4	Cell Resolution of DEM (integer)	integer value		1
5	Watershed Input Information	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Input Files\\Watersheds
6	HECRAS Analysis (Y/N)	boolean	Y	Y
7	Bridge Analysis (Y/N)	boolean	N	N
8	Culvert Analysis (Y/N)	boolean	N	N
9	Subwatershed Analysis (Y/N)	boolean	N	N
10	Subwatershed as Polygon Feature (.shp) if Subwatershed Analy	path		
11				
12	**Pre-defined Parameters**			
13	Initial Drainage Threshold for Stream Extraction (m^2)	integer value	1000	1000
14	Rural Discharge Coefficient ($Q=xA^{}$)	float value	0.25	0.25
15	Rural Discharge Exponent ($Q=xA^{t}$)	float value	0.91	0.91
16	Rural Width Coefficient ($W=bA^{}$)	float value	1.16	1.16
17	Rural Width Exponent ($W=xA^{e}$)	float value	0.51	0.51
18	Slope Smoothing Distance (m)	integer value	500	500
19	Slope Smoothing Threshold for Drainage Difference Ratio	float value	0.10	0.10
20	User Desired Pruning Threshold Area (m2). Use this value if the	integer value	100000.00	100000.00
21	**Land Use**			
22	Land Use Analysis (Y/N)	boolean	Y	Y
23	Impervious Percent Raster (Y/N)	boolean	N	N
24	Impervious Percent Raster (.tiff) Raster cells must be from 0 -10	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Input Files\\ImpRaster2.tif
25	Cell Resolution of DEM (integer)	integer value		1
26	Land Use Shapefile (Y/N)	boolean	Y	Y
27	Land Use Shapefile	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Input Files\\ELC_Land_Use_2022.shp
28	Land Use Shapefile Imperviousness	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Input Files\\Impervious_Percents.csv
29	Clipping Boundary	path		
30	Sample points along stream for LU Scenario	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Input Files\\SamplePointsLU.csv
31	Urban Discharge Coefficent ($Q = 0.248(DA(1-IA))^{0.91} + c(DA(IA))^{d}$)	float value	9.14	9.14
32	Urban Discharge Coefficent ($Q = 0.248(DA(1-IA))^{0.91} + c(DA(IA))^{d}$)	float value	0.61	0.61
33	Urban Width Coefficient ($W=W_{rural}(1+c(Q_{urban}-Q_{rural})^{0.5}$	float value	0.01	0.01
34	Engineered SWM Solutions (Y/N)	boolean	Y	Y
35	Engineered Ponds	path		C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\Masters\\Input Files\\Levi_SWMPonds.shp
36	Urban Discharge Coefficent ($Q = 0.248(DA(1-IA))^{0.91} + c(DA(IA))^{d}$)	float value	9.96	9.96
37	Urban Discharge Coefficent ($Q = 0.248(DA(1-IA))^{0.91} + c(DA(IA))^{d}$)	float value	0.138	0.138
38				
39				

This Excel is used to provide information to the Python script:

Working Directory	Provide the file path to the output folder
Enforced DEM (.tiff)	Provide the file path to the erosion DEM raster
Cell Resolution of DEM (integer)	Provide the cell resolution of the DEM (which can be checked using GIS)
Watershed Input Information	File path to the "Watersheds" folder
HECRAS Analysis (Y/N)	Y/N whether this analysis should be performed
Bridge Analysis (Y/N)	Y/N whether this analysis should be performed
Culvert Analysis (Y/N)	Y/N whether this analysis should be performed
Subwatershed Analysis (Y/N)	Y/N whether this analysis should be performed

Subwatershed as Polygon Feature (.shp) if Subwatershed Analysis = Y	File path for the polygon for the subwatershed analysis.
Initial Drainage Threshold for Stream Extraction (km2)	Value for drainage threshold
Rural Discharge Coefficient (Q=**x**A^c)	Rural discharge coefficient (x). Please enter this regional value as this also makes up a component of the urban discharge equation.
Rural Discharge Exponent (Q=xA^**c**)	Rural discharge exponent (c). Please enter this regional value as this also makes up a component of the urban discharge equation.
Rural Width Coefficient (W=bAe)	Coefficients for the rural width (Phillips & Deloges equation)
Rural Width Exponent (W=xAe)	Coefficients for the rural width (Phillips & Deloges equation)
Slope Smoothing Distance (m)	Integer value for slope smoothing.
Slope Smoothing Threshold for Drainage Difference Ratio	Float value for slope smoothing differences.
Land Use Analysis (Y/N)	Y/N whether this analysis should be performed
Impervious Percent Raster (Y/N)	Y/N whether the user has a land use shapefile to use in the analysis. Additional processing is necessary for this format.
Impervious Percent Raster file path	File path to the impervious percent raster. Raster has cell values from 0-100.
Impervioius Percent Raster cell size	Cell size of the raster.
Land Use Shapefile (Y/N)	Y/N whether the user has a land use shapefile to use in the analysis. Additional processing is necessary for this format.
Land Use Shapefile file path	File path to the land use shapefile
Land Use Imperviousness	File path to the excel with the impervious values
Clipping Boundary	If the analysis is to be performed for a specific area (and multiple watershed) please provide the file path for the boundary. If it is a single watershed, please leave this blank.
Sample points along stream for LU Scenario	If running the land use for a single watershed, please provide a longitude and latitude coordinates of key watercourse headwaters that you are interested in.
Urban Discharge Coefficient (Q = 0.248(DA(1-IA))0.91 + **c**(DA(IA))d)	The c coefficient for the urban discharge equation exponent is 9.14. If there is another coefficient that you would like to use, please update it here in the Excel sheet.
Urban Discharge Coefficent (Q = $0.248(DA(1-IA))^{0.91} + c(DA(IA))^{\mathbf{d}}$)	The d coefficient for the urban discharge equation exponent is 0.81. If there is another coefficient that you would like to use, please update it here in the Excel sheet.

Urban Width Coefficient ($W=W_{rural}(1+\mathbf{c}(Q_{urban}-Q_{rural})^{0.5}$	The c coefficient for the urban w equation exponent is 0.01. If there is another coefficient that you would like to use, please update it here in the Excel sheet.
Engineered SWM Solutions (Y/N)	Y/N whether this analysis should be performed
Engineered Ponds	File path to the shapefile that contains the ponds within the watershed
SWM Urban Discharge Coefficent ($Q = 0.248(DA(1-IA))^{0.91} + \mathbf{c}(DA(IA))^{d}$)	The c coefficient for the urban discharge equation exponent is 9.96. If there is another coefficient that you would like to use, please update it here in the Excel sheet.
SWM Urban Discharge Coefficent ($Q = 0.248(DA(1-IA))^{0.91} + c(DA(IA))^{\mathbf{d}}$)	The d coefficient for the urban discharge equation exponent is 0.138. If there is another coefficient that you would like to use, please update it here in the Excel sheet.

Setting Up Scenarios

Single Watershed

Modelled Stream Power Analysis Input Data Format

Please save all the files for each watershed within its own respective folder and **follow the naming conventions** for each type of file.

File Name:

- "hec_csv.csv" - hecras stations data
- "hec_shp.shp" - hecras as points shapefile
- "bridge_hec_csv.csv" - hecras bridge data
- "bridge_point_raw.shp" - hecras bridge data as shapefiles
- "culv_hec_csv.csv" - culvert data
- "culv_point_raw.shp" - culvert data as a shapefile

Digital Elevation Model, DEM (.tif)

The DEM must be enforced and must have a projection (e.g. NAD_1983_UTM_Zone_17N EPSG Code: 26917).

HecRas Flood Event Data (.csv)

- These columns must be provided. Columns in bold must exist and contain values (or padded with 0s). Other columns can be ignored.
  1. **Reach**
  2. **River Sta**
  3. **Profile**
  4. **Top W Chnl**
  5. **Q Total**
  6. **Q Channel**
  7. **Top Width**
  8. **Length Chnl (Length channel at a bridge should be corrected manually; distance from the u/s section to the d/s section avoiding bridges and crossings)**
  9. ~~W.S. Elev~~

~~10. Vel Chnl~~
~~11. Min Ch El~~
~~12. E.G. Elev~~
~~13. Flow Area Ch~~
~~14. Frctn Slope~~
~~15. Hydr Depth~~
~~16. Shear Chan~~

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
Reach	River Sta	Description	Profile	W.S. Elev	Vel Chnl	Top W Chnl	Q Total	Q Channel	Top Width	Length Chnl	Min Ch El	E.G. Elev	Flow Area Ch	Frctn Slope	Hydr Depth	Shear Chan
				(m)	(m/s)	(m)	(m3/s)	(m3/s)	(m)	(m)	(m)	(m)	(m2)	(m/m)	(m)	(N/m2)
2211	10070		2yr_Fut	75.01	3.18	26.97	88	88	26.97	70	73.5	75.52	27.7		1.03	61.54
2211	10070		5yr_Fut	75.28	3.5	28.24	123.48	123.48	28.24	70	73.5	75.9	35.26		1.25	70.17
2211	10070		10yr_Fut	75.53	3.74	29.82	159.47	159.47	29.82	70	73.5	76.25	42.6		1.43	76.73
2211	10070		25yr_Fut	75.75	3.81	31.58	188.21	187.85	46.3	70	73.5	76.49	49.26		1.1	77.3
2211	10070		50yr_Fut	75.9	3.94	31.58	215.37	213.9	48.84	70	73.5	76.69	54.25		1.2	80.04
2211	10070		100yr_Fut	75.94	4.36	31.58	244.26	242.23	49.29	70	73.5	76.91	55.51		1.23	97.29
2211	10100		2yr_Fut	75.36	2.54	24.88	88	88	24.88	30.34	73.5	75.69	34.66	0.003912	1.39	35.73
2211	10100		5yr_Fut	75.64	2.95	27.2	123.48	123.48	27.2	30.34	73.5	76.09	41.88	0.00421	1.54	46.62
2211	10100		10yr_Fut	75.88	3.29	29.33	159.47	159.47	29.33	30.34	73.5	76.43	48.46	0.004434	1.65	56.72
2211	10100		25yr_Fut	75.89	3.85	29.47	188.21	188.21	29.47	30.34	73.5	76.65	48.9	0.005024	1.66	77.47
2211	10100		50yr_Fut	76.03	4.06	30.73	215.37	215.37	30.73	30.34	73.5	76.87	53.08	0.005001	1.73	84.98
2211	10100		100yr_Fut	76.19	4.21	32.2	244.26	244.26	32.2	30.34	73.5	77.09	58.08	0.005485	1.8	89.95
2211	10105	1-Waterfront Tr Bridge														
2211	10109		2yr_Fut	75.37	2.9	25.08	88	88	25.08	3.1	73.72	75.8	30.35	0.00361	1.21	48.61
2211	10109		5yr_Fut	75.7	3.19	26.67	123.48	123.48	26.67	3.1	73.72	76.21	38.72	0.003515	1.45	55.42
2211	10109		10yr_Fut	75.97	3.45	28.01	159.47	159.47	28.01	3.1	73.72	76.58	46.24	0.003494	1.65	62.18
2211	10109		25yr_Fut	76.21	3.55	29.17	188.21	188.21	29.17	3.1	73.72	76.85	52.98	0.003283	1.82	63.95
2211	10109		50yr_Fut	76.43	3.62	30.25	215.37	215.37	30.25	3.1	73.72	77.09	59.5	0.003078	1.97	64.69
2211	10109		100yr_Fut	76.62	3.73	31.21	244.26	244.26	31.21	3.1	73.72	77.33	65.5	0.002998	2.1	67.23

- **River Sta** must be numeric and unique. String and symbols will result in errors.

- **Profile** must contain unique numbers. Flood events are differentiated by numbers only. Event of 2 year is mandatory.

HecRas Flood Event Points Data (.shp):

- The shapefile must be projected.
- These columns must be provided
  1. **RIVER_STA**
- **RIVER_STA** must be numeric and unique. String and symbols will result in errors. The values must correspond to the values of **River Sta** column in HecRas Flood Event Data (.csv).

	FID	RIVER_STA
▶	0	10070
	1	10100
	2	10109
	3	10144
	4	10204
	5	10323
	6	10406
	7	10467
	8	10532

HecRas Bridge Data (.csv):

- These columns must be provided.
  1. **Reach**
  2. **River Sta**

3. **Profile**
4. **Q Total**
5. **Q Channel upstream**
6. **Q Bridge**
7. **Deck Width**
8. **Top W Chnl upstream**
9. **Total Span (d/s)** - for bridge damage (manually calculated)
10. **Total Span (u/s)** - for bridge damage (manually calculated)
11. **d/s station** - for mapping
12. **u/s station** - for mapping
13. **Distance from u/s station** – for mapping (distance from u/s station to the bridge)
14. **Scour Type –** 0/1 value

A	B	C	D	E	F	G	H	I	J	K	L	M
Reach	River Sta	Profile	Q Total	Q Bridge	Q Channel	Top W Chnl upstream	Deck Width	total span (d/s)	total span (u/s)	d/s station	u/s station	Distance from u/s station
			(m3/s)	(m3/s)	(m3/s)	(m)	(m)	incl. piers (m)	incl. piers (m)			(m)
2211	10105	2yr_Fut	70.1	70.1	70.1	24.2	3.4	39.2	39.2	10100	10109	10.39808316
2211	10105	5yr_Fut	103.5	103.5	103.5	25.79	3.4	39.2	39.2	10100	10109	10.39808316
2211	10105	10yr_Fut	136.2	136.2	136.2	27.13	3.4	39.2	39.2	10100	10109	10.39808316
2211	10105	25yr_Fut	161.3	161.3	161.3	28.08	3.4	39.2	39.2	10100	10109	10.39808316
2211	10105	50yr_Fut	185.3	185.3	185.3	29.05	3.4	39.2	39.2	10100	10109	10.39808316
2211	10105	100yr_Fut	210.8	210.8	210.8	30.07	3.4	39.2	39.2	10100	10109	10.39808316
2211	10549	2yr_Fut	70.1	70.1	70.1	24.49	22.44	24.08	24.88	10532	10567	35.20369452

- **River Sta** must be numeric and unique. String and symbols will result in errors.
- **Profile** must contain unique numbers. Flood events are differentiated by numbers only.

HecRas Bridge Data Stations (.shp):

- The shapefile must be projected.
- These columns must be provided.
  1. **RIVER_STA**

	FID	Shape	RIVER_STA
▶	0	Point	10105
	1	Point	10549
	2	Point	10976
	3	Point	13139
	4	Point	13594
	5	Point	13938
	6	Point	14074
	7	Point	14102

- **RIVER_STA** must be numeric and unique. String and symbols will result in errors. The values must correspond to the values of the **River Sta** column in HecRas Bridge Data (.csv).

HecRas Culvert Data (.csv):

- These columns must be provided
  1. **Reach**
  2. **River Sta**
  3. **ID** - *to identify multiple culverts*
  4. **Profile**

5. **Q Culv Group**
6. **Culv Span**
7. **Q Total**
8. **d/s station** - *for mapping*
9. **u/s station** - *for mapping*

A	B	C	D	E	F	G	H	I	J	K
Reach	River Sta	ID	Profile	Q Culv Group	Q Total	Culv Span	d/s station	u/s station	River Sta Culv	Description
				(m3/s)	(m3/s)	(m)				
Combined	9205	1	2-Year	6.17	6.17	11.4	9103	9211	92051	9205 Culvert #1
Combined	9205	1	5-Year	16.4	16.4	11.4	9103	9211	92051	9205 Culvert #1
Combined	9205	1	10-Year	27.4	27.4	11.4	9103	9211	92051	9205 Culvert #1
Combined	9205	1	50-Year	67.3	67.3	11.4	9103	9211	92051	9205 Culvert #1
Combined	9205	1	100-Year	88.88	92.3	11.4	9103	9211	92051	9205 Culvert #1
Combined	26600	1	2-Year	2.89	3.82	4.77	26574	26766	266001	26600 SouthCulvert
Combined	26600	2	2-Year	0.93	3.82	4.71	26574	26766	266002	26600 NorthCulvert

- **River Sta** must be numeric and unique. String and symbols will result in errors.
- **ID** is used to identify multiple culverts at one location. It must be a positive integer and unique. If there aren't multiple culverts, a single value must be used for all locations.

HecRas Culvert Data Stations (.shp):

- The shapefile must be projected.
- These columns must be provided
  1. **RIVER_STA**
  2. **ID**

	FID	Shape *	ID	RIVER_STA
▶	0	Point	1	13317
	1	Point	2	13317
	2	Point	3	13317
	3	Point	1	30324
	4	Point	1	30575
	5	Point	1	31446
	6	Point	1	31933
	7	Point	1	32356
	8	Point	1	21097

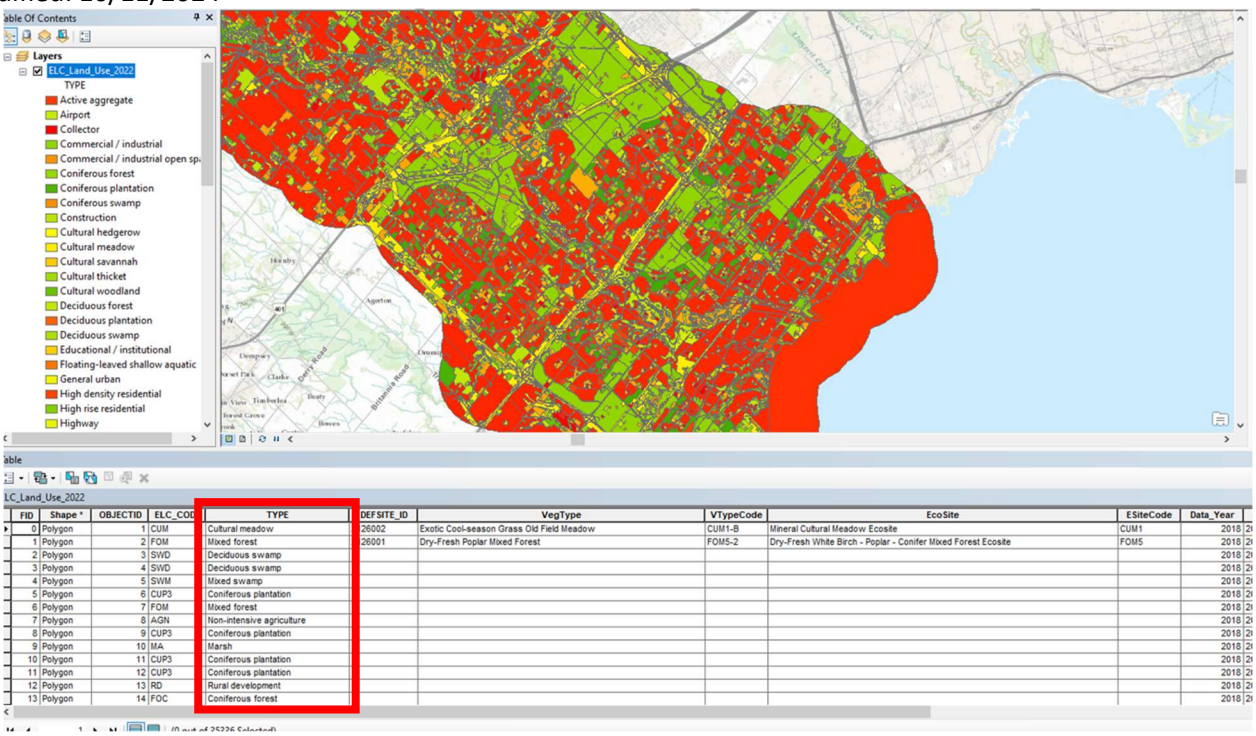
- **RIVER_STA** must be numeric and unique. String and symbols will result in errors. The values must correspond to the values of **River Sta** column in HecRas Culvert Data (.csv).

Land Use Stream Power Analysis Input Data Format

The Land Use Stream Power Scenario is present for situations where no modelled data available.

The input data that is required to run this analysis include:

- Digital Elevation Model (DEM)
- Land Use Shapefile (with a column labelled "TYPE" recording each land use type)



ELC_Land_Use_2022

FID	Shape *	OBJECTID	ELC_COD	TYPE	DEFSITE_ID	VegType	VTypeCode	EcoSite	ESiteCode	Data_Year
0	Polygon	1	CUM	Cultural meadow	26002	Exotic Cool-season Grass Old Field Meadow	CUM1-B	Mineral Cultural Meadow Ecosite	CUM1	2018
1	Polygon	2	FOM	Mixed forest	26001	Dry-Fresh Poplar Mixed Forest	FOM5-2	Dry-Fresh White Birch - Poplar - Conifer Mixed Forest Ecosite	FOM5	2018
2	Polygon	3	SWD	Deciduous swamp						2018
3	Polygon	4	SWD	Deciduous swamp						2018
4	Polygon	5	SWM	Mixed swamp						2018
5	Polygon	6	CUP3	Coniferous plantation						2018
6	Polygon	7	FOM	Mixed forest						2018
7	Polygon	8	AGN	Non-intensive agriculture						2018
8	Polygon	9	CUP3	Coniferous plantation						2018
9	Polygon	10	MA	Marsh						2018
10	Polygon	11	CUP3	Coniferous plantation						2018
11	Polygon	12	CUP3	Coniferous plantation						2018
12	Polygon	13	RD	Rural development						2018
13	Polygon	14	FOC	Coniferous forest						2018

- Excel Sheet assigning impervious values (%) based on the land use type. Please ensure that the Excel Sheet headings are outlined below. The categories in the "TYPE" column from the shapefile should be the same in the Excel Sheet.

TYPE	Imperviousness
Active aggregate	90
Airport	90
Collector	90
Commercial / industrial	78
Commercial / industrial open space	78
Coniferous forest	10
Coniferous plantation	10
Coniferous swamp	10
Construction	90
Cultural hedgerow	10
Cultural meadow	10
Cultural savannah	10
Cultural thicket	10

- Excel sheet containing coordinates within the watershed of interest (UTM Northing and Easting). These coordinates represent key watercourses that should be captured in the analysis. This helps the program determine the stream pruning threshold.
    - i.e. 611863.22, 4826408.52
- If pond footprints are available and the user would like to analyze the effect of ponds, please upload a shapefile with polygons of the pond to the input folder.

Multiple Watersheds

The same inputs as a Single Watershed scenario can be used aside from the following modification:

- A boundary shapefile of the area of interest is necessary.
- Multiple Modelled Stream Power
    - Inputs for multiple watersheds that fall within or intersect the boundary are necessary.
        - The inputs for each watershed are the same as outlined in the single watershed scenario.
    - Please separate information for each watershed using separate folders
- Multiple Land Use Stream Power
    - The inputs for each watershed are the same as outlined in the single watershed scenario.

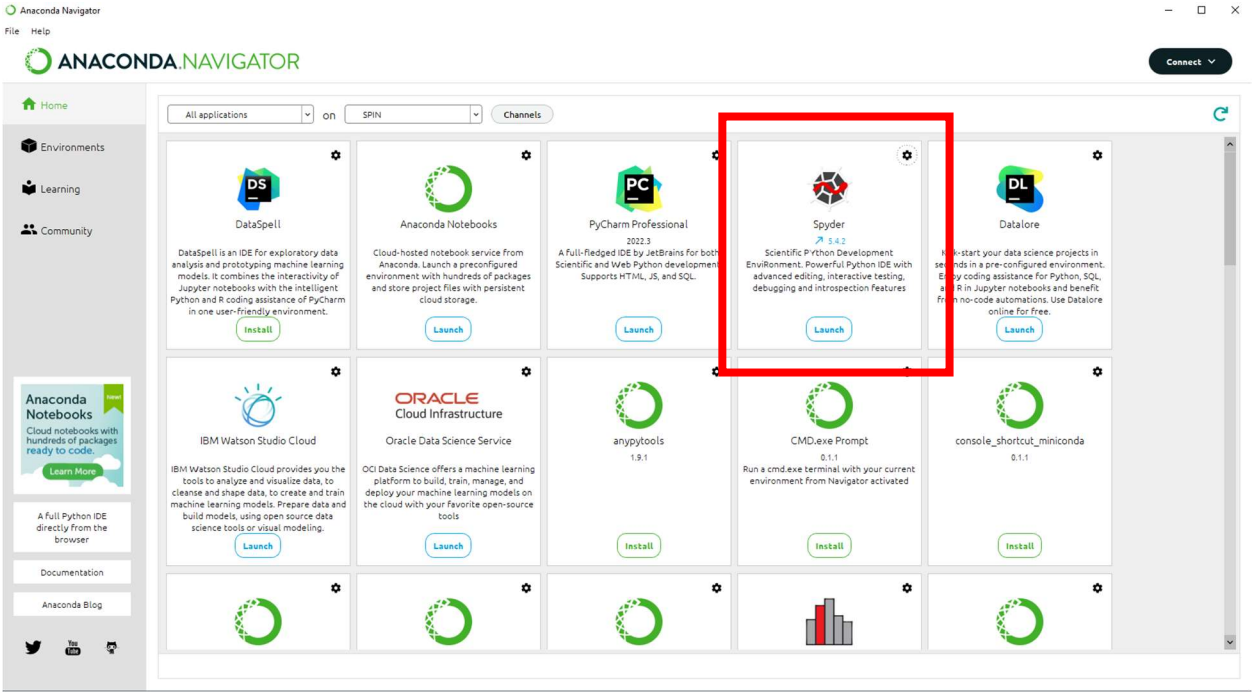
Running the Tool

Once the folder has been downloaded from GitHub and the inputs folder has been prepared as necessary, do the following:

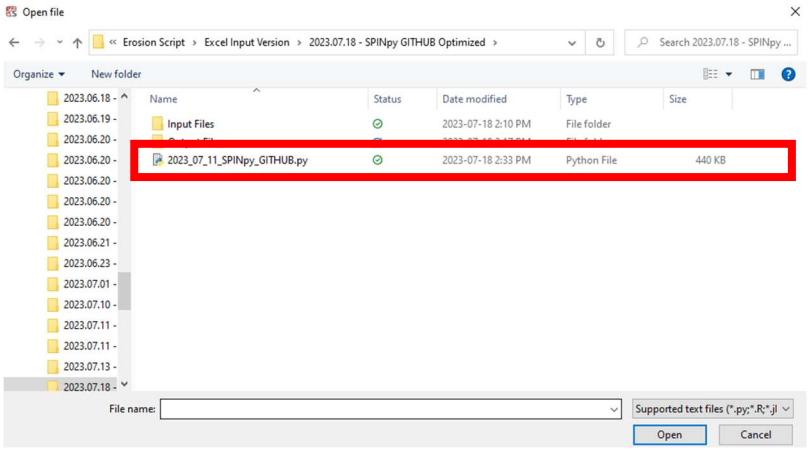
1. Open Anaconda and activate the SPIN environment which has been created previously. (Refer to Setting Up Python Environment)

   conda activate SPIN

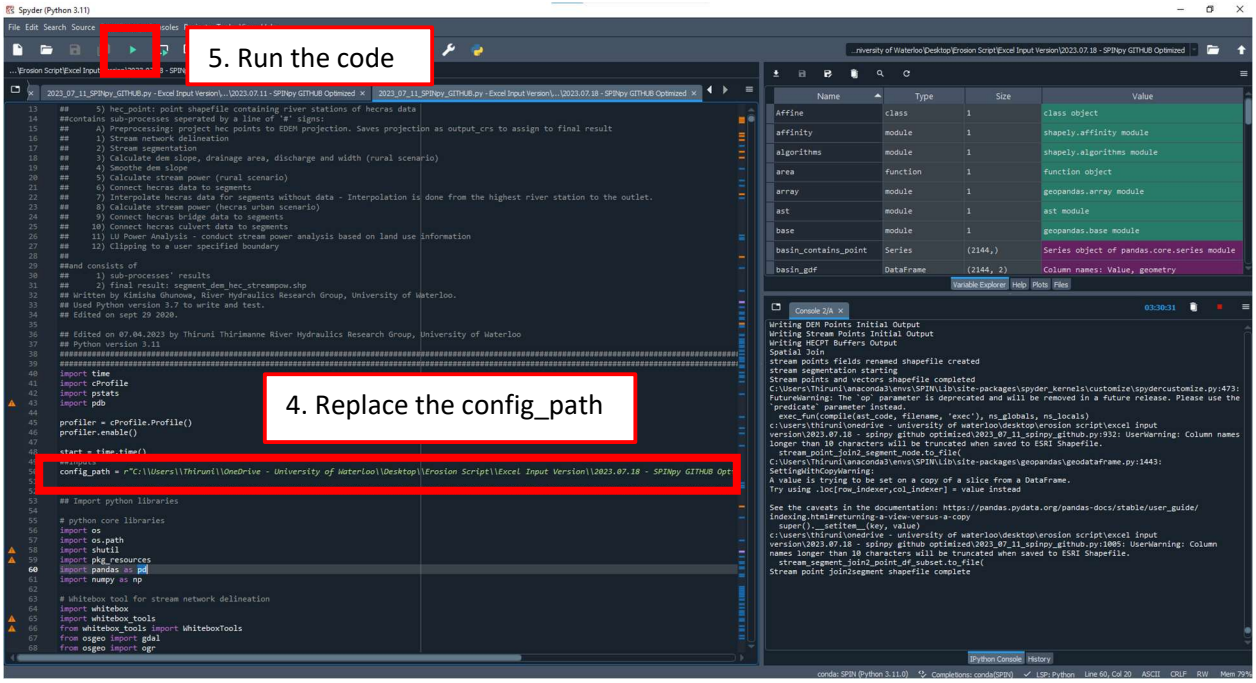
2. Launch the Spyder application.



3. File > Open:
   a. Navigate to where your folder structure is saved and select the Python file.



4. Then replace the config_path file path with your file path to the folder.
5. Press run and the code will execute in the console.

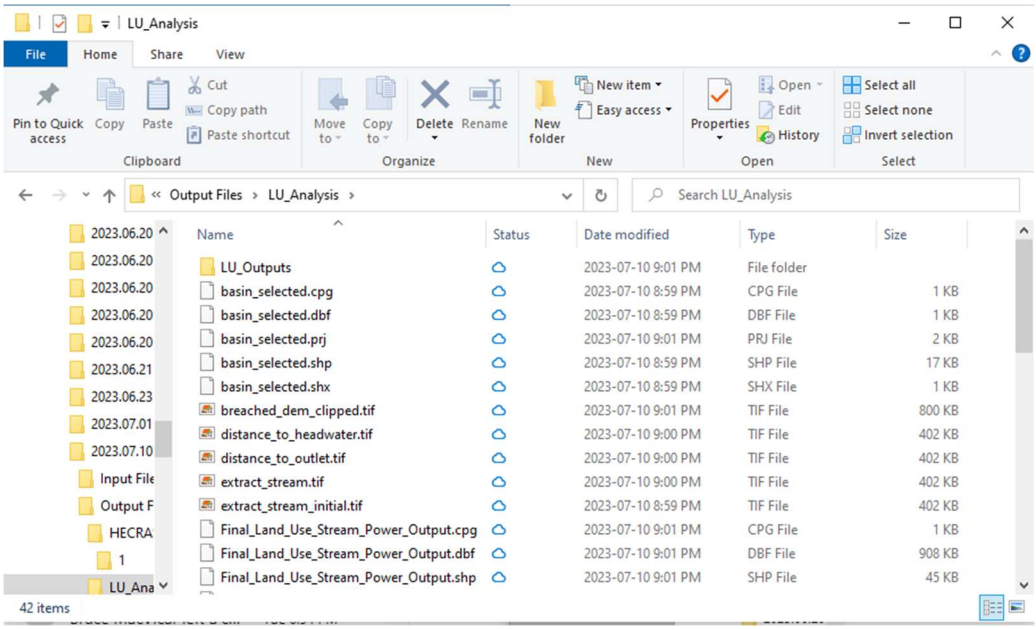


6. Remember that the amount of time it takes to execute depends on the level of detail of the DEM, the size of the file, and the processing speed of your system.
7. Once the code has been executed, the outputs folder will look similar to the figure below.

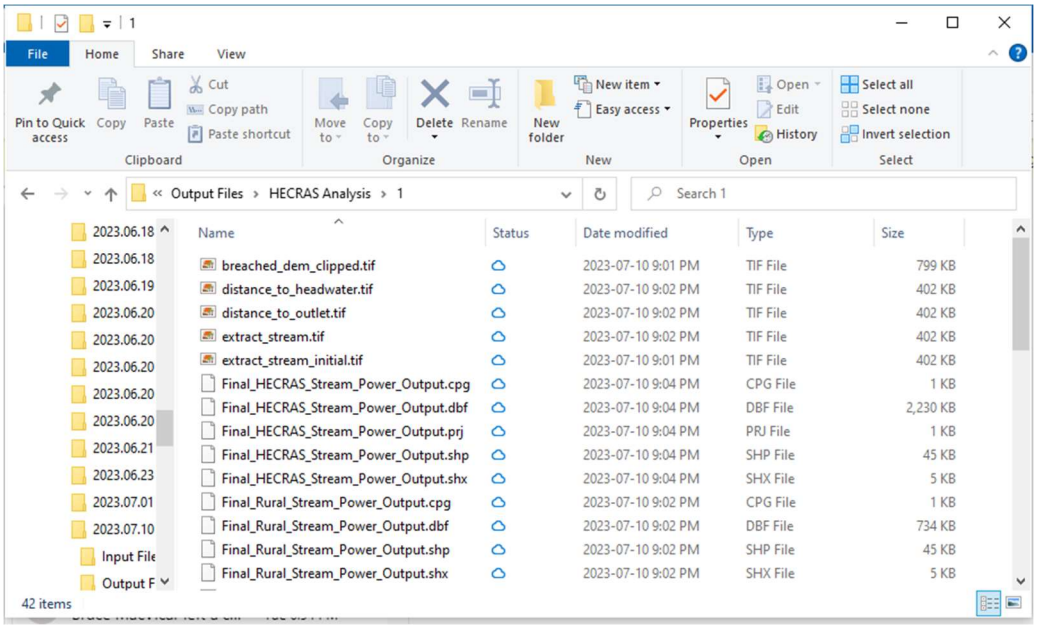
Main Output Folder:

Name	Status	Date modified	Type	Size
HECRAS Analysis		2023-07-10 9:01 PM	File folder	
LU_Analysis		2023-07-10 9:01 PM	File folder	
All Scenario Outputs Combined.cpg		2023-07-10 9:04 PM	CPG File	1 KB
All Scenario Outputs Combined.dbf		2023-07-10 9:04 PM	DBF File	2,634 KB
All Scenario Outputs Combined.prj		2023-07-10 9:04 PM	PRJ File	1 KB
All Scenario Outputs Combined.shp		2023-07-10 9:04 PM	SHP File	45 KB
All Scenario Outputs Combined.shx		2023-07-10 9:04 PM	SHX File	5 KB
basins.tif		2023-07-10 8:59 PM	TIF File	468 KB
breached_dem.tif		2023-07-10 8:59 PM	TIF File	932 KB
dem_filled.tif		2023-07-10 8:59 PM	TIF File	468 KB
flow_accum_extra.tif		2023-07-10 8:59 PM	TIF File	468 KB
flow_dir_dem.tif		2023-07-10 8:59 PM	TIF File	236 KB
flow_dir_extra.tif		2023-07-10 8:59 PM	TIF File	468 KB
profile_results.txt		2023-07-10 9:04 PM	Text Document	1,372 KB

LU Output Subfolder:

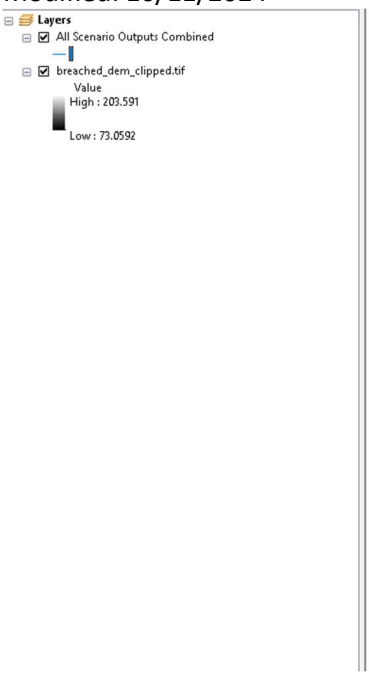


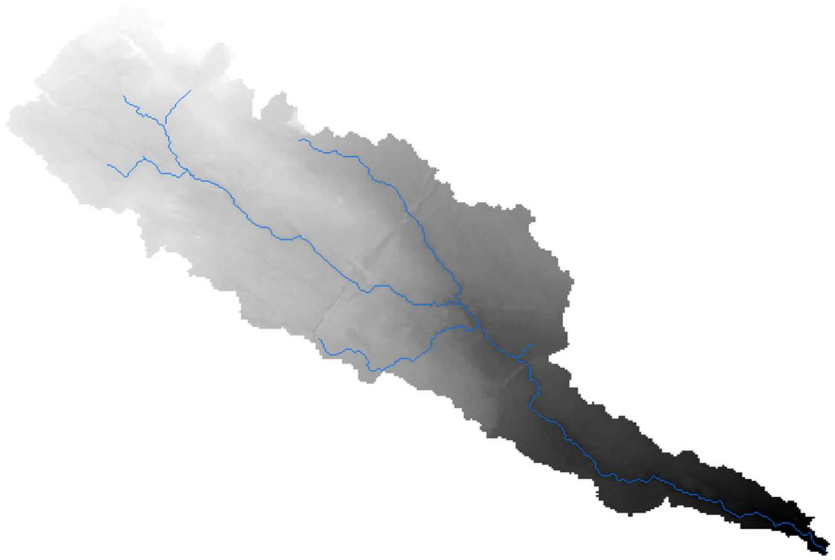
Modelled Output Subfolder:



Sample Output

One of the output files includes the breached dem pictured below. Another output is a shapefile called the "All Scenario Outputs Combined", which contains information such as stream power, discharge, and imperviousness at each segment.

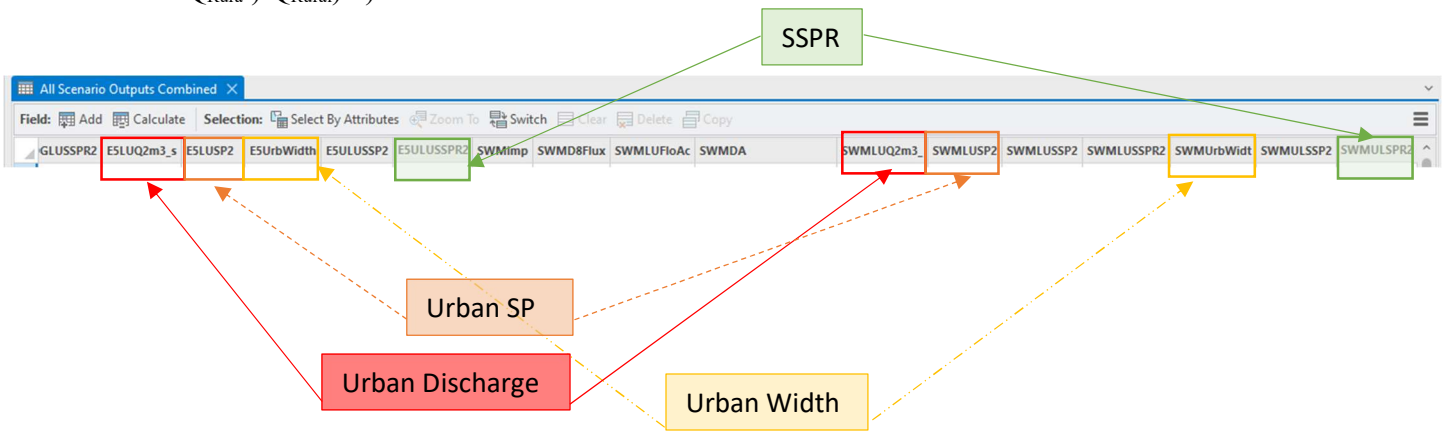




Key Columns to Consider (Attribute Table)

Notation:

- "OG" in front of it represents the original discharge equation $Q = 0.248(DA)^{0.91}(IA)^{0.30}$
- "E5" represents the NO SWM scenario calculations done using the new discharge equation. $Q = 0.248(DA(1\text{-}IA))^{0.91} + 11.08(DA(IA))^{0.60}$
- "SWM" represents the W SWM scenario using the new with swm discharge equation. $Q = 0.248(DA(1\text{-}IASWM))^{0.91} + 9.39(DA(IASWM))^{0.25}$
- The urban width equation for the land use scenario is $W_{urban} = W_{rural}\ (1+ 0.01((Q_{Urban} - Q_{Rural})/Q_{Rural})^{0.5})$



Appendix D
SPINpy 2 Code

```
##########################################################################################################################################
##########################################
##########################################################################################################################################
##########################################
## PURPOSE
## TO CALCULATE SPECIFIC STREAM POWER INDICES AS SPECIFIC STREAM POWER RATIOS (SSPR) TO ESTIMATE DAMAGE COSTS OF DIFFERENT TYPES OF INFRASTRUCTURE.
## TO RUN THIS SCRIPT, PYTHON LIBRARIES MUST BE INSTALLED. A PDF FILE CONTAINING INSTRUCTIONS IS ATTACHED.
## ------------------------------------------------------------------------------------------------------------------------------------
## INPUTS REQUIRED
## ------------------------------------------------------------------------------------------------------------------------------------
##     1) DATA_DIR: WORKING DIRECTORY TO STORE INTERMEDIATE AND FINAL RESULTS
##     2) DEM: ENFORCED DIGITAL ELEVATION MODEL (MUST HAVE SAME PROJECTION AS HEC_POINT INPUT) ASSUMPTION: 10M

##     URBAN MODELLED SCENARIO
##     3) HEC_CSV: HECRAS .CSV DATA (MUST FOLLOW FORMAT AS FORMAT (INCLUDING TEXTS) IN EXAMPLE FILE)
##     4) BRIDGE_HEC_CSV: BRIDGE .CSV DATA (MUST FOLLOW FORMAT AS FORMAT (INCLUDING TEXTS) IN EXAMPLE FILE)
##     5) HEC_POINT: POINT SHAPEFILE CONTAINING RIVER STATIONS OF HECRAS DATA

##     URBAN LAND USE SCENARIO
##     6) LAND USE RASTER OR SHAPEFILE THAT CONTAINS THE IMPERVIOUS DATA (PROVIDE ONE)
##         6.1) IF PROVIDING A SHAPEFILE THE COLUMN 'TYPE' MUST BE INCLUDED AND IT IS ASSUMED IMPERVIOUSNESS IS NOT A COLUMN IN THE SHAPEFILE
##             INSTEAD A .CSV FILE NEEDS TO BE PROVIDED WITH A TABLE FOR EACH UNIQUE TYPE REPRESENTED IN THE SHAPEFILE AND ITS ASSOCIATED IMPERVIOUS VALUE
##     7) SAMPLE POINTS .CSV FILE THAT CONTAINS POINTS ALONG THE STREAMS OF INTEREST AND THEIR APPROXIMATE HEADWATERS (USED FOR STREAM PRUNING THRESHOLD)

##     WITH STORMWATER MANAGEMENT SCENARIO
##     8) SHAPEFILE CONTAINING THE BOUNDARY OF THE PONDS

##     OPTIONAL
##     9) BOUNDARY FOR WHICH THE ANALYSIS IS TO BE CONDUCTED FOR
## ------------------------------------------------------------------------------------------------------------------------------------
## POSSIBLE SCNEARIOS IN THE PROGRAM
## ------------------------------------------------------------------------------------------------------------------------------------
##     0) POLITICAL BOUNDARY OR SINGLE WATERSHED
##     1) RURAL SCENARIO
##         A) PREPROCESSING: PROJECT HEC POINTS TO EDEM PROJECTION. SAVES PROJECTION AS OUTPUT_CRS TO ASSIGN TO FINAL RESULT
##         B) STREAM NETWORK DELINEATION
##         C) STREAM SEGMENTATION
##         D) CALCULATE DEM SLOPE, DRAINAGE AREA, DISCHARGE AND WIDTH (RURAL SCENARIO)
##         E) SMOOTHE DEM SLOPE
##         F) CALCULATE STREAM POWER (RURAL SCENARIO)

##     2) URBAN SCENARI0
##         A) PREPROCESSING: PROJECT HEC POINTS TO EDEM PROJECTION. SAVES PROJECTION AS OUTPUT_CRS TO ASSIGN TO FINAL RESULT
##         B) STREAM NETWORK DELINEATION
##         C) STREAM SEGMENTATION
##         D) CALCULATE DEM SLOPE, DRAINAGE AREA, DISCHARGE AND WIDTH (RURAL SCENARIO)
##         E) SMOOTHE DEM SLOPE
##         F) CALCULATE STREAM POWER (RURAL SCENARIO)

##         2.1) LAND USE DATA ANALYSIS
##             A) PREPROCESS IMPERVIOUS RASTER AND CONNECT IMPERVIOUS DATA TO SEGMENTS
##             B) INTERPOLATE DATA FOR SEGMENTS WITHOUT DATA - INTERPOLATION IS DONE FROM THE HIGHEST RIVER STATION TO THE OUTLET.
##             C) CALCULATE DISCHARGE, WIDTH AND STREAM POWER USING THE NEW EQUATIONS (LAND USE URBAN SCENARIO)

##             2.1.1) NO STORMWATER MANAGEMENT ANALYSIS
##             2.1.2) WITH STORMWATER MANAGEMENT ANALYSIS

##         2.2) MODELLED DATA ANALYSIS
##             A) CONNECT HECRAS DATA TO SEGMENTS
##             B) INTERPOLATE HECRAS DATA FOR SEGMENTS WITHOUT DATA - INTERPOLATION IS DONE FROM THE HIGHEST RIVER STATION TO THE OUTLET.
##             C) CALCULATE STREAM POWER (HECRAS URBAN SCENARIO)
##             D) CONNECT HECRAS BRIDGE DATA TO SEGMENTS
##             E) CONNECT HECRAS CULVERT DATA TO SEGMENTS

##             2.2.1) NO STORMWATER MANAGEMENT ANALYSIS
##             2.2.2) WITH STORMWATER MANAGEMENT ANALYSIS
## ------------------------------------------------------------------------------------------------------------------------------------
## OUTPUTS
## ------------------------------------------------------------------------------------------------------------------------------------
##     1) INDIVIDUAL SUB-PROCESSES' RESULTS WITH SPECIFIC FOLDERS
##     2) MERGED RESULT: ALL SCENARIO OUTPUTS.SHP

## SPINPY 1 AND SPINPY 2 EDITED ON 09.25.2024 BY THIRUNI THIRIMANNE RIVER HYDRAULICS RESEARCH GROUP, UNIVERSITY OF WATERLOO
## PYTHON VERSION 3.11
##########################################################################################################################################
##########################################
##########################################################################################################################################
##########################################
import gc
from memory_profiler import profile
import time
import cProfile
import pstats

profiler = cProfile.Profile()
profiler.enable()

start = time.time()
##INPUTS
config_path = r"C:\\Users\\Thiruni\\OneDrive - University of Waterloo\\Desktop\\Erosion Script\\Excel Input Version\\2024.10.20 Masters Outputs\\User_Inputs.xls"

## IMPORT PYTHON LIBRARIES
import os
import os.path
import shutil
import pkg_resources
import pandas as pd
import numpy as np
import whitebox
import whitebox_tools
from whitebox_tools import WhiteboxTools
from osgeo import gdal
from osgeo import ogr
from osgeo import gdal, osr
import sys
import shapefile
import geopandas
from geopandas import *
import geopandas as gpd
import shapefile
import shapely
from shapely import *
from shapely.geometry import (
    box,
    LineString,
    MultiLineString,
    MultiPoint,
    Point,
    Polygon,
    mapping,
)  # ASSHAPE)
from shapely.ops import split, snap, nearest_points
import fiona
from fiona import crs
from fiona.crs import from_epsg, from_string
from fiona import collection
```

```
import osgeo.osr as osr
import struct
import pyproj
from pyproj import Proj, transform
import rasterio
from rasterio.features import shapes
import pyepsg
from pyproj import _datadir, datadir
from matplotlib import pyplot as plt
from sklearn import linear_model
import xlrd
import math
import csv
from rasterio.features import geometry_mask
from rasterio.features import rasterize
from rasterio.windows import from_bounds
from rasterio.warp import reproject, Resampling
from rasterio.mask import mask
from rasterio.merge import merge
from rasterio.plot import show
from rasterio.windows import Window
import shapely
from shapely.geometry import shape, mapping, MultiPolygon, Point, Polygon
from shapely.geometry import box
from functools import partial
import shapely.wkt
import json
from geocube.api.core import make_geocube
from geocube.rasterize import rasterize_image
from functools import partial
from rasterio.enums import MergeAlg
from shapely.ops import nearest_points
import shapely.geometry as geom
from shapely.ops import unary_union, cascaded_union
import itertools
import rioxarray as riox
from prettytable import PrettyTable
import matplotlib.pyplot as plt
import rasterio.plot
import ast
import glob
from affine import Affine
import scipy
from scipy.spatial import cKDTree
from scipy.stats import linregress
import tempfile
from rasterio.transform import from_origin
from rasterio.mask import mask as rio_mask
import xarray as xr
wbt = whitebox.WhiteboxTools()  # EXTRACTING INFO FROM DEMS
#################################################################################################################################
#############################################
#################################################################################################################################
#############################################
## READ INPUT DATA
loc = config_path

# TO OPEN WORKBOOK
wb = xlrd.open_workbook(loc)
sheet = wb.sheet_by_index(0)

# CELL BY ROW X AND COLUMN Y . FIRST ROW,COLUMN = (0,0)
# REQUIRED
# STREAM POWER
param_1 = sheet.cell_value(1, 3)  # WORKING DIRECTORY
param_2 = sheet.cell_value(2, 3)  # ENFORCED DEM
param_3 = sheet.cell_value(3, 3)  # CELL RESOLUTION DEM
param_4 = sheet.cell_value(
    4, 3
)  # WATERSHED FILES (CONTAINS HECRAS, BRIDGE, CULVERT FILES)

param_5 = sheet.cell_value(5, 3)  # Y/N HECRAS ANALYSIS

# BRIDGE
param_6 = sheet.cell_value(6, 3)  # BRIDGE ANALYSIS (Y/N)

# CULVERT
param_7 = sheet.cell_value(7, 3)  # CULVERT ANALYSIS (Y/N)

# SUBWATERSHED POLYGON
param_8 = sheet.cell_value(8, 3)  # SUBWATERSHED ANALYSIS (Y/N)
param_9 = sheet.cell_value(9, 3)  # SUBWATERSHED POLYGON

# PRE-DEFINED
param_10 = sheet.cell_value(12, 3)  # INITIAL DRAINAGE THRESHOLD FOR STREAM EXTRACTION
param_11 = sheet.cell_value(13, 3)  # RURAL Q COEFF
param_12 = sheet.cell_value(14, 3)  # RURAL Q EXP
param_13 = sheet.cell_value(15, 3)  # RURAL W COEFF
param_14 = sheet.cell_value(16, 3)  # RURAL W EXP
param_15 = sheet.cell_value(17, 3)  # SLOPE SMOOTHING DISTANCE
param_16 = sheet.cell_value(18, 3)  # SLOPE SMOOTHING THRESHOLD FOR DRAINAGE DIFFERENCE RATIO
param_23 = sheet.cell_value(19,3) # USER SPECIFIED DRAINAGE AREA FOR PRUNING

# LAND USE
param_17 = sheet.cell_value(21, 3)  # Y/N LU ANALYSIS
param_30 = sheet.cell_value(22, 3) #Y/N LAND USE RASTER
param_31 = sheet.cell_value(23, 3) #LU RASTER PATH
param_32 = sheet.cell_value(24, 3) #LU RASTER CELL SIZE

param_33 = sheet.cell_value(25, 3)  # LU SHAPEFILE Y/N
param_18 = sheet.cell_value(26, 3)  # LU SHAPEFILE
param_19 = sheet.cell_value(27, 3)  # OBTAINING THE CSV FILE THAT HAS IMPERVIOUSNESS VALUES
param_20 = sheet.cell_value(28, 3)  # BOUNDARY TO CLIP TO

# SINGLE WATERSHED XY COORDINATE (FOR LU) - WILL BE USING THIS TO EXTRACT A BASIN
param_21 = sheet.cell_value(29, 3)  # SAMPLE POINTS ON THE STREAM (CSV)
param_22 = sheet.cell_value(30, 3)  # URBAN DISCHARGE EQUATION COEFFICIENT C
param_26 = sheet.cell_value(31,3) # URBAN DISCHARGE EQUATION COEFFICIENT D
param_29 = sheet.cell_value(32,3) #URBAN WIDTH COEFFICIENT C

#ENGINEERED SOLUTIONS
param_24 = sheet.cell_value(33, 3)# ASK THE USER IF THEY HAVE SWM SOLUTIONS TO CONSIDER
param_25 = sheet.cell_value(34, 3) #OBTAIN THE POND BOUNDARIES
param_27 = sheet.cell_value(35, 3) #WITH SWM URBAN DISCHARGE EQUATION COEFF C
param_28 = sheet.cell_value(36, 3) #WITH SWM URBAN DISCHARGE EQUATION COEFF D
## ------------------------------------------------------------------------------------------------------------------------------
# RSI NEW LU CODE
# ASK FOR A CSV FILE - ONLY OPTION - OR IMPERVIOUS RASTER FILE
# ASSUME NO IMPERVIOUS FIELD IN SHAPEFILE - USER PROVIDES CSV (TYPE, %IMP)
# RSI TO CREATE UTILITY TOOL TO EXTRACT LAND USE TYPES INTO CSV FOR USER INPUT
## ------------------------------------------------------------------------------------------------------------------------------
# READ THE CSV TABLE
if param_17 == "Y":
    if param_30 == "Y":
        print("There is a LU Raster")

    if param_33 == "Y":
        print("There is a LU Shapefile")
        with open(param_19, newline="") as csvfile:
            reader = csv.reader(csvfile)
            data = [row for row in reader]

        # CREATE A PRETTY TABLE
        table = PrettyTable()
        table.field_names = data[0]
        for row in data[1:]:
            table.add_row(row)
        print(table)

        # CREATE THE VALUE MAPPING DICTIONARY FROM CSV DATA
```

```
        value_mapping = {}
        for row in data[1:]:
            value_mapping[row[0]] = float(row[1])
        print(value_mapping)

# ASSIGN INPUT DATA

data_dir = str(param_1)
dem = str(param_2)
cell_x = int(param_3)

# RSI NEW LU CODE
## INSERT BOOLEAN IF THERE IS A HECRAS FILE
hec_analysis = str(param_5)

hec_watershed = str(param_4)  # ALL HECRAS WATERSHED FILES

bridge_analysis = str(param_6)

culvert_analysis = str(param_7)

subwatershed_analysis = str(
    param_8
)  # MUST BE SPECIFIED BY USER, IF YES, TYPE "Y", OR ELSE TYPE "N": N
subwshed_raw = str(
    param_9
)  # FOR SUBWATERSHED ANALYSIS, A SHAPEFILE MUST BE PROVIDED AND THE FIELD SUBWATERSHED_ANALYSIS MUST
BE "Y" BELOW.

# FLOW ACCUMULATION CELL AREA
cell_size = cell_x**2  # AREA OF DEM CELL

# INITIAL STREAM NETWORK EXTRACTION THRESHOLD BEFORE PRUNING
initial_dathres_param = int(param_10)
print("Initial_dathres_param = " + str(initial_dathres_param))
initial_dathres = round(
    initial_dathres_param / cell_size
)  # STARTING WITH 10 CELLS FOR 10MX10M DEM  : 1000 M2/100M2 = 10 CELLS

# EMPIRICAL EQUATIONS
rQ_a = float(param_11)  # RURAL DISCHARGE COEFFICIENT : 0.248
rQ_b = float(param_12)  # RURAL DISCHARGE EXPONENT : 0.91
rW_a = float(param_13)  # RURAL WIDTH COEFFICIENT :  1.16
rW_b = float(param_14)  # RURAL WIDTH EXPONENT : 0.508
uQ_c = float(param_22) #URBAN DISCHARGE IMPERVIOUSNESS COEFFICIENT: C NO SWM
uQ_d = float(param_26) #URBAN DISCHARGE IMPERVIOUSNESS COEFFICIENT: D NO SWM

u_Qs_c = float(param_27) #URBAN DISCHARGE IMPERVIOUSNESS COEFFICIENT: D W SWM
u_Qs_d = float(param_28) #URBAN DISCHARGE IMPERVIOUSNESS COEFFICIENT: D W SWM
urb_width = float(param_29) #URBAN WIDTH COEFFICIENT C

# SLOPE SMOOTHING
Savg_cell_window_param = int(param_15)
Savg_cell_window = (
    int(Savg_cell_window_param / cell_x) + 1
)  # WINDOW FOR 500 M SMOOTHING SLOPE (25 CELLS UPSTREAM AND 25 CELLS DOWNSTREAM): 500
thres_da = float(
    param_16
)  # DRAINAGE THRESHOLD RATIO DIFFERENCE FOR SMOOTHING SLOPE : 0.1
try:
    pruningda = int(param_23)
except ValueError:  # HANDLE THE EXCEPTION IF THE CELL IS EMPTY OR NOT CONVERTIBLE TO AN INTEGER
    param_23 = None
    pruningda = None
print("User Specified Pruning Value:", pruningda)

engswmanalysis = param_24
engineeredsswm = param_25
print(engswmanalysis)
print(engineeredsswm)

# RSI NEW LU CODE
# LU PARAMETERS
lu_analysis = str(param_17)
lu_file = str(param_18)
boundary = str(param_20)
lu_streampts = str(param_21)
lurasteranalysis = str(param_30)
lurasteranalysispath = str(param_31)
lushapefileanalysis = str(param_33)
## RSI TO ENSURE ALL OUTPUT SHAPEFILES HAVE A .PRJ FILE
##############################################################################################################################################
##############################################
##############################################################################################################################################
##############################################
# DEFINE THE STUDY AREA WATERSHED
# GETTING THE BASIN
## 1) STREAM NETWORK DELINEATION

# SET WORKING DIRECTORY
wbt.set_working_dir(data_dir)
wbt.verbose = False

# BREACH ALL OF THE DEPRESSIONS IN A DEM.
wbt.breach_depressions(dem, "breached_dem.tif")

# GET FLOW DIRECTION (D8 METHOD)
wbt.d8_pointer("breached_dem.tif", "flow_dir_dem.tif")

# FULL WORKFLOW
wbt.flow_accumulation_full_workflow(
    dem=dem,
    out_dem="dem_filled.tif",
    out_pntr="flow_dir_extra.tif",
    out_accum="flow_accum_extra.tif",
)

# GET BASIN/ SUBBASIN
wbt.basins("flow_dir_dem.tif", "basins.tif")
## ----------------------------------------------------------------------------------------------------------------------------------------
## VARIABLES FOR HOW MANY LOOPS
## ----------------------------------------------------------------------------------------------------------------------------------------
check_boundary = len(boundary)

if check_boundary == 0:
    counter = 0
    print("SINGLE WATERSHED")
elif check_boundary > 0:
    counter = 2
    print("MULTIPLE WATERSHEDS")

# OPEN THE DEM RASTER FILE TO GET THE EPSG CODE
with rasterio.open(dem) as src:
    # GET THE CRS OF THE RASTER FILE
    crs = src.crs

    # USE PYPROJ TO DETECT THE EPSG CODE OF THE CRS
    dem_espg2int = pyproj.CRS.from_string(crs.to_string()).to_epsg()

    print(f"EPSG code: {dem_espg2int}")

# DEFINE A DEFAULT CRS
default_crs = "EPSG:" + str(dem_espg2int)
##############################################################################################################################################
##############################################
##############################################################################################################################################
##############################################
## SINGLE WATERSHED AND LAND USE SELECTED
start1 = time.time()
```

```
if check_boundary == 0 and lu_analysis == "Y":
    counter == 0

if counter == 0 and lu_analysis == "Y":
    print("LU STARTING FOR SINGLE WATERSHEDS")

    with rasterio.open(dem) as src:
        crs = src.crs

    # THIS IS THE CRS THAT EVERYTHING SHOULD BE
    proj_crs = crs

    # CREATE A FOLDER FOR LAND USE WATERSHED, SAVE EVERYTHING HERE
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CLIP DEM TO BASIN
## ------------------------------------------------------------------------------------------------------------------------------------------------------

    watershed_output = os.path.join(data_dir, "LU_Analysis")
    if os.path.exists(watershed_output):
        print("FOLDER CREATED")
    else:
        intermediate_dir = os.makedirs(watershed_output)

    wbt.set_working_dir(watershed_output)
    wbt.verbose = False

    if not os.path.exists(watershed_output):
        os.makedirs(watershed_output)

    lu_analysis_file = os.path.join(watershed_output, "LU_Analysis.gpkg")

    # CHECK IF THE GEOPACKAGE ALREADY EXISTS
    if os.path.exists(lu_analysis_file):
        print("GeoPackage already exists")
    else:
        print("GeoPackage created:", lu_analysis_file)

    # CREATE A SHAPEFILE OUT OF THE USER PROVIDED SAMPLE POINTS THIS VALUE IS USED IN THE EXTRACTION OF THE
BASIN AND THE THRESHOLD VALUE DETERMINATION LATER ON
    lupoints = pd.read_csv(lu_streampts)

    geometry = [Point(xy) for xy in zip(lupoints['X'], lupoints['Y'])]
    lupoints['geometry'] = geometry
    lugdf = gpd.GeoDataFrame(lupoints, geometry='geometry')
    lugdf.crs = crs
    lusamplepoints = os.path.join(watershed_output, "LU_Userinput_Points.shp")
    lugdf.to_file(lusamplepoints)

    print("created a shapefile for the user provided location points")

    # CONVERT BASIN TO POLYGON
    basin_raster_path = os.path.join(data_dir, "basins.tif")
    basin_polygon_path = "basins_polygon"

    mask = None
    with rasterio.Env():
        with rasterio.open(basin_raster_path) as src:
            image = src.read(1)
            results = (
                {"properties": {"Value": v}, "geometry": s}
                for i, (s, v) in enumerate(
                    shapes(image, mask=mask, transform=src.transform)
                )
            )
        geoms = list(results)
        polygonized_basin_df = gpd.GeoDataFrame.from_features(geoms)
        polygonized_basin_df.to_file(lu_analysis_file, layer=basin_polygon_path, driver="GPKG")

    print("Basin polygon saved to GeoPackage:", basin_polygon_path)

    # READS THE SHAPEFILE CONTAINING ALL OF THE POINTS THAT THE USER HAS PROVIDED
    lu_points_gdf = gpd.read_file(lusamplepoints)

    # CREATE A DICTIONARY TO STORE THE COUNT OF POINTS WITHIN EACH BASIN
    basin_counts = {}

    # THE SHAPEFILE WHERE THE USER INPUTS VALUES ARE LIKELY MULTIPLE POINTS
    for index, row in lu_points_gdf.iterrows():
        # STEP 2: EXTRACT LONG AND LAT COORDINATES FROM THE SHAPEFILE
        long = row['X']  # REPLACE WITH THE ACTUAL COLUMN NAME
        lat = row['Y']    # REPLACE WITH THE ACTUAL COLUMN NAME

        # STEP 3: CREATE A GEODATAFRAME CONTAINING THESE POINTS WITH THE SPECIFIED CRS
        location_point = Point(long, lat)
        proj_crs = crs#REPLACE WITH YOUR CRS
        basin_loc_gdf = gpd.GeoDataFrame(geometry=[location_point], crs=proj_crs)

        # STEP 4: WRITE THE POINT GEODATAFRAME TO A GEOPACKAGE FILE AS "LOCATION_POINT" LAYER
        basin_loc_gdf.to_file(lu_analysis_file, layer=f"Location_Point_{index}", driver="GPKG")
        print(f"Location point {index} saved to GeoPackage")

        # STEP 5: READ IN THE BASIN SHAPEFILE FROM THE GEOPACKAGE
        basin_gdf = gpd.read_file(lu_analysis_file, layer='basins_polygon')

        # STEP 6: CHECK WHICH BASIN THE POINT LIE WITHIN USING THE CONTAINS METHOD
        basin_contains_point = basin_gdf.contains(location_point)

        # STEP 7: UPDATE THE BASIN COUNTS BASED ON WHICH BASIN(S) THE POINT(S) LIE WITHIN
        for basin_index, contains in enumerate(basin_contains_point):
            if contains:
                if basin_index not in basin_counts:
                    basin_counts[basin_index] = 1
                else:
                    basin_counts[basin_index] += 1

    # STEP 8: SELECT THE BASIN WITH THE HIGHEST COUNT (MAJORITY OF POINTS)
    if basin_counts:
        selected_basin_index = max(basin_counts, key=basin_counts.get)
        selected_basin_gdf = basin_gdf.iloc[selected_basin_index:selected_basin_index+1]

        # STEP 9: WRITE THE SELECTED BASIN TO THE GEOPACKAGE AS "BASIN_SELECTED" LAYER
        selected_basin_gdf.to_file(lu_analysis_file, layer=f'basin_selected', driver='GPKG')
        print(f"Selected Basin {selected_basin_index} with the majority of points")

        basin_polygon_sub_path = os.path.join(watershed_output, "basin_selected.shp")
        # WRITE THE GEODATAFRAME TO THE SHAPEFILE
        selected_basin_gdf.to_file(basin_polygon_sub_path, driver='ESRI Shapefile')

        print("Created Basin Selected Shapefile")

    # STEP 10: DEFINE PATHS FOR THE DEM FILES (SAME AS IN THE PREVIOUS RESPONSE)
    overallbreached_dem = os.path.join(data_dir, "breached_dem.tif")
    dem_clipped_path = os.path.join(watershed_output, "breached_dem_clipped.tif")

    with rasterio.open(overallbreached_dem) as src:
        # READ THE RASTER DATA
        raster_data = src.read()

        # OPEN THE SHAPEFILE OR POLYGON FILE
        with fiona.open(basin_polygon_sub_path, "r") as shapefile:
            # EXTRACT THE GEOMETRY OF THE POLYGON
            geometry = [feature["geometry"] for feature in shapefile]

            # PERFORM THE CLIPPING OPERATION
            clipped_data, clipped_transform = rasterio.mask.mask(src, geometry, crop=True)

            # UPDATE THE METADATA
            clipped_meta = src.meta
            clipped_meta.update({"transform": clipped_transform, "height": clipped_data.shape[1], "width": clipped_data.shape[2], "count":
clipped_data.shape[0]})
```

```
        # WRITE THE CLIPPED DATA TO A NEW RASTER FILE
        with rasterio.open(dem_clipped_path, "w", **clipped_meta) as dst:
            dst.write(clipped_data)

    # CHECK IF THE OUTPUT FILE WAS CREATED
    if os.path.isfile(dem_clipped_path):
        print("Saved the Breached DEM Clipped")
    else:
        print("Failed to clip the raster to the polygon.")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## GENERAL WORKFLOW GET FLOW DIRECTION (D8 METHOD)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    gen_flow_dir = os.path.join(watershed_output, "flow_dir.tif")
    wbt.d8_pointer(dem_clipped_path, gen_flow_dir)

    # GET FLOW ACCUMULATION
    gen_flow_accum = os.path.join(watershed_output, "flow_accum.tif")
    wbt.d8_flow_accumulation(dem_clipped_path, gen_flow_accum, out_type="cells")

    # EXTRACT STREAMS BASED ON DRAINAGE THRESHOLD
    print("Pruning Threshold:",pruningda)

    if pruningda == None:
        gen_extr_strm = os.path.join(watershed_output, "extract_stream_initial.tif")
        wbt.extract_streams(gen_flow_accum, gen_extr_strm, initial_dathres)
    else:
        gen_extr_strm = os.path.join(watershed_output, "extract_stream_usprune.tif")
        wbt.extract_streams(gen_flow_accum, gen_extr_strm, (pruningda/cell_size))

    # GET STREAM VECTOR FROM RASTER STREAM
    gen_strm_vec = os.path.join(watershed_output, "stream_vector_initial.shp")
    wbt.raster_streams_to_vector(gen_extr_strm, gen_flow_dir, gen_strm_vec)

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_vector_initial'
    gdf = gpd.read_file(gen_strm_vec)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    initial_dem_points_path = os.path.join(watershed_output, "dem_points_initial.shp")
    wbt.raster_to_vector_points(gen_extr_strm, initial_dem_points_path)
    # SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
    initial_dem_points = geopandas.read_file(initial_dem_points_path)

    # WRITE TO A GEOPACKAGE
    print("Writing DEM Points Initial Output")
    layer_name = 'dem_points_initial'
    initial_dem_points.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    initial_stream_point_path = os.path.join(
        watershed_output, "stream_points_initial.shp"
    )
    initial_dem_points[initial_dem_points["VALUE"] == 1].to_file(
        initial_stream_point_path, driver="ESRI Shapefile"
    )
    # WRITE TO A GEOPACKAGE
    print("Writing Stream Points Initial Output")
    layer_name = 'stream_points_initial'
    gdf = gpd.read_file(initial_stream_point_path)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## STREAM PRUNING: REDEFINE STREAM THRES BASED ON MIN FLOW ACCUM OF HEC POINTS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # EXTRACT FLOW ACCUMULATION TO STREAM POINTS
    wbt.extract_raster_values_at_points("flow_accum.tif", initial_stream_point_path)

    strm_point_faccum_gdf = geopandas.read_file(initial_stream_point_path)
    before_last = len(strm_point_faccum_gdf.columns) - 2
    name_map = {strm_point_faccum_gdf.columns[before_last]: "Faccum"}
    strm_point_faccum_gdf.rename(columns=name_map, inplace=True)

    # OPEN LU SAMPLE POINTS TO CREATE BUFFER AROUND IT
    lu_pts_buffer_gdf = geopandas.read_file(lusamplepoints)
    lu_pts_buffer_gdf["geometry"] = lu_pts_buffer_gdf.geometry.buffer(50)

    lu_pts_buffer_gdf = lu_pts_buffer_gdf.rename(columns={"FID": "RIVER_STA"})

    buffers_lu_path = os.path.join(watershed_output, "lupt_buffers50m.shp")
    lu_pts_buffer_gdf.to_file(buffers_lu_path, driver="ESRI Shapefile")
    print("Writing LU Buffers Output")

    # WRITE TO A GEOPACKAGE
    layer_name = 'lupt_buffers50m'
    gdf = gpd.read_file(buffers_lu_path)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    print("Spatial Join")

    chunk_size = 500  # ADJUST THE CHUNK SIZE AS PER YOUR MEMORY LIMITATIONS

    # DIVIDE THE GEODATAFRAMES INTO SMALLER SUBSETS AND PERFORM THE SPATIAL JOIN
    point_in_buffer = gpd.GeoDataFrame()

    for i, chunk_left in enumerate(lu_pts_buffer_gdf.groupby(np.arange(len(lu_pts_buffer_gdf)) // chunk_size)):
        chunk_right = strm_point_faccum_gdf  # ASSUMING RIGHT_DF IS THE OTHER GEODATAFRAME FOR THE JOIN

        # PERFORM SPATIAL JOIN ON THE CURRENT SUBSET
        chunk_join = gpd.sjoin(chunk_left[1], chunk_right, how='inner')

        # CONCATENATE THE CURRENT JOIN RESULT TO THE FINAL RESULT
        point_in_buffer = gpd.GeoDataFrame(pd.concat([point_in_buffer, chunk_join], ignore_index=True))

    # DISPLAY THE COLUMN NAMES
    column_names = point_in_buffer.columns.tolist()
    print("Column names:", column_names)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## BRANCHING
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # GROUP BY "RIVER_STA" AND CALCULATE THE MAXIMUM "FACCUM" AT EACH RIVER STATION
    point_in_buffer_max = point_in_buffer.groupby("RIVER_STA")["Faccum"].apply(lambda x: x.max())

    if pruningda == None:
        #PLOT FOR ALL OF THE POINTS NOT SEPERATED BY REACH ID
        point_in_buffer = point_in_buffer.dropna()
        point_in_buffer = point_in_buffer[point_in_buffer['Faccum'] != 0]

        # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA
        x_coordinates = point_in_buffer['RIVER_STA']
        flow_accumulation = point_in_buffer['Faccum']

        # PLOT THE X-COORDINATE VS. FLOW ACCUMULATION
        plt.scatter(x_coordinates, flow_accumulation)

        # PERFORM LINEAR REGRESSION
        slope, intercept, r_value, p_value, std_err = linregress(x_coordinates, flow_accumulation)
        # ADD A REGRESSION LINE TO THE PLOT
        plt.plot(x_coordinates, intercept + slope * x_coordinates, color='red', label='Regression Line')

        print(intercept + slope * x_coordinates)
        print("Intercept:", intercept)
        print(x_coordinates, flow_accumulation)

        # CUSTOMIZE THE PLOT
        plt.xlabel('River Station')
        plt.ylabel('Flow Accumulation')
        #PLT.TEXT(0.1, 0.9, F'INTERCEPT: {INTERCEPT}', TRANSFORM=PLT.GCA().TRANSAXES, FONTSIZE=7)
        plt.title(f'X-coordinate vs. Flow Accumulation for all Points')
        plt.legend()

        # SAVE THE PLOT AS AN IMAGE FILE
```

```
plot_filename = os.path.join(watershed_output,"LUAll_Reaches.tif")
plt.savefig(plot_filename, format="tif")

# SHOW OR SAVE THE PLOT
plt.show()

# CREATE AN EMPTY DATAFRAME TO STORE THE MAXIMUM FLOW ACCUMULATIONS
min_flow_table = pd.DataFrame(columns=['REACH', 'Min_Flow_Accumulation'])

# PLOTS FOR EACH INDIVIDUAL REACH ID
# EXTRACT UNIQUE REACH NUMBERS FROM YOUR DATA
unique_reach_numbers = point_in_buffer['REACH'].unique()

# PRINT THE UNIQUE REACH NUMBERS FOR REFERENCE
print("Unique REACH Numbers:", unique_reach_numbers)

# LOOP THROUGH EACH UNIQUE REACH NUMBER
for reach_number in unique_reach_numbers:
    # FILTER DATA BASED ON THE CURRENT REACH NUMBER
    filtered_data = point_in_buffer[point_in_buffer['REACH'] == reach_number]

    # FILTER OUT DATA POINTS WITH ZERO FLOW ACCUMULATION
    filtered_data = filtered_data[filtered_data['Faccum'] != 0]

    # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING
    if not filtered_data.empty:

        # CALCULATE THE MAXIMUM FLOW ACCUMULATION WITHIN THE FILTERED DATA FOR EACH RIVER_STA
        max_flow_per_station = filtered_data.groupby("RIVER_STA")["Faccum"].max().reset_index()

    # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING WITHIN EACH RIVER_STA GROUP
        if not max_flow_per_station.empty:
            # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA FROM MAX_FLOW_PER_STATION
            x_coordinates = max_flow_per_station['RIVER_STA'].values.reshape(-1, 1)
            flow_accumulation = max_flow_per_station['Faccum']

            # CREATE A NEW PLOT FOR EACH REACH NUMBER
            plt.figure()
            plt.scatter(x_coordinates, flow_accumulation)

            # CREATE A LINEAR REGRESSION MODEL
            regr = linear_model.LinearRegression()
            regr.fit(x_coordinates, flow_accumulation)

            # MAKE PREDICTIONS USING THE X FOR LINEAR LINE
            Y_pred = regr.predict(x_coordinates)

            # ASSIGN THE Y_PRED VALUES TO A NEW COLUMN 'Y_PRED' IN MAX_FLOW_PER_STATION
            max_flow_per_station['Y_pred'] = Y_pred

            # REPLACE NEGATIVE VALUES IN COLUMN 'Y_PRED' WITH VALUES FROM COLUMN 'FACCUM'
            mask = max_flow_per_station['Y_pred'] < 0
            max_flow_per_station.loc[mask, 'Y_pred'] = max_flow_per_station.loc[mask, 'Faccum']

            # PLOT FLOW ACCUMULATION AGAINST RIVER STATION
            plt.scatter(x_coordinates, flow_accumulation, color="black")
            # ADD A REGRESSION LINE TO THE PLOT
            plt.plot(x_coordinates, Y_pred, color='red', label='Regression Line')

            # CUSTOMIZE THE PLOT
            plt.xlabel('River Station')
            plt.ylabel('Flow Accumulation')
            plt.title(f'X-coordinate vs. Flow Accumulation for REACH {reach_number}')
            plt.legend()

            # SAVE THE PLOT AS AN IMAGE FILE
            plot_filename = os.path.join(watershed_output, f"lureach_{reach_number}_plot.tif")
            plt.savefig(plot_filename, format="tif")

            # SHOW THE CURRENT PLOT
            plt.show()

        #MINIMUM FLOW ACCUMULATION OF THE REACH
        print(f'{reach_number}',max_flow_per_station)
        min_flow_accumulation = max_flow_per_station['Y_pred'].min()
        print(f'{reach_number}', min_flow_accumulation)

        # APPEND THE RESULT TO THE MAX_FLOW_TABLE DATAFRAME USING CONCAT
        min_flow_table = pd.concat([min_flow_table, pd.DataFrame({'REACH': [reach_number],'Min_Flow_Accumulation':
[min_flow_accumulation]})], ignore_index=True)

    # FIND THE MINIMUM FACCUM_MIN_YPRED
    min_faccum_min_ypred = min_flow_table["Min_Flow_Accumulation"].min()

    # CALCULATE FACCUM_THRES_25PCNT
    if min_faccum_min_ypred < 0:
        faccum_thres_25pcnt = initial_dathres_param
    else:
        faccum_thres_25pcnt = int(min_faccum_min_ypred)

    # PRINT THE RESULTS
    print("Result Table:")
    print(min_flow_table)
    path = os.path.join(watershed_output,"min_flow_table.csv")
    min_flow_table.to_csv(path, index=False)
    print("Minimum faccum_min_ypred:", min_faccum_min_ypred)
    print("faccum_thres_25pcnt:", faccum_thres_25pcnt)

else:
    faccum_thres_25pcnt = round(pruningda/cell_size)
    print(faccum_thres_25pcnt)

# EXTRACT STREAM
wbt.extract_streams("flow_accum.tif", "extract_stream.tif", faccum_thres_25pcnt)
print("Streams have been extracted with a threshold applied")

# GET POINTS FROM RASTER STREAM BASED ON VALUE = 1
wbt.raster_to_vector_points("extract_stream.tif", "dem_points.shp")

# SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
dem_points_path = os.path.join(watershed_output,"dem_points.shp")
dem_points = geopandas.read_file(dem_points_path)

# WRITE TO A GEOPACKAGE
layer_name = 'dem_points'
gdf = gpd.read_file(gen_strm_vec)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

new_stream_point_path = os.path.join(watershed_output,"stream_points.shp")
dem_points[dem_points["VALUE"] == 1].to_file(
    new_stream_point_path, driver="ESRI Shapefile"
)
# WRITE TO A GEOPACKAGE
layer_name = 'stream_points'
gdf = gpd.read_file(new_stream_point_path)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

# GET STREAM VECTOR FROM RASTER STREAM
wbt.raster_streams_to_vector(
    "extract_stream.tif", "flow_dir.tif", "stream_vector_pruned.shp"
)

# CLIP STREAM VECTOR TO SUBWATERSHED ANALYSIS
if subwatershed_analysis == "Y":
    subwshed_df = geopandas.read_file(subwshed_poly)

    stream_vector_pruned_path = os.path.join(watershed_output,"stream_vector_pruned.shp")
    stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
    subwshed_stream_df = geopandas.sjoin(
        stream_vector_pruned_df, subwshed_df, op="intersects"
```

```
        )
        subwshed_stream_list = subwshed_stream_df["FID"].tolist()
        # SUBSET BY LIST OF LINES ONLY WITHIN SUBWATERSHED
        subwshed_stream_selected_df = stream_vector_pruned_df[
            stream_vector_pruned_df["FID"].isin(subwshed_stream_list)
        ]
        stream_vector_path = os.path.join(watershed_output,"stream_vector.shp")
        subwshed_stream_selected_df.to_file(stream_vector_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_vector'
        gdf = gpd.read_file(gen_strm_vec)
        gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    else:
        stream_vector_pruned_path = os.path.join(watershed_output,"stream_vector_pruned.shp")
        stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
        stream_vector_path = os.path.join(watershed_output,"stream_vector.shp")
        stream_vector_pruned_df.to_file(stream_vector_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_vector_pruned'
        gdf = gpd.read_file(stream_vector_pruned_path)
        gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_output, 'stream_vector_pruned.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_vector'
        gdf = gpd.read_file(stream_vector_path)
        gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## EXTRACT ALL RASTER VALUES TO STREAM POINTS
## GET STREAM LINK CLASS
## CLASSIFICATION KEY
## VALUE  CLASS
## 1      EXTERIOR LINK
## 2      INTERIOR LINK
## 3      SOURCE NODE (HEADWATER)
## 4      LINK NODE (CONFLUENCE)
## 5      SINK NODE (OUTLET)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    wbt.stream_link_class("flow_dir.tif", "extract_stream.tif", "stream_link_class.tif")

    # GET UNIQUE ID FOR EACH STREAM LINK CLASS
    wbt.stream_link_identifier(
        "flow_dir.tif", "extract_stream.tif", "stream_link_id.tif"
    )

    # GET LENGTH FOR EACH STREAM LINK CLASS
    wbt.stream_link_length(
        "flow_dir.tif", "stream_link_id.tif", "stream_link_length.tif"
    )

    # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
    wbt.tributary_identifier("flow_dir.tif", "extract_stream.tif", "tributary_id.tif")

    # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
    wbt.strahler_stream_order(
        "flow_dir.tif", "extract_stream.tif", "strahler_order.tif"
    )

    # GET DISTANCE TO OUTLET
    wbt.distance_to_outlet(
        "flow_dir.tif", "extract_stream.tif", "distance_to_outlet.tif"
    )

    # GET THE DISTANCE TO THE FURTHEST UPSTREAM CHANNEL HEAD FOR EACH STREAM CELL.
    wbt.farthest_channel_head(
        "flow_dir.tif", "extract_stream.tif", "distance_to_headwater.tif"
    )

    # GET STREAM SLOPE ALONG RASTER STREAM
    # WBT.STREAM_SLOPE_CONTINUOUS("FLOW_DIR.TIF", "STREAM_LINK_ID.TIF", DEM , "STREAM_SLOPE.TIF")

    # EXTRACT ALL RASTER VALUES TO SELECTED STREAM POINTS
    wbt.extract_raster_values_at_points(

"distance_to_headwater.tif;distance_to_outlet.tif;stream_link_class.tif;stream_link_id.tif;stream_link_length.tif;tributary_id.tif;strahler_order.tif;f
low_accum.tif",
        new_stream_point_path,
    )

    # EXTRACT RASTER DEM VALUES TO STREAM POINTS
    src_filename = dem
    shp_filename = new_stream_point_path

    src_ds = gdal.Open(src_filename)
    gt = src_ds.GetGeoTransform()
    rb = src_ds.GetRasterBand(1)

    ds = ogr.Open(shp_filename, 1)
    lyr = ds.GetLayer()
    lyr.CreateField(ogr.FieldDefn("Elev", ogr.OFTReal))
    for feat in lyr:
        geom = feat.GetGeometryRef()
        mx, my = geom.GetX(), geom.GetY()  # COORD IN MAP UNITS

        # CONVERT FROM MAP TO PIXEL COORDINATES.
        # ONLY WORKS FOR GEOTRANSFORMS WITH NO ROTATION.
        px = int((mx - gt[0]) / gt[1])  # X PIXEL
        py = int((my - gt[3]) / gt[5])  # Y PIXEL

        intval = rb.ReadAsArray(px, py, 1, 1)
        elev = intval[0]
        for value in elev:
            v = float(value)
            # PRINT(V)

        # UPDATE NEW FIELD
        lyr.SetFeature(feat)
        feat.SetField("Elev", v)
        lyr.SetFeature(feat)

    # RENAME FIELDS
    stream_point = geopandas.read_file(new_stream_point_path)

    name_map = {
        stream_point.columns[2]: "DistUp",
        stream_point.columns[3]: "DistDown",
        stream_point.columns[4]: "LinkClass",
        stream_point.columns[5]: "LinkID",
        stream_point.columns[6]: "LinkLength",
        stream_point.columns[7]: "TribID",
        stream_point.columns[8]: "Strahler",
        stream_point.columns[9]: "FlowAccum",
        stream_point.columns[10]: "Elev",
    }
    stream_point.rename(columns=name_map, inplace=True)

    # GET XY COORDINATES
    stream_point["X"] = stream_point.geometry.x
    stream_point["Y"] = stream_point.geometry.y
    stream_point["pXY"] = stream_point["X"].map(str) + "," + stream_point["Y"].map(str)
```

```
    # SAVE AS SHAPEFILE
    stream_point_field_rename_path = os.path.join(watershed_output,"stream_points_fields_renamed.shp")
    stream_point.to_file(stream_point_field_rename_path, driver="ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_points_fields_renamed'
    gdf = gpd.read_file(stream_point_field_rename_path)
    stream_point.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')
    print("stream points fields renamed shapefile created ")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## STREAM SEGMENTATION
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    print("stream segmentation starting")
    stream_point = os.path.join(watershed_output,"stream_points.shp")
    stream_vector = os.path.join(watershed_output,"stream_vector.shp")

    # CREATE FUNCTION TO SPLIT LINE AT POINTS
    def split_line_at_points(line, point):
        gdf_line = line
        gdf_points = point

        # JOIN ALL LINE AND POINTS
        line = gdf_line.geometry.unary_union
        coords = gdf_points.geometry.unary_union

        # SNAP AND SPLIT COORDS ON LINE
        # RETURNS GEOMETRYCOLLECTION
        split_line = shapely.ops.split(line, coords)

        # CHECK IF SPLIT_LINE IS A GEOMETRYCOLLECTION
        if isinstance(split_line, shapely.geometry.collection.GeometryCollection):
            # ITERATE OVER INDIVIDUAL GEOMETRIES IN THE COLLECTION
            segments = [feature for feature in split_line.geoms]
        else:
            segments = [split_line]

        # TRANSFORM GEOMETRY COLLECTION TO GEODATAFRAME
        gdf_segments = geopandas.GeoDataFrame(
            list(range(len(segments))), geometry=segments
        )
        gdf_segments.columns = ["index", "geometry"]

        # SAVE SEGMENTS
        stream_segments = os.path.join(watershed_output,"stream_segments.shp")
        gdf_segments.to_file(stream_segments, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments'
        gdf = gpd.read_file(stream_segments)
        gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

        # RETURN GDF_SEGMENTS

    split_line_at_points(gpd.read_file(stream_vector), gpd.read_file(stream_point))
    # 3 MINS PROCESSING TIME

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_vector'
    gdf = gpd.read_file(stream_vector)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_vector.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_points'
    gdf = gpd.read_file(stream_point)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## GET NODES OF SEGMENTS AND JOIN POINT ATTRIBUTES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    print("getting nodes of segments and joining point attributes")
    # GET LINE NODES (START AND END) USING WHITEBOX TO MATCH POINT NODES
    stream_segments = os.path.join(watershed_output,"stream_segments.shp")
    stream_segment_nodes = os.path.join(watershed_output,"stream_segment_nodes.shp")
    wbt.extract_nodes(stream_segments, stream_segment_nodes)

    # CREATE JOIN IDENTIFIER - FIELD "INDEX" (PLUS 1) OF STREAM SEGMENT IS FIELD "PARENTID" OF STREAM SEGMENT
NODE.
    segment_attribute = ogr.Open(stream_segments, 1)
    segment_lyr = segment_attribute.GetLayer()
    segment_lyr.CreateField(ogr.FieldDefn("JoinID", ogr.OFTInteger))

    for feat in segment_lyr:
        # GET INDEX VALUE
        feat_v = feat.GetField("index")
        new_v = feat_v + 1
        # UPDATE NEW FIELD
        segment_lyr.SetFeature(feat)
        feat.SetField("JoinID", new_v)
        segment_lyr.SetFeature(feat)

    # JOIN STREAM POINTS DATA TO SEGMENTS' NODES
    stream_point_field_rename_df = geopandas.read_file(stream_point_field_rename_path)
    stream_segment_node_df = geopandas.read_file(stream_segment_nodes)

    stream_point_join2_segment_node = geopandas.sjoin(
        stream_segment_node_df,
        stream_point_field_rename_df,
        how="inner",
        op="intersects",
    )
    stream_segment_join2_point_path = os.path.join(watershed_output,"stream_point_join2_segment_node.shp"
    )
    stream_point_join2_segment_node.to_file(
        stream_segment_join2_point_path, driver="ESRI Shapefile"
    )

    # USE DISTANCE FROM OUTLET TO FIND THE COORDINATES OF START NODES AND END NODES OF SEGMENTS. SAVE
THE END COORDINATES IN THE SAME ROW AS THE START COORDINATES.
    stream_segment_join2_point_df = geopandas.read_file(stream_segment_join2_point_path)

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_point_join2_segment_node'
    stream_segment_join2_point_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_point_join2_segment_node.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segment_nodes'
    stream_segment_node_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_segment_nodes.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)

    stream_segment_join2_point_df["X"] = stream_segment_join2_point_df.geometry.x
    stream_segment_join2_point_df["Y"] = stream_segment_join2_point_df.geometry.y
    stream_segment_join2_point_df["StartXY"] = (
        stream_segment_join2_point_df["X"].map(str)
```

```
    + ","
    + stream_segment_join2_point_df["Y"].map(str)
)

# FIND MIN DISTANCE FROM OUTLET (I.E. LOWER NODE) AND UPDATE NEW COLUMN WITH ENDXY COORDINATES.
stream_segment_join2_point_df[
    "DistDown_min"
] = stream_segment_join2_point_df.groupby(["PARENT_ID"])["DistDown"].transform(min)

# SUBSET DATA BY EXTRACTING START NODES
stream_segment_join2_point_df_startnodes = stream_segment_join2_point_df[
    stream_segment_join2_point_df["DistDown_min"]
    != stream_segment_join2_point_df["DistDown"]
]
stream_segment_join2_point_df_endnodes = stream_segment_join2_point_df[
    stream_segment_join2_point_df["DistDown_min"]
    == stream_segment_join2_point_df["DistDown"]
]
stream_segment_join2_point_df_endnodes.loc[
    :, "EndXY"
] = stream_segment_join2_point_df_endnodes["StartXY"]

stream_segment_join2_point_df_subset = pd.merge(
    stream_segment_join2_point_df_startnodes,
    stream_segment_join2_point_df_endnodes[["PARENT_ID", "EndXY"]],
    how="left",
    on="PARENT_ID",
)

# MATCH THE ELEV VALUE FROM THE STREAM POINTS FOR BOTH NODES OF STREAM SEGMENTS
stream_segment_join2_point_df_subset[
    "EndElev"
] = stream_segment_join2_point_df_subset["EndXY"].map(
    stream_point_field_rename_df.set_index("pXY")["Elev"]
)
stream_segment_join2_point_df_subset[
    "StartElev"
] = stream_segment_join2_point_df_subset["StartXY"].map(
    stream_point_field_rename_df.set_index("pXY")["Elev"]
)

stream_segment_join2_point_df_subset_path = os.path.join(watershed_output,"stream_point_join2_segment_node_subset.shp")
stream_segment_join2_point_df_subset.to_file(
    stream_segment_join2_point_df_subset_path, driver="ESRI Shapefile"
)
print("stream segment join2 point complete")
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## JOIN STREAM SEGMENT NODES TO STREAM SEGMENT BASED ON PARENT ID
## ---------------------------------------------------------------------------------------------------------------------------------------------------
stream_segments_df = geopandas.read_file(stream_segments)
stream_segment_join2_point_df_subset_df = geopandas.read_file(
    stream_segment_join2_point_df_subset_path
)

# MUST DROP GEOMETRY OF POINT TO ALLOW FILE TO BE SAVED IN THE FUTURE
stream_segment_join2_point_df_subset_df = (
    stream_segment_join2_point_df_subset_df.drop("geometry", axis=1)
)

# STREAM_SEGMENTS_ATTRIBUTE_DF =
STREAM_SEGMENTS_DF.JOIN(STREAM_SEGMENT_JOIN2_POINT_DF_SUBSET.SET_INDEX('PARENT_ID'), ON='JOINID',
RSUFFIX = "_P")
stream_segments_attribute_df = stream_segments_df.merge(
    stream_segment_join2_point_df_subset_df,
    left_on="JoinID",
    right_on="PARENT_ID",
    how="left",
)

# WRITE TO A GEOPACKAGE
layer_name = 'stream_point_join2_segment_node_subset'
stream_segment_join2_point_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

# DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
file_pattern = os.path.join(watershed_output, 'stream_point_join2_segment_node_subset.*')
file_list = glob.glob(file_pattern)
for file_path in file_list:
    os.remove(file_path)
# WRITE TO A GEOPACKAGE
layer_name = 'stream_segments'
gdf = gpd.read_file(stream_segments)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE DEM SLOPE, DRAINAGE AREA, DISCHARGE, AND WIDTH (RURAL SCENARIO)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
print("calculating dem slope , drainage area, discharge, and width")
# CALCULATE LENGTH OF SEGMENTS
stream_segments_attribute_df["Length"] = stream_segments_attribute_df.length

# CALCULATE SLOPE USING DEM ELEVATION
stream_segments_attribute_df["Slope"] = (
    (
        stream_segments_attribute_df["EndElev"]
        - stream_segments_attribute_df["StartElev"]
    )
    / (stream_segments_attribute_df["Length"])
) * -1

# CALCULATE DRAINAGE AREA (DA = (FLOW ACCUMULATION/1000000)*100
stream_segments_attribute_df["DA_km2"] = (
    stream_segments_attribute_df["FlowAccum"] / 1000000
) * cell_size

# CALCULATE RURAL DISCHARGE USING Q=CA^X WHERE C AND X ARE COEFFICCIENTS AND A IS THE DRAINAGE AREA
stream_segments_attribute_df["Q_m3pers"] = rQ_a * (
    stream_segments_attribute_df["DA_km2"] ** rQ_b
)

# CALCULATE WIDTH (EMPIRICAL - DA WIDTH)
stream_segments_attribute_df["RurWidth"] = rW_a * (
    stream_segments_attribute_df["DA_km2"] ** rW_b
)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## SMOOTH DEM SLOPE
## ---------------------------------------------------------------------------------------------------------------------------------------------------
print("Smoothing dem slope")

# SET INDEX FROM HIGH ELEVATION TO LOW, I.E. BY DECREASING DISTANCE DOWNSTREAM
stream_segments_attribute_df = stream_segments_attribute_df.sort_values(
    by=["DistDown"], ascending=False
)
stream_segments_attribute_df["SID"] = np.arange(len(stream_segments_attribute_df))
stream_segments_attribute_df.set_index("SID", inplace=True)
stream_segments_attribute_df["index"] = np.arange(len(stream_segments_attribute_df))

# ADD NEW FIELDS TO STORE RESULTS
stream_segments_attribute_df["deltaDA"] = None
stream_segments_attribute_df["SavgID"] = None
stream_segments_attribute_df["SavgRID"] = None
stream_segments_attribute_df["SlopeAVGLU"] = None

# CREATE SEGMENTS BLOCKS OVER WHICH TO SMOOTH SLOPE
# CREATE LIST WITH ALL ELEVATION VALUES AND SORT IN DESCENDING ORDER. REMOVE ROWS WITHOUT
ATTRIBUTES
stream_segments_attribute_df = stream_segments_attribute_df.dropna(
    axis=0, subset=["PARENT_ID"]
)

elev_set = set(stream_segments_attribute_df.apply(lambda row: (row["index"], row["EndElev"]), axis=1))
```

```
    elev_list_u = list(elev_set)
    elev_list = sorted(elev_list_u, key=lambda x: x[0])  # ACTIVATED

    # START FROM THE SEGMENT WITH HIGHEST ELEVATION, ASSIGN THE SAME SEGMENT ID WHILE TRACING
DOWNSTREAM OF THE REACH UNTIL THE DIFFERENCE RATIO IN DRAINAGE AREA IS >=0.1.
    seg_id = 0
    elev_list_count = elev_list

    while len(elev_list_count) != 0:
        elev_pair = elev_list_count[0]
        max_elev = elev_pair[1]
        max_id = elev_pair[0]
        seg_id = seg_id + 1
        del elev_list_count[0]

        # MAX_ELEV AND MAX_ID ONLY HAS ONE POSSIBLE MATCH, SKIP THE LOOP AND USE EXISTING INDEXES
        seg_r_id = 0
        start_endxy = stream_segments_attribute_df.loc[max_id, "EndXY"]
        start_da = stream_segments_attribute_df.loc[max_id, "DA_km2"]
        stream_segments_attribute_df.loc[max_id, "SavgID"] = seg_id
        stream_segments_attribute_df.loc[max_id, "SavgRID"] = seg_r_id

        if start_da is None or start_da == 0:
            start_da = 0.0

        # RATHER THAN LOOPING THROUGH MULTIPLE TIMES, QUERY THE DATAFRAME FOR THE MAIN CONDITION
        ssaIdxs = stream_segments_attribute_df.query(
            "@start_endxy == StartXY"
        ).index.tolist()
        # NOW LOOP THROUGH THE INITIAL MATCHES AND DO MATH
        while len(ssaIdxs) != 0:
            index = ssaIdxs[0]
            del ssaIdxs[0]
            r_da = stream_segments_attribute_df.loc[index, "DA_km2"]
            r_seg_id = stream_segments_attribute_df.loc[index, "SavgID"]
            if r_da == 0 or r_da == None:
                delta_da = 0.0
            else:
                diff_da = r_da - start_da
                delta_da = (diff_da) / r_da
            if delta_da < thres_da and r_seg_id == None:
                seg_r_id = seg_r_id + 1
                stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
                stream_segments_attribute_df.loc[index, "SavgRID"] = seg_r_id
                stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
                stream_segments_attribute_df.loc[index, "deltaDA"] = delta_da
                start_endxy = stream_segments_attribute_df.loc[index, "EndXY"]
                # FIND MATCHES FOR THE NEW START_ENDXY
                ssaIdxs = stream_segments_attribute_df.query(
                    "@start_endxy == StartXY"
                ).index.tolist()
                # DELETE PAIR IN LIST IF EXIST
                r_elev = stream_segments_attribute_df.loc[index, "EndElev"]
                r_id = index
                pair = (r_id, r_elev)
                # PRINT PAIR
                if pair in elev_list_count:
                    elev_list_count.remove(pair)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE ROLLING MEAN OF SEGMENT USING 25 REACHES UPSTREAM AND 25 REACHES DOWNSTREAM IN A
SEGMENT
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    print("calculating rolling mean")

    # OPTIMIZED CODE
    seg_id_l_unique = stream_segments_attribute_df["SavgID"].unique().tolist()

    # LOOP THROUGH TABLE AND COLLECT REACHIDS
    avg_window = {}

    for seg_id in seg_id_l_unique:
        # DONT LOOP AND CHECK CONDITIONS, GET INDEXES THAT MATCH CONDITION INSTEAD THEN LOOP
        ssaIdxs = stream_segments_attribute_df.query(
            "@seg_id == SavgID"
        ).index.tolist()
        for index in ssaIdxs:
            seg_r_id = stream_segments_attribute_df.loc[index, "SavgRID"]
            slope_raw = stream_segments_attribute_df.loc[index, "Slope"]
            avg_window[seg_r_id] = slope_raw

        avg_window_df = pd.Series(avg_window)
        roll_avg_df = avg_window_df.rolling(
            window=Savg_cell_window, min_periods=1, center=True
        ).mean()

        # LOOP THROUGH SET AND CALCULATE ROLLING MEAN
        for i, j in roll_avg_df.items():
            seg_r_id_avg = i
            avg_value = j
            ssaIdxs = stream_segments_attribute_df.query(
                "(@seg_id == SavgID) & (@seg_r_id_avg == SavgRID)"
            ).index.tolist()
            for index in ssaIdxs:
                stream_segments_attribute_df.loc[index, "SlopeAVGLU"] = abs(avg_value)

    # MUST FILL NULL SLOPE WITH 0
    stream_segments_attribute_df.fillna(value=0, inplace=True)

    stream_segments_smoothedslope_df_path = os.path.join(watershed_output,"stream_segments_SAVGslope.shp")
    stream_segments_attribute_df.to_file(
        stream_segments_smoothedslope_df_path, driver="ESRI Shapefile"
    )

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments_SAVGslope'
    stream_segments_attribute_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_segments_SAVGslope.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)

    print("SAVG Written")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE RURAL STREAM POWER
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    print("calculating rural stream power")

    # CALCULATE TOTAL STREAM POWER (SP) AND SPECIFIC STREAM POWER (SSP)
    stream_segments_attribute_df["RuralSP"] = pd.to_numeric(
        9810
        * stream_segments_attribute_df["SlopeAVGLU"]
        * stream_segments_attribute_df["Q_m3pers"],
        errors='coerce'
    )

    # RURAL SSP (EMPIRICAL FORMUL - DA SSP)
    stream_segments_attribute_df["RuralSSP"] = pd.to_numeric(
        stream_segments_attribute_df["RuralSP"] / stream_segments_attribute_df["RurWidth"],
        errors='coerce'
    )

    # FIND THE MAXIMUM RURAL WIDTH OLD METHOD
    max_val_oldmethod = stream_segments_attribute_df["RurWidth"].max()

    # # CALCULATE D84
    # # EQUATION 16 FROM FERGUSON (2005) ASSUMPTION: DI = DB
    # # CONSTANTS FOR D84 PREDICTION
    gamma = 9790 # N/M3
```

```
    kappa = 0.41 # CONSTANT FROM FERGUSON 2005 - WORKS FOR D IN MM, POWER IN W/M2
    R = 1.65 # SUBMERGED SPECIFIC GRAVITY
    rho = 1000 # DENSITY OF WATER
    theta_cb = 0.045 # SHEAR STRESS THRESHOLD
    m =2.80 # ROUGHNESS MULTIPLIER FOR D84 FROM LOPEZ AND BARRANGAN E.G. HEY 1979
    gee = 9.81 #ACCELERATION BY GRAVITY

    # C = LOG10(30*THETA_CB*R/(E*M*!S_MPERM!))"
    # EXPRESSIONDB =
"((KAPPA*!SPOWER_WPERM!/(2.30*RHO*LOG10(30*THETA_CB*R/(EXP*M*!S_MPERM!))))**(2/3))/(THETA_CB*R*GEE)"
    stream_segments_attribute_df["SlopeAVGLU"] = pd.to_numeric(stream_segments_attribute_df["SlopeAVGLU"], errors='coerce')
    stream_segments_attribute_df["eqlogC"] = np.log10(30 * theta_cb * R / (np.exp(1) * m *
np.abs(stream_segments_attribute_df["SlopeAVGLU"])))
    stream_segments_attribute_df["D84_mm"] = ((((kappa * stream_segments_attribute_df["RuralSSP"]) / (2.30 * rho *
stream_segments_attribute_df["eqlogC"])) ** (2.0 / 3)) / (theta_cb * R * gee)) * rho

    # SAVE FILE, MUST FILL NULL WITH 0 OR THE COLUMN WILL BE IMPORTED AS A STRING
    stream_segments_attribute_df.fillna(value=0, inplace=True)

    stream_segments_attribute_df_path = os.path.join(watershed_output,"stream_segments_attribute.shp")
    stream_segments_attribute_df.to_file(
        stream_segments_attribute_df_path, driver="ESRI Shapefile"
    )

    # SAVE A COPY OF SEGMENTS FOR MANIPULATION
    stream_segments_attribute_df_path_c = os.path.join(watershed_output,"stream_segments_attribute_edited.shp")
    stream_segments_attribute_df.to_file(
        stream_segments_attribute_df_path_c, driver="ESRI Shapefile"
    )

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments_attribute'
    stream_segments_attribute_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_segments_attribute.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    print("Rural stream power calculation ")
    gdf_strm = gpd.read_file(
        stream_segments_attribute_df_path_c
    )  # READ THE ORIGINAL SHAPEFILE

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments_attribute_edited'
    gdf_strm.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    copy_strm_net = (
        str(watershed_output) + "\Final_Rural_Stream_Power_Output.shp"
    )  # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
    gdf_strm.to_file(copy_strm_net, driver="ESRI Shapefile")

## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## END OF RURAL STREAM POWER CALCULATION
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## WRITE SHAPEFILES TO A GEOPACKAGE TO SAVE SPACE
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_points_initial.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_segments_attribute.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_vector_initial.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'dem_points.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_vector.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'hecpt_buffers50m_join.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'dem_points_initial.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'hecpt_buffers50m.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    with rasterio.open(dem) as src:
        x_res, y_res = src.res

    # CREATE A FOLDER TO STORE INTERMEDIATE FILES
    intermediate_storage_files = os.path.join(watershed_output, "LU_Outputs")
    if os.path.exists(intermediate_storage_files):
        print("FOLDER EXISTS")
    else:
        intermediate_dir = os.makedirs(intermediate_storage_files)
        print("FOLDER CREATED")

    # IMPORT FLOW ACCUMULATION
    flow_accum = os.path.join(watershed_output, "flow_accum.tif")

    # IMPORT FLOW DIRECTION
    flow_dir = os.path.join(watershed_output, "flow_dir.tif")
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## PRJ FILES FOR ALL OF THE SHAPEFILES
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
    # DEFINE WHAT THE CRS IS
    crs = pyproj.CRS.from_epsg(int(dem_espg2int))

    # WRITE THE CRS TO A .PRJ FILE
    # ORIGINAL DISCHARGE POINTS
    with open(
        os.path.join(watershed_output, "stream_points_fields_renamed.prj"), "w"
    ) as prj_file:
        prj_file.write(crs.to_wkt())
    data1 = gpd.read_file(stream_point_field_rename_path)
    data1 = data1.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA1.CRS)

    # STREAM_NET
    with open(
        os.path.join(watershed_output, "stream_segments_attribute_edited.prj"), "w"
    ) as prj_file:
        prj_file.write(crs.to_wkt())
    data2 = gpd.read_file(stream_segments_attribute_df_path_c)
    data2 = data2.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA2.CRS)

    # COPY OF STREAM_NET
```

```
    with open(
        os.path.join(watershed_output, "Final_Rural_Stream_Power_Output.prj"), "w"
    ) as prj_file:
        prj_file.write(crs.to_wkt())
    data3 = gpd.read_file(copy_strm_net)
    data3 = data3.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA3.CRS)

    with open(os.path.join(watershed_output, "basin_selected.prj"), "w") as prj_file:
        prj_file.write(crs.to_wkt())
    data4 = gpd.read_file(basin_polygon_sub_path)
    data4 = data4.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA4.CRS)

    if lushapefileanalysis == "Y":
## ---------------------------------------------------------------------------------------------------------------------------------------------
## PREPROCESS THE INPUT RASTERS
## ---------------------------------------------------------------------------------------------------------------------------------------------
        # RESOLUTION EXTRACTED AT TOP
        with rasterio.open(dem_clipped_path) as src:
            # GET THE AFFINE TRANSFORM FROM THE METADATA
            transform = src.transform

            # THE RESOLUTION IS STORED IN THE CELLSIZE OF THE AFFINE TRANSFORM
            x_resolution = transform.a
            y_resolution = transform.e

            # THE CELLSIZE IS THE ABSOLUTE VALUE OF THE RESOLUTION, AS IT MAY BE NEGATIVE IF THE RASTER IS
ROTATED
            x_cellsize = abs(x_resolution)
            y_cellsize = abs(y_resolution)

        crs = rasterio.crs.CRS.from_epsg(int(dem_espg2int))

        # ABSORPTION RASTER
        # PREPROCESS THE ABSORPTION RASTER, WE WANT TO HAVE ZERO ABSORPTION
        # OPEN THE RASTER FILE
        with rasterio.open(dem_clipped_path) as src:
            # READ THE DATA INTO A NUMPY ARRAY
            data = src.read(1)

            # SET ALL CELLS TO THE DESIRED VALUE
            data[:] = 0

            # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
            meta = src.meta.copy()
            meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

            absorption_file = os.path.join(intermediate_storage_files, "Absorption.tif")
            # WRITE THE UPDATED ARRAY TO A NEW FILE
            with rasterio.open(absorption_file, "w", **meta) as dst:
                dst.write(data, 1)

        # EFFICIENCY RASTER
        # PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY
        with rasterio.open(dem_clipped_path) as src:
            # READ THE DATA INTO A NUMPY ARRAY
            data = src.read(1)

            # SET ALL CELLS TO THE DESIRED VALUE
            data[:] = 1

            # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
            meta = src.meta.copy()
            meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

            efficiency_file = os.path.join(intermediate_storage_files, "Efficiency.tif")
            # WRITE THE UPDATED ARRAY TO A NEW FILE
            with rasterio.open(efficiency_file, "w", **meta) as dst:
                dst.write(data, 1)
```

```
## ---------------------------------------------------------------------------------------------------------------------------------------------
## CROP LAND USE SHAPEFILE TO DEM OUTLINE - TO BE USED IN THE WHITEBOX TOOLS FUNCTION
## ---------------------------------------------------------------------------------------------------------------------------------------------
        # SHAPEFILE FILE
        lu_shapefile = gpd.read_file(lu_file)
        # READ SHAPEFILE
        crop_shapefile = gpd.read_file(basin_polygon_sub_path)
        buffered_crop = crop_shapefile.buffer(50)  # CREATES A BUFFER AROUND THE BOUNDARY

        crop_shapefile.crs = pyproj.CRS("EPSG:" + str(dem_espg2int))

        # CREATE AN EMPTY LIST TO HOLD THE CROPPED LAND USE POLYGONS
        cropped_lu_list = []

        # LOOP THROUGH EACH POLYGON IN THE CROP SHAPEFILE
        for poly in buffered_crop.geometry:
            # CROP THE LAND USE POLYGONS USING THE CURRENT CROP POLYGON
            cropped = gpd.overlay(
                lu_shapefile, gpd.GeoDataFrame(geometry=[poly]), how="intersection"
            )

            # APPEND THE CROPPED POLYGONS TO THE LIST
            cropped_lu_list.append(cropped)

        # CONCATENATE THE CROPPED POLYGONS INTO A SINGLE GEODATAFRAME
        cropped_lu = gpd.GeoDataFrame(
            pd.concat(cropped_lu_list, ignore_index=True), crs=lu_shapefile.crs
        )

        # WRITE THE CROPPED LAND USE POLYGONS TO A NEW SHAPEFILE
        lu_clipped = os.path.join(intermediate_storage_files, "LU_Clipped.shp")
        cropped_lu.to_file(lu_clipped)
        lu_clip = gpd.read_file(lu_clipped)
        # PRINT(LU_CLIP.CRS)
        print("LU Clip spatial extents:", lu_clip.total_bounds)
## ---------------------------------------------------------------------------------------------------------------------------------------------
## ASSIGN IMPERVIOUSNESS VALUES - IF THERE ARE NO IMPERVIOUS VALUES IN THE SHAPEFILE
## VALUES IN SHAPEFILE SHOULD BE RATIOS
## ---------------------------------------------------------------------------------------------------------------------------------------------
        # TAKE THE CROPPED LU SHAPEFILE TO THE DEM OUTLINE AND
        # ASSIGN IMPERVIOUSNESS TO EACH OF THE CELLS
        # READ THE LAND USE SHAPEFILE
        lu_sp = gpd.read_file(lu_clipped)

        # DEFINE THE COLUMN TO CHECK FOR TYPE OF LAND USE
        column_to_check = "TYPE"  # BASED ON USER INPUT

        # DEFINE A NEW COLUMN
        new_column = "Imperv"

        # MAP VALUES USING THE VALUE_MAPPING DICTIONARY
        lu_sp[new_column] = lu_sp[column_to_check].map(value_mapping).fillna(0)

        # DIVIDE BY 100 TO GET RATIO
        lu_sp[new_column] = lu_sp[new_column] / 100

        # WRITE THE UPDATED GEODATAFRAME TO A NEW SHAPEFILE
        lu_new_field = os.path.join(
            intermediate_storage_files, "Land_Use_Shapefile_Impervious_Values.shp"
        )

        lu_sp.to_file(lu_new_field, driver="ESRI Shapefile")

        print("NEW COLUMN OF RUNOFF COEFFICIENTS ADDED", lu_new_field)
## ---------------------------------------------------------------------------------------------------------------------------------------------
## CREATE A NEW FIELD AND CALCULATE POLYGON AREA
## ---------------------------------------------------------------------------------------------------------------------------------------------
        # OPEN THE INPUT SHAPEFILE
        with fiona.open(lu_new_field, "r") as src:
            # GET THE SCHEMA OF THE INPUT SHAPEFILE
```

```
        schema = src.schema.copy()

        # ADD THE FIELD TO THE SCHEMA
        schema["properties"]["area"] = "float:19.11"

        # CREATE A NEW SHAPEFILE TO WRITE THE CALCULATED AREA TO
        output_shapefile = os.path.join(
            intermediate_storage_files, "LU_Imp_Polygon_Area.shp"
        )
        with fiona.open(
            output_shapefile, "w", "ESRI Shapefile", schema, src.crs
        ) as dst:
            # LOOP THROUGH THE FEATURES IN THE INPUT SHAPEFILE
            for feature in src:
                # GET THE GEOMETRY OF THE FEATURE
                polygon = shapely.geometry.shape(feature["geometry"])

                # CALCULATE THE AREA OF THE POLYGON
                area = polygon.area

                # UPDATE THE FEATURE TO INCLUDE THE CALCULATED AREA
                feature["properties"]["area"] = area

                # WRITE THE UPDATED FEATURE TO THE OUTPUT SHAPEFILE
                dst.write(feature)
    print("Column Populated with Area")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CREATE A RASTER OF THE LAND USE IMPERVIOUSNESS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # SHAPEFILE THAT IS TO BE WRITTEN TO RASTER
    # POLYGONSTORASTER
    # READ POLYGON SHAPEFILE
    polygon_toraster = gpd.read_file(output_shapefile)
    # POLYGON_TORASTER = POLYGON_TORASTER.RENAME(COLUMNS={'IMPERVIOUS': 'IMP'})

    # REPROJECTING TO COORDINATE SYSTEM
    polygon_toraster = polygon_toraster.to_crs("EPSG:" + str(dem_espg2int))

    # USING GEOCUBE TO RASTERIZE THE VECTOR

    # LABEL THE NAME OF THE COLUMN FOR THE MEASUREMENTS DEPENDING ON INPUT
    measurement_var = new_column

    polygon_toraster_rasterize = make_geocube(
        vector_data=polygon_toraster,
        measurements=[measurement_var],
        resolution=(-y_res, x_res),
        fill=0,
    )

    # SAVE RASTER CENSUS RASTER
    # RASTER NAME
    lu_raster = os.path.join(intermediate_storage_files, "LU_Raster.tif")
    polygon_toraster_rasterize.rio.to_raster(
        lu_raster
    ) # IF IT IS OPEN SOMEWHERE THE SCRIPT WILL NOT RUN PAST HERE, WILL NEED TO RUN IN A NEW DIRECTORY

    with rasterio.open(lu_raster, "r+") as dst:
        # DEFINE THE CRS USING EPSG CODE
        crs = rasterio.crs.CRS.from_epsg(str(dem_espg2int))

        # WRITE THE CRS TO THE FILE
        dst.crs = crs
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CROP THE LAND USE TIF'S
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # SHAPEFILE FILE
    ras_lu = lu_raster

    # READ SHAPEFILE
    gdf = gpd.read_file(basin_polygon_sub_path)

    gdf["count"] = [i + 1 for i, _ in enumerate(gdf.geometry)]
    outputfile = os.path.join(intermediate_storage_files, "basin_selected_count.shp")
    gdf.to_file(outputfile)

    gdf1 = gpd.read_file(outputfile)

    # GDF.CRS
    names = [x for x in gdf1["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

    # READ SHAPEFILE
    gdf3 = gpd.read_file(outputfile)

    # GDF.CRS
    names = [x for x in gdf3["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

    # READ RASTER DATA
    ras_data = rasterio.open(ras_lu)

    # OUTPUT PATH
    out_raster_lu = os.path.join(
        intermediate_storage_files, "LURasterIntermediateSteps"
    )
    if not os.path.exists(out_raster_lu):
        os.makedirs(out_raster_lu)

    # SAVE OUTPUT FILES AS PER SHAPEFILE FEATURES
    for i in range(len(gdf3)):
        geom = []
        coord = shapely.geometry.mapping(gdf3)["features"][i]["geometry"]
        geom.append(coord)
        with rasterio.open(ras_lu) as src:
            out_image, out_transform = rasterio.mask.mask(src, geom, crop=True)
            out_meta = src.meta
            out_meta.update(
                {
                    "driver": "GTiff",
                    "height": out_image.shape[1],
                    "width": out_image.shape[2],
                    "transform": out_transform,
                }
            )
            # FILE_PATH = ''
            with rasterio.open(
                f"{out_raster_lu}/{names[i]}.tif", "w", **out_meta
            ) as dest:
                dest.write(out_image)

    luraster_files = glob.glob(out_raster_lu + "/*.tif")
    mergedfilelu = os.path.join(intermediate_storage_files, "Final_LU_Raster.tif")

    # OPEN INPUT RASTERS
    src_files_to_mosaic2 = []
    # PRINT(SRC_FILES_TO_MOSAIC2)

    for file in luraster_files:
        src = rasterio.open(file)
        src_files_to_mosaic2.append(src)

    # MERGE INPUT RASTERS INTO ONE
    mosaic, out_trans = rasterio.merge.merge(src_files_to_mosaic2)

    # CLOSE ALL INPUT RASTER FILES
    for src in src_files_to_mosaic2:
        src.close()

    # UPDATE METADATA FOR OUTPUT RASTER
    out_meta = src.meta.copy()
    out_meta.update(
```

```
        {
            "driver": "GTiff",
            "height": mosaic.shape[1],
            "width": mosaic.shape[2],
            "transform": out_trans,
        }
    )

    # WRITE OUTPUT RASTER TO DISK
    with rasterio.open(mergedfilelu, "w", **out_meta) as dest:
        dest.write(mosaic)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## REPROJECT THE FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    reprojectedfiles = os.path.join(intermediate_storage_files, "WhiteBoxToolFiles")
    if not os.path.exists(reprojectedfiles):
        os.makedirs(reprojectedfiles)

    # DEFINE EPSG CODE
    epsg_code = dem_espg2int

    # CREATE SPATIALREFERENCE OBJECT FROM EPSG CODE
    srs = osr.SpatialReference()
    srs.ImportFromEPSG(int(epsg_code))

    # CONVERT SPATIALREFERENCE OBJECT TO WKT STRING
    new_crs_wkt = srs.ExportToWkt()

    # COPY THE DEMS
    # OPEN INPUT RASTER FILE
    input_raster = gdal.Open(dem_clipped_path, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform = input_raster.GetGeoTransform()
    new_geotransform = (
        existing_geotransform[0],
        existing_geotransform[1],
        0,
        existing_geotransform[3],
        0,
        existing_geotransform[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster.SetGeoTransform(new_geotransform)
    input_raster.SetProjection(new_crs_wkt)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputdemreproj = os.path.join(reprojectedfiles, "Reprojected_DEM.tif")
    output_raster = driver.CreateCopy(outputdemreproj, input_raster)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster = None
    output_raster = None

    # COPY THE ABSORPTION
    # OPEN INPUT RASTER FILE
    input_raster1 = gdal.Open(absorption_file, gdal.GA_Update)

    driver = gdal.GetDriverByName("GTiff")
    outputabsreproj = os.path.join(reprojectedfiles, "Copy_Absorption.tif")
    output_raster1 = driver.CreateCopy(outputabsreproj, input_raster1)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster1 = None
    output_raster1 = None

    # COPY THE EFFICIENCY
    # OPEN INPUT RASTER FILE
    input_raster2 = gdal.Open(efficiency_file, gdal.GA_Update)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputeffreproj = os.path.join(reprojectedfiles, "Copy_Efficiency.tif")
    output_raster2 = driver.CreateCopy(outputeffreproj, input_raster2)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster2 = None
    output_raster2 = None

    # COPY THE LAND USE
    # OPEN INPUT RASTER FILE
    input_raster3 = gdal.Open(mergedfilelu, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform3 = input_raster3.GetGeoTransform()
    new_geotransform3 = (
        existing_geotransform3[0],
        existing_geotransform3[1],
        0,
        existing_geotransform3[3],
        0,
        existing_geotransform3[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster3.SetGeoTransform(new_geotransform3)
    input_raster3.SetProjection(new_crs_wkt)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputlureproj = os.path.join(reprojectedfiles, "Reprojected_LandUse.tif")
    output_raster3 = driver.CreateCopy(outputlureproj, input_raster3)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster3 = None
    output_raster3 = None

    # COPY THE FLOWACCUM
    # OPEN INPUT RASTER FILE
    input_raster4 = gdal.Open(flow_accum, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform4 = input_raster4.GetGeoTransform()
    new_geotransform4 = (
        existing_geotransform4[0],
        existing_geotransform4[1],
        0,
        existing_geotransform4[3],
        0,
        existing_geotransform4[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster4.SetGeoTransform(new_geotransform4)
    input_raster4.SetProjection(new_crs_wkt)
    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputflowaccumreproj = os.path.join(reprojectedfiles, "Reprojected_FlowAccum.tif")
    output_raster4 = driver.CreateCopy(outputflowaccumreproj, input_raster4)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster4 = None
    output_raster4 = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CHECK SPATIAL EXTENTS OF THE FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
```

```
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # DICTIONARY TO STORE PROCESSED RASTERS
    raster_dict = {}

    # LOOP THROUGH ALL GEOTIFF FILES IN THE FOLDER
    for file_name in os.listdir(reprojectedfiles):
        if file_name.endswith(".tif"):
            file_path = os.path.join(reprojectedfiles, file_name)

            # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
            raster = gdal.Open(file_path, gdal.GA_Update)
            raster_gt = raster.GetGeoTransform()
            raster_proj = raster.GetProjection()
            raster_size = (raster.RasterXSize, raster.RasterYSize)

            # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
            if (
                raster_gt == small_raster.GetGeoTransform()
                and raster_proj == small_raster.GetProjection()
                and raster_size == small_size
            ):
                print(
                    f"{file_name} is already in the correct CRS, resolution, and size."
                )
            else:
                # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
                raster_update_path = os.path.join(
                    reprojectedfiles, f"{file_name.split('.')[0]}_update.tif"
                )
                new_raster = gdal.Translate(
                    raster_update_path,
                    raster,
                    format="GTiff",
                    width=small_size[0],
                    height=small_size[1],
                )
                new_raster.SetGeoTransform(small_raster.GetGeoTransform())
                new_raster.SetProjection(small_raster.GetProjection())
                print(f"{file_name} has been updated.")

            # CLOSE THE RASTER FILE
            raster = None
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## LOOP THROUGH THE FOLDER AND CHECK IF THERE ARE UPDATE FILES THEN ASSIGN IT VARIABLES
## IF THERE IS NO CHANGE IN CRS/RES/EXTENTS USE THE OLD TIF
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    wbt_demfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_DEM_update") and filename.endswith(".tif"):
            wbt_demfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_demfile is not None:
        print("Update file found:", wbt_demfile)
    else:
        wbt_demfile = outputdemreproj
        print("Using the old file:", wbt_demfile)

    wbt_absfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Absorption_update") and filename.endswith(
            ".tif"
        ):
            wbt_absfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_absfile is not None:
        print("Update file found:", wbt_absfile)
    else:
        wbt_absfile = outputabsreproj
        print("Using the old file:", wbt_absfile)

    wbt_efffile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Efficiency_update") and filename.endswith(
            ".tif"
        ):
            wbt_efffile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_efffile is not None:
        print("Update file found:", wbt_efffile)
    else:
        wbt_efffile = outputeffreproj
        print("Using the old file:", wbt_efffile)
    wbt_lufile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_LandUse_update") and filename.endswith(
            ".tif"
        ):
            wbt_lufile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_lufile is not None:
        print("Update file found:", wbt_lufile)
    else:
        wbt_lufile = outputlureproj
        print("Using the old file:", wbt_lufile)

    dividing_flowaccum = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_FlowAccum_update") and filename.endswith(
            ".tif"
        ):
            dividing_flowaccum = os.path.join(reprojectedfiles, filename)
            break
    if dividing_flowaccum is not None:
        print("Update file found:", dividing_flowaccum)
    else:
        dividing_flowaccum = outputflowaccumreproj
        print("Using the old file:", dividing_flowaccum)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## USE WBT TO OUTPUT THE D8 MASSFLUX
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # BRING IN WHITEBOX TOOLS
    wbt = whitebox.WhiteboxTools()

    # DEFINE THE OUTPUT_FILE
    dmassoutput = os.path.join(intermediate_storage_files, "D8MassFluxOutput.tif")

    # USE D8MASSFLUX TOOL AND OUTPUT A FILE
    wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_efffile, wbt_absfile, dmassoutput)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## SAVE IN LOG SCALE
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    with rasterio.open(dmassoutput) as src:
        # READ THE RASTER DATA INTO A NUMPY ARRAY
        data = src.read(1)
        # APPLY LOG TRANSFORMATION
        data_log = np.log(data + 1)

        # CREATE NEW RASTER FILE
        profile = src.profile  # META DATA COPY
        profile.update(nodata=np.nan)

        LogScale_D8 = os.path.join(
            intermediate_storage_files, "LogScale_D8MassFlux.tif"
        )
        with rasterio.open(LogScale_D8, "w", **profile) as dst:
            dst.write(data_log, 1)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## GET THE IMPERVIOUSNESS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
    # TO GET THE IMPERVIOUSNESS = OUTPUTFILE/FLOW ACCUM
    # WE NEED FLOW ACCUM, D8MASSFLUXOUTPUT, AN OUTPUT FOR THE CLIPPED FILE, AN OUTPUT FOR THE
REPROJECTED FILE
    # AND AN OUTPUT FOR THE IMPERVIOUSNESS
    # OUTPUT_FILE = D8MASSFLUX OUTPUT
    output_impervious = os.path.join(intermediate_storage_files, "Final_Imperviousness.tif")

    # DIVIDING!
    # OPEN THE TWO RASTER FILES
    src3 = gdal.Open(dmassoutput)
    src4 = gdal.Open(dividing_flowaccum)

    # GET THE SIZE OF THE RASTERS
    cols = src3.RasterXSize
    rows = src3.RasterYSize

    # CREATE AN OUTPUT RASTER WITH THE SAME SIZE AND PROJECTION AS THE INPUT RASTERS
    driver = gdal.GetDriverByName("GTiff")
    out_raster = driver.Create(output_impervious, cols, rows, 1, gdal.GDT_Float32)
    out_raster.SetProjection(src3.GetProjection())
    out_raster.SetGeoTransform(src3.GetGeoTransform())

    # READ THE DATA FROM BOTH RASTERS INTO NUMPY ARRAYS
    data1 = src3.ReadAsArray()
    data2 = src4.ReadAsArray()

    # PERFORM ELEMENT-WISE DIVISION OF THE TWO ARRAYS
    result = np.divide(data1, data2)

    # SET THE RANGE OF VALUES TO WRITE TO THE OUTPUT RASTER
    data_min = 0
    data_max = 1

    # CLIP THE RESULT ARRAY TO THE SPECIFIED RANGE
    result_clipped = np.clip(result, data_min, data_max)

    # WRITE THE RESULT TO THE OUTPUT RASTER
    out_raster.GetRasterBand(1).WriteArray(result_clipped)

    # CLOSE THE RASTERS
    src3 = None
    src4 = None
    out_raster = None

    print("Divided Rasters, Imperviousness Found")

    with rasterio.open(output_impervious) as src:
        imp = src.read(1)

    # PLOT THE RASTER
    plt.figure(figsize=(10, 10))
    rasterio.plot.show(imp, transform=src.transform, cmap="terrain")
    plt.show()

    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
    clippedimperviousraster = gdal.Open(output_impervious, gdal.GA_Update)
    rasterimp_gt = clippedimperviousraster.GetGeoTransform()
    rasterimp_proj = clippedimperviousraster.GetProjection()
    rasterimp_size = (
        clippedimperviousraster.RasterXSize,
        clippedimperviousraster.RasterYSize,
    )

    # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
    if (
        rasterimp_gt == small_raster.GetGeoTransform()
        and rasterimp_proj == small_raster.GetProjection()
        and rasterimp_size == small_size
    ):
        print("Imperviousness is already in the correct CRS, resolution, and size.")
    else:
        # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
        raster_imp_update = os.path.join(
            intermediate_storage_files, "Final_Imperviousness_update.tif"
        )
        new_raster1 = gdal.Translate(
            raster_imp_update,
            clippedimperviousraster,
            format="GTiff",
            width=small_size[0],
            height=small_size[1],
        )
        new_raster1.SetGeoTransform(small_raster.GetGeoTransform())
        new_raster1.SetProjection(small_raster.GetProjection())
        print("Imperviousness has been updated.")

    # CLOSE THE RASTER FILES
    raster = None

    divided_imp = None
    for filename in os.listdir(intermediate_storage_files):
        if filename.startswith("Final_Imperviousness_update") and filename.endswith(
            ".tif"
        ):
            divided_imp = os.path.join(intermediate_storage_files, filename)
            break
    if divided_imp is not None:
        print("Update file found:", divided_imp)
    else:
        divided_imp = output_impervious
        print("Using the old file:", divided_imp)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## ATTACH IMPERVIOUSNESS TO THE STREAM SEGMENTS BASED ON THE END XY
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # FIRST EXTRACT THE VALUE OF THE RASTER AT THE STREAM POINT LOCATION
    # STREAM POINT FILE = STREAM_POINT_FIELD_RENAME_PATH
    # RASTER = OUTPUT_IMP

    extract_pts = os.path.join(intermediate_storage_files, "Urban_Stream_Points_Imperviousness.shp")

    # READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
    gdf = gpd.read_file(stream_point_field_rename_path)

    gdf["D8Flux"] = np.nan
    gdf["LUFloAccum"] = np.nan

    # OPEN THE RASTER FILE FOR READING
    with rasterio.open(divided_imp) as src, rasterio.open(
        dmassoutput
    ) as src1, rasterio.open(flow_accum) as src2:
        # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
        for index, row in gdf.iterrows():
            # GET THE X AND Y COORDINATES OF THE POINT
            point = row.geometry
            x, y = point.x, point.y

            # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
            impvalue = list(src.sample([(x, y)]))
            massfluxvalue = list(src1.sample([(x, y)]))
            flowaccumvalue = list(src2.sample([(x, y)]))

            # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
            gdf.loc[index, "Imp"] = impvalue[0][0]
            gdf.loc[index, "D8Flux"] = massfluxvalue[0][0]
            gdf.loc[index, "LUFloAccum"] = flowaccumvalue[0][0]
```

```
    # WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
    gdf.to_file(extract_pts)

    print("Impervious Values Extracted to Match Stream Points")

    # NOW MERGE THE VALUES BASED ON THE COLUMN
    points_gdf = gpd.read_file(extract_pts)
    lines_gdf = gpd.read_file(copy_strm_net)

    intstep = os.path.join(intermediate_storage_files, "Copy_Rural_Stream_Power.shp")
    urban_segments = os.path.join(
        intermediate_storage_files, "Urban_Imperviousness_Attachment.shp"
    )

    lines_gdf.to_file(intstep)

    lines_gdf2 = gpd.read_file(intstep)

    # POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
    # THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
    common_column = "pXY"

    # MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
    merged_gdf = lines_gdf2.set_index(common_column).join(
        points_gdf.set_index(common_column)[["Imp", "D8Flux", "LUFloAccum"]]
    )
    print(merged_gdf.columns)

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf.to_file(urban_segments)

    gdf = gpd.read_file(urban_segments)
    print(
        "Impervious, D8 Mass Flux, Flow Accumulation Values Appended to Stream Segments"
    )

elif lurasteranalysis == "Y":
    print("Raster of LU Imperviousness is provided")
    # ENSURE INTERMEDIATE STORAGE FOLDER EXISTS
    if not os.path.exists(intermediate_storage_files):
        os.makedirs(intermediate_storage_files)

    # DEFINE THE NEW LOCATION TO SAVE THE RASTER
    impraster = os.path.join(intermediate_storage_files, "LU_Imperviousness_raster.tif")

    # OPEN THE ORIGINAL RASTER FILE
    with rasterio.open(lurasteranalysispath) as src:
        raster_data = src.read(1)  # READ THE FIRST BAND OF THE RASTER
        profile = src.profile  # GET THE METADATA FOR THE RASTER
        # NORMALIZE IMPERVIOUSNESS VALUES (IF VALUES ARE IN PERCENTAGE FORM)
        raster_data = raster_data / 100.0
        raster_data = np.clip(raster_data, 0, 1)  # CLAMP VALUES BETWEEN 0 AND 1

    # SAVE THE RASTER TO THE OUTPUT PATH WITHOUT MODIFICATION
    with rasterio.open(impraster, 'w', **profile) as dst:
        dst.write(raster_data, 1)  # WRITE THE ORIGINAL RASTER DATA TO THE FIRST BAND

    print(f"Raster saved successfully to {impraster}")

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## PREPROCESS THE INPUT RASTERS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # RESOLUTION EXTRACTED AT TOP
    with rasterio.open(dem_clipped_path) as src:
        # GET THE AFFINE TRANSFORM FROM THE METADATA
        transform = src.transform

        # THE RESOLUTION IS STORED IN THE CELLSIZE OF THE AFFINE TRANSFORM
        x_resolution = transform.a
        y_resolution = transform.e

        # THE CELLSIZE IS THE ABSOLUTE VALUE OF THE RESOLUTION, AS IT MAY BE NEGATIVE IF THE RASTER IS ROTATED
        x_cellsize = abs(x_resolution)
        y_cellsize = abs(y_resolution)

    crs = rasterio.crs.CRS.from_epsg(int(dem_espg2int))

    # ABSORPTION RASTER
    # PREPROCESS THE ABSORPTION RASTER, WE WANT TO HAVE ZERO ABSORPTION
    # OPEN THE RASTER FILE
    with rasterio.open(dem_clipped_path) as src:
        # READ THE DATA INTO A NUMPY ARRAY
        data = src.read(1)

        # SET ALL CELLS TO THE DESIRED VALUE
        data[:] = 0

        # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
        meta = src.meta.copy()
        meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

        absorption_file = os.path.join(intermediate_storage_files, "Absorption.tif")
        # WRITE THE UPDATED ARRAY TO A NEW FILE
        with rasterio.open(absorption_file, "w", **meta) as dst:
            dst.write(data, 1)

    # EFFICIENCY RASTER
    # PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY
    with rasterio.open(dem_clipped_path) as src:
        # READ THE DATA INTO A NUMPY ARRAY
        data = src.read(1)

        # SET ALL CELLS TO THE DESIRED VALUE
        data[:] = 1

        # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
        meta = src.meta.copy()
        meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

        efficiency_file = os.path.join(intermediate_storage_files, "Efficiency.tif")
        # WRITE THE UPDATED ARRAY TO A NEW FILE
        with rasterio.open(efficiency_file, "w", **meta) as dst:
            dst.write(data, 1)

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CROP THE LAND USE TIF'S
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # SHAPEFILE FILE
    ras_lu = impraster

    # READ SHAPEFILE
    gdf = gpd.read_file(basin_polygon_sub_path)

    gdf["count"] = [i + 1 for i, _ in enumerate(gdf.geometry)]
    outputfile = os.path.join(intermediate_storage_files, "basin_selected_count.shp")
    gdf.to_file(outputfile)

    gdf1 = gpd.read_file(outputfile)

    # GDF.CRS
    names = [x for x in gdf1["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

    # READ SHAPEFILE
    gdf3 = gpd.read_file(outputfile)

    # GDF.CRS
    names = [x for x in gdf3["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

    # READ RASTER DATA
```

```
    ras_data = rasterio.open(ras_lu)

    # OUTPUT PATH
    out_raster_lu = os.path.join(
        intermediate_storage_files, "LURasterIntermediateSteps"
    )
    if not os.path.exists(out_raster_lu):
        os.makedirs(out_raster_lu)

    # SAVE OUTPUT FILES AS PER SHAPEFILE FEATURES
    for i in range(len(gdf3)):
        geom = []
        coord = shapely.geometry.mapping(gdf3)["features"][i]["geometry"]
        geom.append(coord)
        with rasterio.open(ras_lu) as src:
            out_image, out_transform = rasterio.mask.mask(src, geom, crop=True)
            out_meta = src.meta
            out_meta.update(
                {
                    "driver": "GTiff",
                    "height": out_image.shape[1],
                    "width": out_image.shape[2],
                    "transform": out_transform,
                }
            )
            # FILE_PATH = ''
            with rasterio.open(
                f"{out_raster_lu}/{names[i]}.tif", "w", **out_meta
            ) as dest:
                dest.write(out_image)

    luraster_files = glob.glob(out_raster_lu + "/*.tif")
    mergedfilelu = os.path.join(intermediate_storage_files, "Final_LU_Raster.tif")

    # OPEN INPUT RASTERS
    src_files_to_mosaic2 = []
    # PRINT(SRC_FILES_TO_MOSAIC2)

    for file in luraster_files:
        src = rasterio.open(file)
        src_files_to_mosaic2.append(src)

    # MERGE INPUT RASTERS INTO ONE
    mosaic, out_trans = rasterio.merge.merge(src_files_to_mosaic2)

    # CLOSE ALL INPUT RASTER FILES
    for src in src_files_to_mosaic2:
        src.close()

    # UPDATE METADATA FOR OUTPUT RASTER
    out_meta = src.meta.copy()
    out_meta.update(
        {
            "driver": "GTiff",
            "height": mosaic.shape[1],
            "width": mosaic.shape[2],
            "transform": out_trans,
        }
    )

    # WRITE OUTPUT RASTER TO DISK
    with rasterio.open(mergedfilelu, "w", **out_meta) as dest:
        dest.write(mosaic)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## REPROJECT THE FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    reprojectedfiles = os.path.join(intermediate_storage_files, "WhiteBoxToolFiles")
    if not os.path.exists(reprojectedfiles):
        os.makedirs(reprojectedfiles)

# DEFINE EPSG CODE
epsg_code = dem_espg2int

# CREATE SPATIALREFERENCE OBJECT FROM EPSG CODE
srs = osr.SpatialReference()
srs.ImportFromEPSG(int(epsg_code))

# CONVERT SPATIALREFERENCE OBJECT TO WKT STRING
new_crs_wkt = srs.ExportToWkt()

# COPY THE DEMS
# OPEN INPUT RASTER FILE
input_raster = gdal.Open(dem_clipped_path, gdal.GA_Update)

# GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
existing_geotransform = input_raster.GetGeoTransform()
new_geotransform = (
    existing_geotransform[0],
    existing_geotransform[1],
    0,
    existing_geotransform[3],
    0,
    existing_geotransform[5],
)

# SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
input_raster.SetGeoTransform(new_geotransform)
input_raster.SetProjection(new_crs_wkt)

# CREATE OUTPUT RASTER FILE
driver = gdal.GetDriverByName("GTiff")
outputdemreproj = os.path.join(reprojectedfiles, "Reprojected_DEM.tif")
output_raster = driver.CreateCopy(outputdemreproj, input_raster)

# CLOSE INPUT AND OUTPUT RASTER FILES
input_raster = None
output_raster = None

# COPY THE ABSORPTION
# OPEN INPUT RASTER FILE
input_raster1 = gdal.Open(absorption_file, gdal.GA_Update)

driver = gdal.GetDriverByName("GTiff")
outputabsreproj = os.path.join(reprojectedfiles, "Copy_Absorption.tif")
output_raster1 = driver.CreateCopy(outputabsreproj, input_raster1)

# CLOSE INPUT AND OUTPUT RASTER FILES
input_raster1 = None
output_raster1 = None

# COPY THE EFFICIENCY
# OPEN INPUT RASTER FILE
input_raster2 = gdal.Open(efficiency_file, gdal.GA_Update)

# CREATE OUTPUT RASTER FILE
driver = gdal.GetDriverByName("GTiff")
outputeffreproj = os.path.join(reprojectedfiles, "Copy_Efficiency.tif")
output_raster2 = driver.CreateCopy(outputeffreproj, input_raster2)

# CLOSE INPUT AND OUTPUT RASTER FILES
input_raster2 = None
output_raster2 = None

# COPY THE LAND USE
# OPEN INPUT RASTER FILE
input_raster3 = gdal.Open(mergedfilelu, gdal.GA_Update)

# GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
existing_geotransform3 = input_raster3.GetGeoTransform()
new_geotransform3 = (
```

```
        existing_geotransform3[0],
        existing_geotransform3[1],
        0,
        existing_geotransform3[3],
        0,
        existing_geotransform3[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster3.SetGeoTransform(new_geotransform3)
    input_raster3.SetProjection(new_crs_wkt)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputlureproj = os.path.join(reprojectedfiles, "Reprojected_LandUse.tif")
    output_raster3 = driver.CreateCopy(outputlureproj, input_raster3)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster3 = None
    output_raster3 = None

    # COPY THE FLOWACCUM
    # OPEN INPUT RASTER FILE
    input_raster4 = gdal.Open(flow_accum, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform4 = input_raster4.GetGeoTransform()
    new_geotransform4 = (
        existing_geotransform4[0],
        existing_geotransform4[1],
        0,
        existing_geotransform4[3],
        0,
        existing_geotransform4[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster4.SetGeoTransform(new_geotransform4)
    input_raster4.SetProjection(new_crs_wkt)
    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputflowaccumreproj = os.path.join(reprojectedfiles, "Reprojected_FlowAccum.tif")
    output_raster4 = driver.CreateCopy(outputflowaccumreproj, input_raster4)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster4 = None
    output_raster4 = None
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CHECK SPATIAL EXTENTS OF THE FILES
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # DICTIONARY TO STORE PROCESSED RASTERS
    raster_dict = {}

    # LOOP THROUGH ALL GEOTIFF FILES IN THE FOLDER
    for file_name in os.listdir(reprojectedfiles):
        if file_name.endswith(".tif"):
            file_path = os.path.join(reprojectedfiles, file_name)

            # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
            raster = gdal.Open(file_path, gdal.GA_Update)
            raster_gt = raster.GetGeoTransform()
            raster_proj = raster.GetProjection()
            raster_size = (raster.RasterXSize, raster.RasterYSize)

            # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
            if (
                raster_gt == small_raster.GetGeoTransform()
                and raster_proj == small_raster.GetProjection()
                and raster_size == small_size
            ):
                print(
                    f"{file_name} is already in the correct CRS, resolution, and size."
                )
            else:
                # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
                raster_update_path = os.path.join(
                    reprojectedfiles, f"{file_name.split('.')[0]}_update.tif"
                )
                new_raster = gdal.Translate(
                    raster_update_path,
                    raster,
                    format="GTiff",
                    width=small_size[0],
                    height=small_size[1],
                )
                new_raster.SetGeoTransform(small_raster.GetGeoTransform())
                new_raster.SetProjection(small_raster.GetProjection())
                print(f"{file_name} has been updated.")

            # CLOSE THE RASTER FILE
            raster = None
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## LOOP THROUGH THE FOLDER AND CHECK IF THERE ARE UPDATE FILES THEN ASSIGN IT VARIABLES
## IF THERE IS NO CHANGE IN CRS/RES/EXTENTS USE THE OLD TIF
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    wbt_demfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_DEM_update") and filename.endswith(".tif"):
            wbt_demfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_demfile is not None:
        print("Update file found:", wbt_demfile)
    else:
        wbt_demfile = outputdemreproj
        print("Using the old file:", wbt_demfile)

    wbt_absfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Absorption_update") and filename.endswith(
            ".tif"
        ):
            wbt_absfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_absfile is not None:
        print("Update file found:", wbt_absfile)
    else:
        wbt_absfile = outputabsreproj
        print("Using the old file:", wbt_absfile)

    wbt_efffile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Efficiency_update") and filename.endswith(
            ".tif"
        ):
            wbt_efffile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_efffile is not None:
        print("Update file found:", wbt_efffile)
    else:
        wbt_efffile = outputeffreproj
        print("Using the old file:", wbt_efffile)
    wbt_lufile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_LandUse_update") and filename.endswith(
            ".tif"
        ):
```

```
            wbt_lufile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_lufile is not None:
        print("Update file found:", wbt_lufile)
    else:
        wbt_lufile = outputlureproj
        print("Using the old file:", wbt_lufile)

    dividing_flowaccum = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_FlowAccum_update") and filename.endswith(
            ".tif"
        ):
            dividing_flowaccum = os.path.join(reprojectedfiles, filename)
            break
    if dividing_flowaccum is not None:
        print("Update file found:", dividing_flowaccum)
    else:
        dividing_flowaccum = outputflowaccumreproj
        print("Using the old file:", dividing_flowaccum)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## USE WBT TO OUTPUT THE D8 MASSFLUX
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    # BRING IN WHITEBOX TOOLS
    wbt = whitebox.WhiteboxTools()

    # DEFINE THE OUTPUT_FILE
    dmassoutput = os.path.join(intermediate_storage_files, "D8MassFluxOutput.tif")

    # USE D8MASSFLUX TOOL AND OUTPUT A FILE
    wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_efffile, wbt_absfile, dmassoutput)
    #WBT.D8_MASS_FLUX(DEM_CLIPPED_PATH, MERGEDFILELU, EFFICIENCY_FILE, ABSORPTION_FILE, DMASSOUTPUT)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## SAVE IN LOG SCALE
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    with rasterio.open(dmassoutput) as src:
        # READ THE RASTER DATA INTO A NUMPY ARRAY
        data = src.read(1)
        # APPLY LOG TRANSFORMATION
        data_log = np.log(data + 1)

        # CREATE NEW RASTER FILE
        profile = src.profile  # META DATA COPY
        profile.update(nodata=np.nan)

        LogScale_D8 = os.path.join(
            intermediate_storage_files, "LogScale_D8MassFlux.tif"
        )
        with rasterio.open(LogScale_D8, "w", **profile) as dst:
            dst.write(data_log, 1)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## GET THE IMPERVIOUSNESS
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    # TO GET THE IMPERVIOUSNESS = OUTPUTFILE/FLOW ACCUM
    # WE NEED FLOW ACCUM, D8MASSFLUXOUTPUT, AN OUTPUT FOR THE CLIPPED FILE, AN OUTPUT FOR THE
REPROJECTED FILE
    # AND AN OUTPUT FOR THE IMPERVIOUSNESS
    # OUTPUT_FILE = D8MASSFLUX OUTPUT
    output_impervious = os.path.join(intermediate_storage_files, "Final_Imperviousness.tif")

    # DIVIDING!
    # OPEN THE TWO RASTER FILES
    src3 = gdal.Open(dmassoutput)
    src4 = gdal.Open(dividing_flowaccum)

    # GET THE SIZE OF THE RASTERS
    cols = src3.RasterXSize
    rows = src3.RasterYSize

    # CREATE AN OUTPUT RASTER WITH THE SAME SIZE AND PROJECTION AS THE INPUT RASTERS
    driver = gdal.GetDriverByName("GTiff")
    out_raster = driver.Create(output_impervious, cols, rows, 1, gdal.GDT_Float32)
    out_raster.SetProjection(src3.GetProjection())
    out_raster.SetGeoTransform(src3.GetGeoTransform())

    # READ THE DATA FROM BOTH RASTERS INTO NUMPY ARRAYS
    data1 = src3.ReadAsArray()
    data2 = src4.ReadAsArray()

    # PERFORM ELEMENT-WISE DIVISION OF THE TWO ARRAYS
    result = np.divide(data1, data2)

    # SET THE RANGE OF VALUES TO WRITE TO THE OUTPUT RASTER
    data_min = 0
    data_max = 1

    # CLIP THE RESULT ARRAY TO THE SPECIFIED RANGE
    result_clipped = np.clip(result, data_min, data_max)

    # WRITE THE RESULT TO THE OUTPUT RASTER
    out_raster.GetRasterBand(1).WriteArray(result_clipped)

    # CLOSE THE RASTERS
    src3 = None
    src4 = None
    out_raster = None

    print("Divided Rasters, Imperviousness Found")

    with rasterio.open(output_impervious) as src:
        imp = src.read(1)

    # PLOT THE RASTER
    plt.figure(figsize=(10, 10))
    rasterio.plot.show(imp, transform=src.transform, cmap="terrain")
    plt.show()

    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
    clippedimperviousraster = gdal.Open(output_impervious, gdal.GA_Update)
    rasterimp_gt = clippedimperviousraster.GetGeoTransform()
    rasterimp_proj = clippedimperviousraster.GetProjection()
    rasterimp_size = (
        clippedimperviousraster.RasterXSize,
        clippedimperviousraster.RasterYSize,
    )

    # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
    if (
        rasterimp_gt == small_raster.GetGeoTransform()
        and rasterimp_proj == small_raster.GetProjection()
        and rasterimp_size == small_size
    ):
        print("Imperviousness is already in the correct CRS, resolution, and size.")
    else:
        # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
        raster_imp_update = os.path.join(
            intermediate_storage_files, "Final_Imperviousness_update.tif"
        )
        new_raster1 = gdal.Translate(
            raster_imp_update,
            clippedimperviousraster,
            format="GTiff",
            width=small_size[0],
            height=small_size[1],
        )
        new_raster1.SetGeoTransform(small_raster.GetGeoTransform())
```

```
        new_raster1.SetProjection(small_raster.GetProjection())
        print("Imperviousness has been updated.")

    # CLOSE THE RASTER FILES
    raster = None

    divided_imp = None
    for filename in os.listdir(intermediate_storage_files):
        if filename.startswith("Final_Imperviousness_update") and filename.endswith(
            ".tif"
        ):
            divided_imp = os.path.join(intermediate_storage_files, filename)
            break
    if divided_imp is not None:
        print("Update file found:", divided_imp)
    else:
        divided_imp = output_impervious
        print("Using the old file:", divided_imp)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## ATTACH IMPERVIOUSNESS TO THE STREAM SEGMENTS BASED ON THE END XY
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # FIRST EXTRACT THE VALUE OF THE RASTER AT THE STREAM POINT LOCATION
    # STREAM POINT FILE = STREAM_POINT_FIELD_RENAME_PATH
    # RASTER = OUTPUT_IMP

    extract_pts = os.path.join(intermediate_storage_files, "Urban_Stream_Points_Imperviousness.shp")

    # READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
    gdf = gpd.read_file(stream_point_field_rename_path)

    gdf["D8Flux"] = np.nan
    gdf["LUFloAccum"] = np.nan

    # OPEN THE RASTER FILE FOR READING
    with rasterio.open(divided_imp) as src, rasterio.open(
        dmassoutput
    ) as src1, rasterio.open(flow_accum) as src2:
        # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
        for index, row in gdf.iterrows():
            # GET THE X AND Y COORDINATES OF THE POINT
            point = row.geometry
            x, y = point.x, point.y

            # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
            impvalue = list(src.sample([(x, y)]))
            massfluxvalue = list(src1.sample([(x, y)]))
            flowaccumvalue = list(src2.sample([(x, y)]))

            # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
            gdf.loc[index, "Imp"] = impvalue[0][0]
            gdf.loc[index, "D8Flux"] = massfluxvalue[0][0]
            gdf.loc[index, "LUFloAccum"] = flowaccumvalue[0][0]

    # WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
    gdf.to_file(extract_pts)

    print("Impervious Values Extracted to Match Stream Points")

    # NOW MERGE THE VALUES BASED ON THE COLUMN
    points_gdf = gpd.read_file(extract_pts)
    lines_gdf = gpd.read_file(copy_strm_net)

    intstep = os.path.join(intermediate_storage_files, "Copy_Rural_Stream_Power.shp")
    urban_segments = os.path.join(
        intermediate_storage_files, "Urban_Imperviousness_Attachment.shp"
    )

    lines_gdf.to_file(intstep)

    lines_gdf2 = gpd.read_file(intstep)

    # POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
    # THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
    common_column = "pXY"

    # MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
    merged_gdf = lines_gdf2.set_index(common_column).join(
        points_gdf.set_index(common_column)[["Imp", "D8Flux", "LUFloAccum"]]
    )
    print(merged_gdf.columns)

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf.to_file(urban_segments)

    gdf = gpd.read_file(urban_segments)
    print(
        "Impervious, D8 Mass Flux, Flow Accumulation Values Appended to Stream Segments"
    )
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## ATTACH IMPERVIOUSNESS TO THE SEGMENTS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # FIRST EXTRACT THE VALUE OF THE RASTER AT THE STREAM POINT LOCATION
    # STREAM POINT FILE = STREAM_POINT_FIELD_RENAME_PATH
    # RASTER = OUTPUT_IMP

    extract_pts = os.path.join(
        intermediate_storage_files, "Urban_Stream_Points_Imperviousness_raster.shp"
    )

    # READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
    gdf = gpd.read_file(stream_point_field_rename_path)

    gdf["D8Flux"] = np.nan
    gdf["LUFloAccum"] = np.nan

    # OPEN THE RASTER FILE FOR READING
    with rasterio.open(divided_imp) as src, rasterio.open(
        dmassoutput
    ) as src1, rasterio.open(flow_accum) as src2:
        # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
        for index, row in gdf.iterrows():
            # GET THE X AND Y COORDINATES OF THE POINT
            point = row.geometry
            x, y = point.x, point.y

            # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
            impvalue = list(src.sample([(x, y)]))
            massfluxvalue = list(src1.sample([(x, y)]))
            flowaccumvalue = list(src2.sample([(x, y)]))

            # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
            gdf.loc[index, "Imp"] = impvalue[0][0]
            gdf.loc[index, "D8Flux"] = massfluxvalue[0][0]
            gdf.loc[index, "LUFloAccum"] = flowaccumvalue[0][0]

    # WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
    gdf.to_file(extract_pts)

    print("Impervious Values Extracted to Match Stream Points")

    # NOW MERGE THE VALUES BASED ON THE COLUMN
    points_gdf = gpd.read_file(extract_pts)
    lines_gdf = gpd.read_file(copy_strm_net)

    intstep = os.path.join(intermediate_storage_files, "Copy_Rural_Stream_Power.shp")
    urban_segments = os.path.join(
        intermediate_storage_files, "Urban_Imperviousness_Attachment_raster.shp"
    )

    lines_gdf.to_file(intstep)
```

```
    lines_gdf2 = gpd.read_file(intstep)

    # POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
    # THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
    common_column = "pXY"

    # MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
    merged_gdf = lines_gdf2.set_index(common_column).join(
        points_gdf.set_index(common_column)[["Imp"]]
    )
    print(merged_gdf.columns)

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf.to_file(urban_segments)

    gdf = gpd.read_file(urban_segments)
    print(
        "Impervious Values Appended to Stream Segments"
    )
## ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE THE DISCHARGE 2 YR
## ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    calculation = gpd.read_file(urban_segments)

    # CREATES NEW COLUMN AND SETS DEFAULT VALUE TO 0
    calculation["OGLUQ2m3_s"] = 0
    calculation["OGLUSP2"] = 0
    calculation["Wg_Nperm3"] = 9810
    calculation["OGLUSSP2"] = 0

    calculation["OGLUQ2m3_s"] = pd.to_numeric(calculation["OGLUQ2m3_s"], errors="coerce")
    calculation["SlopeAVGLU"] = pd.to_numeric(calculation["SlopeAVGLU"], errors="coerce")
    calculation["Wg_Nperm3"] = pd.to_numeric(calculation["Wg_Nperm3"], errors="coerce")
    calculation["RurWidth"] = pd.to_numeric(calculation["RurWidth"], errors="coerce")
    calculation["RuralSSP"] = pd.to_numeric(calculation["RuralSSP"], errors="coerce")

    # ORIGINAL DISCHARGE EQUATION
    # FORMULA FOR DISCHARGE IS Q=CA^X(IA^B) WHERE C, X AND B ARE COEFFICCIENTS, A IS THE DRAINAGE AREA AND
IA IS TOTAL IMPERVIOUSNESS PERCENTAGE
    # A IS IN THE COLUMN TITLED DA_KM2
    # IA IS IN COLUMN TITLED IMP

    c = rQ_a
    x = rQ_b
    b = 0.30

    calculation["OGLUQ2m3_s"] = calculation.apply(
        lambda row: c * (row["DA_km2"] ** x) * ((row["Imp"] * 100) ** b), axis=1
    )

    ## CALCULATE TOTAL POWER
    calculation["OGLUSP2"] = calculation.apply(
        lambda row: (row["OGLUQ2m3_s"]) * (row["SlopeAVGLU"]) * (row["Wg_Nperm3"]), axis=1
    )

    # ORIGINAL METHOD
    ## CALCULATE SPECIFIC STREAM POWER
    calculation["OGLUSSP2"] = calculation.apply(
        lambda row: row["OGLUSP2"] / row["RurWidth"], axis=1 #USING PHILLIPS-DESLOGES WIDTH
    )

    # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON)
    pos_val = calculation["RuralSSP"] > 0
    if pos_val.any():
        calculation["OGLUSSPR2"] = calculation.apply(
            lambda row: row["OGLUSSP2"] / row["RuralSSP"] if row["RuralSSP"] != 0 else 0,
            axis=1,
        )
    else:
        calculation["OGLUSSPR2"] = 0
## ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    ## REVISED NOSWM DISCHARGE EQUATION
    a = rQ_a
    b = rQ_b
    c = uQ_c
    d = uQ_d

    calculation["E5LUQ2m3_s"] = 0
    calculation["E5LUSP2"] = 0

    calculation["E5LUQ2m3_s"] = pd.to_numeric(calculation["E5LUQ2m3_s"], errors="coerce")
    calculation["E5LUSP2"] = pd.to_numeric(calculation["E5LUSP2"], errors="coerce")

    calculation["E5LUQ2m3_s"] = calculation.apply(
        lambda row: a * (((row["DA_km2"])*(1-row["Imp"])) ** b) +  c * (((row["DA_km2"])*(row["Imp"])) ** d), axis=1
    )

    # TOTAL POWER
    calculation["E5LUSP2"] = calculation.apply(
        lambda row: row["E5LUQ2m3_s"] * row["SlopeAVGLU"] * row["Wg_Nperm3"], axis=1
    )

    # SPECIFIC STREAM POWER
    ### NEWWIDTHURBAN-CALIBRATION
    # ENSURE THAT THE DIFFERENCE IS NON-NEGATIVE
    positive_diff = np.maximum(0, (calculation["E5LUQ2m3_s"] - calculation["Q_m3pers"])/calculation["Q_m3pers"])

    # CALCULATE E5URBWIDTH USING THE NON-NEGATIVE DIFFERENCE
    calculation["E5UrbWidth"] = calculation["RurWidth"]*(1 + urb_width*(positive_diff**0.50))

    # CONVERT COLUMNS TO NUMERIC TYPE
    calculation["E5UrbWidth"] = pd.to_numeric(calculation["E5UrbWidth"], errors="coerce")

    # DISPLAY THE UPDATED DATAFRAME
    print(calculation)

    pos_val4 = calculation["E5UrbWidth"] > 0

    if pos_val4.any():
        calculation["E5ULUSSP2"] = calculation.apply(
            lambda row: row["E5LUSP2"] / row["E5UrbWidth"] if row["E5UrbWidth"] != 0 else 0,
            axis=1,
        )
    else:
        calculation["E5ULUSSP2"] = 0

    # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON)
    pos_val5 = calculation["RuralSSP"] > 0
    if pos_val5.any():
        calculation["E5ULUSSPR2"] = calculation.apply(
            lambda row: row["E5ULUSSP2"] / row["RuralSSP"]
            if row["RuralSSP"] != 0
            else 0,
            axis=1,
        )
    else:
        calculation["E5ULUSSPR2"] = 0

    output_file = os.path.join(
        watershed_output, "Final_Land_Use_Stream_Power_Output.shp"
    )

    calculation.to_file(output_file, driver="ESRI Shapefile")
## ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
## ENGINEERED SWM OPTION
## ------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    if engswmanalysis == "Y":
        print("Running Analysis for Including Engineered SWM")
        engineeredswm_folder = os.path.join(intermediate_storage_files,"Engineered_SWM")
```

```
    # ENSURE THE OUTPUT FOLDER EXISTS, OR CREATE IT IF IT DOESN'T
    if not os.path.exists(engineeredswm_folder):
        os.makedirs(engineeredswm_folder)

    # PATH FOR THE COPY OF DEM
    swm_dem_path = os.path.join(engineeredswm_folder, "DEM_Copy_Lakes.tif")
    wbt.flatten_lakes(dem_clipped_path,engineeredsswm,swm_dem_path)
    print("DEM adjusted for ponds successfully.")

    # GET REVISED FLOW DIRECTION (D8 METHOD)
    swm_flow_dir = os.path.join(engineeredswm_folder, "flow_dir_swm.tif")
    wbt.d8_pointer(swm_dem_path, swm_flow_dir)
    print("Flow direction adjustment completed.")

    # GET FLOW ACCUMULATION
    swm_flow_accum = os.path.join(engineeredswm_folder, "flow_accum_swm.tif")
    wbt.d8_flow_accumulation(swm_dem_path, swm_flow_accum, out_type="cells")
    print("Flow accumulation calculation completed.")
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## EFFICIENCY RASTER - ENGINEERED SWM PONDS OPTION 1
## PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY WHERE THERE ARE NO PONDS, AND 0 FOR WHERE THERE ARE PONDS
## THE USER NEEDS TO PUT A VALUE IN FOR IMPERVIOUSNESS THAT THEY WANT
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    # GETTING THE OUTLET FOR EACH OF THE PONDS AND WRITING AN EFFICIENCY RASTER
    # OPEN THE DEM RASTER TO GET ITS DIMENSIONS AND TRANSFORM
    with rasterio.open(swm_dem_path) as src_dem:
        height = src_dem.height
        width = src_dem.width
        transform = src_dem.transform

    # CREATE A NEW EFFICIENCY RASTER FILE
    engefficiency_raster_file = os.path.join(engineeredswm_folder, "EngineeringEfficiencyRaster.tif")

    # OPEN THE FLOW ACCUMULATION RASTER TO GET ITS DIMENSIONS AND TRANSFORMATION
    with rasterio.open(flow_accum) as src:
        width = src.width
        height = src.height
        transform = src.transform
        crs = src.crs

    # OPEN THE FLOW DIRECTION RASTER TO GET ITS DIMENSIONS AND TRANSFORMATION
    with rasterio.open(swm_flow_dir) as src:  # ASSUMING `FLOW_DIR` IS THE PATH TO THE FLOW DIRECTION RASTER FILE
        flow_dir_data = src.read(1)  # READ THE RASTER DATA
        transform = src.transform
        crs = src.crs
    # CREATE AN EMPTY ARRAY TO HOLD THE RASTERIZED POLYGONS
    efficiency_data = np.ones((height, width), dtype=np.float32)
    # CREATE AN EMPTY LIST TO STORE THE MAXIMUM FLOW ACCUMULATION WITHIN EACH POLYGON
    max_flow_accumulations = []

    # OPEN THE SHAPEFILE TO GET ITS CRS
    with fiona.open(engineeredsswm, "r") as shapefile:
        crs = shapefile.crs

        for feature in shapefile:
            geometry = feature["geometry"]
            rasterized_polygon = rasterize(
                [geometry],
                out_shape=(height, width),
                transform=transform,
                fill=0,  # SET TO 0 TO INDICATE THAT THE PIXEL FALLS WITHIN A POLYGON (POND AREA)
                dtype=np.uint8
            )
            efficiency_data[rasterized_polygon == 0] = 0  # SET THE CORRESPONDING PIXELS TO 0 (POND AREA)

    # SET ALL REMAINING PIXELS (NON-POND AREAS) TO 1
    efficiency_data[efficiency_data != 0] = 1
    # SAVE THE EFFICIENCY RASTER TO A GEOTIFF FILE
    with rasterio.open(
        engefficiency_raster_file,
        "w",
        driver="GTiff",
        width=width,
        height=height,
        count=1,
        dtype=np.float32,
        crs=crs,
        transform=transform,
    ) as dst:
        dst.write(efficiency_data, 1)

    print(f"Engineering efficiency raster saved to {engefficiency_raster_file}")

    # DEFINE THE FUNCTION TO IDENTIFY CONTRIBUTING CELLS
    def identify_contributing_cells(flow_accum_data, outlet_cell):
        contributing_cells = set()
        visited_cells = set()
        queue = [outlet_cell]
        drainage_area = 0  # INITIALIZE DRAINAGE AREA

        while queue:
            x, y = queue.pop(0)
            if (x, y) not in visited_cells:
                visited_cells.add((x, y))
                contributing_cells.add((x, y))

                # INCREMENT DRAINAGE AREA BY 1 FOR EACH CONTRIBUTING CELL
                drainage_area += 1

                # GET THE FLOW ACCUMULATION VALUE OF THE CURRENT CELL
                x, y = int(x), int(y)  # ENSURE X AND Y ARE INTEGERS
                flow_accumulation = flow_accum_data[y, x]

                # FIND NEIGHBORING CELLS WITH HIGHER FLOW ACCUMULATION
                neighbors = get_higher_accumulation_neighbors(x, y, flow_accum_data, visited_cells, flow_accumulation)

                # ADD NEIGHBORING CELLS TO THE QUEUE
                queue.extend(neighbors)
        print(f"Outlet Cell: {outlet_cell}, Drainage Area: {drainage_area}")  # ADD THIS LINE TO PRINT THE OUTLET CELL AND DRAINAGE AREA

        return contributing_cells, drainage_area  # RETURN BOTH CONTRIBUTING CELLS AND DRAINAGE AREA

    # DEFINE THE FUNCTION TO GET NEIGHBORING CELLS WITH HIGHER FLOW ACCUMULATION
    def get_higher_accumulation_neighbors(x, y, flow_accum_data, visited_cells, current_accumulation):
        neighbors = []

        # DEFINE THE NEIGHBORHOOD WINDOW (E.G., 3X3)
        window_size = 3
        half_window = window_size // 2

        # ITERATE OVER THE NEIGHBORHOOD
        for j in range(y - half_window, y + half_window + 1):
            for i in range(x - half_window, x + half_window + 1):
                # CHECK IF THE CELL IS WITHIN THE RASTER BOUNDARIES
                if 0 <= i < flow_accum_data.shape[1] and 0 <= j < flow_accum_data.shape[0]:
                    # CHECK IF THE CELL IS NOT IN VISITED_CELLS
                    if (i, j) not in visited_cells:
                        # CHECK IF THE CELL HAS HIGHER FLOW ACCUMULATION
                        if flow_accum_data[j, i] > current_accumulation:
                            neighbors.append((i, j))

        return neighbors

    # DEFINE THE FUNCTION TO CALCULATE DRAINAGE AREA
    def calculate_drainage_area(contributing_cells, cell_size):
        # ASSUMING EACH CELL CONTRIBUTES A UNIT AREA
        return len(contributing_cells) * cell_size**2  # ASSUMING CELL_SIZE IS IN METERS
```

```
    ### POND OUTLETS
    # OPEN THE FLOW ACCUMULATION RASTER TO READ ITS DATA AND TRANSFORM
    with rasterio.open(flow_accum) as flow_src:
        flow_data = flow_src.read(1)
        flow_transform = flow_src.transform

    # OPEN THE OUTPUT RASTER CONTAINING POND BOUNDARIES TO READ ITS DATA AND TRANSFORM
    with rasterio.open(engefficiency_raster_file) as output_src:
        output_data = output_src.read(1)
        output_transform = output_src.transform

    # OPEN THE SHAPEFILE TO GET ITS CRS
    with fiona.open(engineeredsswm, "r") as shapefile:
        crs = shapefile.crs

    # ITERATE OVER EACH POLYGON IN THE SHAPEFILE
    with fiona.open(engineeredsswm, "r") as shapefile:
        for feature in shapefile:
            # EXTRACT THE GEOMETRY OF THE CURRENT POLYGON
            geometry = feature["geometry"]

            # MASK THE FLOW ACCUMULATION RASTER WITH THE POLYGON BOUNDARY
            mask = geometry_mask([geometry], flow_src.shape, flow_transform, invert=True)
            masked_flow = np.where(mask, flow_data, np.nan)

            # FIND THE MAXIMUM FLOW ACCUMULATION AND ITS LOCATION WITHIN THE POLYGON BOUNDARY
            max_index = np.nanargmax(masked_flow)
            max_flow = np.nanmax(masked_flow)
            max_coordinates = np.unravel_index(max_index, masked_flow.shape)

            # APPEND THE MAXIMUM FLOW ACCUMULATION AND ITS LOCATION TO THE LIST
            max_flow_accumulations.append({
                'max_flow': max_flow,
                'max_coordinates': max_coordinates
            })

    # PRINT THE MAXIMUM FLOW ACCUMULATION AND ITS LOCATION WITHIN EACH POLYGON BOUNDARY THIS IS THE
INFLOW LOCATION
    for i, max_flow_info in enumerate(max_flow_accumulations):
        print(f"Polygon {i+1}: Maximum flow accumulation = {max_flow_info['max_flow']} at location {max_flow_info['max_coordinates']}")

    # CREATE A NEW EFFICIENCY RASTER WITH CELLS CONTAINING THE MAXIMUM FLOW ACCUMULATIONS SET TO 0
AND ALL OTHER CELLS SET TO 1
    efficiency_data = np.ones_like(output_data, dtype=np.uint8)
    for max_flow_info in max_flow_accumulations:
        max_coordinates = max_flow_info['max_coordinates']
        efficiency_data[max_coordinates] = 0

    # WRITE THE EFFICIENCY RASTER TO A NEW GEOTIFF FILE
    maxcellefficiency_raster_file = os.path.join(engineeredswm_folder, "MaxCellEngineeringEfficiencyRaster.tif")
    with rasterio.open(
        maxcellefficiency_raster_file,
        "w",
        driver="GTiff",
        width=output_src.width,
        height=output_src.height,
        count=1,
        dtype=np.uint8,
        crs=output_src.crs,
        transform=output_src.transform,
    ) as dst:
        dst.write(efficiency_data, 1)

    print(f"New efficiency raster saved to {maxcellefficiency_raster_file}")

    # OPEN THE EFFICIENCY RASTER
    with rasterio.open(maxcellefficiency_raster_file) as src:
        # READ THE RASTER DATA
        efficiency_data = src.read(1)

    # GET THE AFFINE TRANSFORMATION MATRIX
    transform = src.transform

    # GET THE CELL SIZE
    cell_size = transform.a

    # FIND THE COORDINATES OF CELLS WITH A VALUE OF 0
    zero_indices = np.where(efficiency_data == 0)
    zero_coords = [src.xy(row, col) for row, col in zip(zero_indices[0], zero_indices[1])]

    # CONVERT COORDINATES TO POINT GEOMETRIES
    points = [Point(x, y) for x, y in zero_coords]

    # CREATE A GEODATAFRAME WITH THE POINT GEOMETRIES
    pour_points_gdf1 = gpd.GeoDataFrame(geometry=points, crs=src.crs)

    # SAVE THE POUR POINTS AS A SHAPEFILE
    pour_points_file1 = os.path.join(engineeredswm_folder,"PondOutlets.shp")
    pour_points_gdf1.to_file(pour_points_file1)

    print(f"Pour points saved to {pour_points_file1}")

    ## CHANGE THE FLOW DIRECTION RASTER
    # READ THE SHAPEFILE
    gdf = gpd.read_file(engineeredsswm)

    # EXTRACT POINTS FROM EACH POLYGON
    extracted_points = []
    for idx, row in gdf.iterrows():
        polygon = row['geometry']
        boundary = polygon.boundary
        line = LineString(boundary)
        points = [(coord[0], coord[1]) for coord in line.coords]
        extracted_points.extend(points)

    # CONVERT EXTRACTED POINTS TO A GEODATAFRAME
    geometry = [Point(x, y) for x, y in extracted_points]
    crs = gdf.crs  # GET THE COORDINATE REFERENCE SYSTEM FROM THE ORIGINAL GEODATAFRAME
    points_gdf = gpd.GeoDataFrame(geometry=geometry, crs=crs)

    # SAVE THE GEODATAFRAME TO A NEW SHAPEFILE
    pondpoints = os.path.join(engineeredswm_folder, "pondboundarypoints.shp")
    points_gdf.to_file(pondpoints)
    print("Pond points extracted")

    da_output_folder = os.path.join(engineeredswm_folder,"Pond_Drainage_Areas")
    # ENSURE THE OUTPUT FOLDER EXISTS, OR CREATE IT IF IT DOESN'T
    if not os.path.exists(da_output_folder):
        os.makedirs(da_output_folder)

    print("Drainage area raster extraction beginning.")

    output_raster = os.path.join(da_output_folder, 'outletdrainagearea.tif')
    wbt.watershed(flow_dir, pour_points_file1, output_raster)
    output_shapefile = os.path.join(da_output_folder, 'outletdrainagearea.shp')
    wbt.raster_to_vector_polygons(output_raster, output_shapefile)

    # OPEN THE INPUT SHAPEFILE
    driver = ogr.GetDriverByName('ESRI Shapefile')
    dataset = driver.Open(output_shapefile, 1)  # OPEN IN EDIT MODE
    layer = dataset.GetLayer()

    target_srs = osr.SpatialReference()
    target_srs.SetFromUserInput(str(dem_espg2int))

    # SET THE CRS FOR THE DRAINAGE AREA SHAPEFILE
    # ADD A NEW FIELD FOR DRAINAGE AREA
    field_defn = ogr.FieldDefn('SWMDA', ogr.OFTReal)
    layer.CreateField(field_defn)
```

```
# GET THE RASTER GEOTRANSFORM
raster_dataset = gdal.Open(output_raster)
gt = raster_dataset.GetGeoTransform()

# LOOP THROUGH EACH FEATURE IN THE SHAPEFILE
for feature in layer:
    # GET THE GEOMETRY OF THE FEATURE
    geometry = feature.GetGeometryRef()

    # CALCULATE THE AREA OF THE FEATURE'S GEOMETRY
    area = geometry.GetArea()

    # CONVERT THE DRAINAGE AREA FROM SQUARE METERS TO SQUARE KILOMETERS
    drainage_area_sqkm = area / 1e6  # 1 SQUARE KILOMETER = 1,000,000 SQUARE METERS

    # SET THE DRAINAGE AREA ATTRIBUTE FOR THE FEATURE
    feature.SetField('SWMDA', drainage_area_sqkm)

    # SAVE THE CHANGES
    layer.SetFeature(feature)

# CLOSE THE DATASET
dataset = None

print("Drainage area extraction and recording completed.")

# OPEN THE POUR POINTS SHAPEFILE
pour_points_dataset = driver.Open(pour_points_file1, 1)  # OPEN IN EDIT MODE
pour_points_layer = pour_points_dataset.GetLayer()

# OPEN THE DRAINAGE AREA SHAPEFILE
drainage_area_dataset = driver.Open(output_shapefile, 0)  # OPEN IN READ-ONLY MODE
drainage_area_layer = drainage_area_dataset.GetLayer()

# GET THE FIELD INDEX FOR SWMDA IN THE DRAINAGE AREA LAYER
swmda_field_index = drainage_area_layer.GetLayerDefn().GetFieldIndex('SWMDA')

# ADD A NEW FIELD FOR DRAINAGE AREA IN THE POUR POINTS SHAPEFILE
pour_points_field_defn = ogr.FieldDefn('SWMDA', ogr.OFTReal)
pour_points_layer.CreateField(pour_points_field_defn)

# LOOP THROUGH EACH POUR POINT FEATURE
for pour_point_feature in pour_points_layer:
    # GET THE GEOMETRY OF THE POUR POINT
    pour_point_geometry = pour_point_feature.GetGeometryRef()

    # RESET SWMDA VALUE TO 0 BY DEFAULT
    pour_point_feature.SetField('SWMDA', 0)

    # LOOP THROUGH EACH DRAINAGE AREA FEATURE
    drainage_area_layer.ResetReading()
    for drainage_area_feature in drainage_area_layer:
        # GET THE GEOMETRY OF THE DRAINAGE AREA FEATURE
        drainage_area_geometry = drainage_area_feature.GetGeometryRef()

        # CHECK IF THE POUR POINT INTERSECTS WITH THE DRAINAGE AREA FEATURE
        if pour_point_geometry.Intersects(drainage_area_geometry):
            # IF THERE IS AN INTERSECTION, ASSIGN THE DRAINAGE AREA VALUE TO THE POUR POINT
            drainage_area_value = drainage_area_feature.GetField(swmda_field_index)
            pour_point_feature.SetField('SWMDA', drainage_area_value)
            break  # EXIT THE LOOP ONCE THE DRAINAGE AREA VALUE IS ASSIGNED

    # SAVE THE CHANGES TO THE POUR POINT FEATURE
    pour_points_layer.SetFeature(pour_point_feature)

# CLOSE THE DATASETS
pour_points_dataset = None
drainage_area_dataset = None

print("Drainage area extraction to pour points shapefile completed.")
```

```
    engineereddmassoutput = os.path.join(engineeredswm_folder,"EngSWMD8MassFluxOutput.tif")

    # # COPY THE ENG SWM EFFICIENCY
    # OPEN INPUT RASTER FILE
    input_raster4 = gdal.Open(maxcellefficiency_raster_file, gdal.GA_Update)

    reprojectedfiles = os.path.join(intermediate_storage_files, "WhiteBoxToolFiles")
    if not os.path.exists(reprojectedfiles):
        os.makedirs(reprojectedfiles)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputswmeffreproj = os.path.join(reprojectedfiles, "Copy_EngSWMEfficiency.tif")
    output_raster4 = driver.CreateCopy(outputswmeffreproj, input_raster4)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster4 = None
    output_raster4 = None

    wbt_engswmfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Copy_EngSWMEfficiency") and filename.endswith(
            ".tif"
        ):
            wbt_engswmfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_engswmfile is not None:
        print("Update file found:", wbt_engswmfile)
    else:
        wbt_engswmfile = outputswmeffreproj
        print("Using the old file:", wbt_engswmfile)

    output_impervious1 = os.path.join(engineeredswm_folder, "Final_Eng_Imperviousness.tif")

    # IF A IMPERVIOUS RASTER IS PROVIDED, THESE STEPS WILL BE EXECUTED
    if lurasteranalysis == "Y":
        print("Raster of LU Imperviousness is provided")
            # ENSURE INTERMEDIATE STORAGE FOLDER EXISTS
        if not os.path.exists(engineeredswm_folder):
            os.makedirs(engineeredswm_folder)

        # DEFINE THE NEW LOCATION TO SAVE THE RASTER
        impraster = os.path.join(engineeredswm_folder, "LU_Imperviousness_raster.tif")

        # OPEN THE ORIGINAL RASTER FILE
        with rasterio.open(lurasteranalysispath) as src:
            raster_data = src.read(1)  # READ THE FIRST BAND OF THE RASTER
            profile = src.profile  # GET THE METADATA FOR THE RASTER
            # NORMALIZE IMPERVIOUSNESS VALUES (IF VALUES ARE IN PERCENTAGE FORM)
            raster_data = raster_data / 100.0
            raster_data = np.clip(raster_data, 0, 1)  # CLAMP VALUES BETWEEN 0 AND 1

        # SAVE THE RASTER TO THE OUTPUT PATH WITHOUT MODIFICATION
        with rasterio.open(impraster, 'w', **profile) as dst:
            dst.write(raster_data, 1)  # WRITE THE ORIGINAL RASTER DATA TO THE FIRST BAND

        print(f"Raster saved successfully to {impraster}")

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## PREPROCESS THE INPUT RASTERS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # RESOLUTION EXTRACTED AT TOP
        with rasterio.open(dem_clipped_path) as src:
            # GET THE AFFINE TRANSFORM FROM THE METADATA
            transform = src.transform

            # THE RESOLUTION IS STORED IN THE CELLSIZE OF THE AFFINE TRANSFORM
            x_resolution = transform.a
            y_resolution = transform.e
```

```
        # THE CELLSIZE IS THE ABSOLUTE VALUE OF THE RESOLUTION, AS IT MAY BE NEGATIVE IF THE RASTER IS
ROTATED
        x_cellsize = abs(x_resolution)
        y_cellsize = abs(y_resolution)

    crs = rasterio.crs.CRS.from_epsg(int(dem_espg2int))

    # ABSORPTION RASTER
    # PREPROCESS THE ABSORPTION RASTER, WE WANT TO HAVE ZERO ABSORPTION
    # OPEN THE RASTER FILE
    with rasterio.open(dem_clipped_path) as src:
        # READ THE DATA INTO A NUMPY ARRAY
        data = src.read(1)

        # SET ALL CELLS TO THE DESIRED VALUE
        data[:] = 0

        # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
        meta = src.meta.copy()
        meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

        absorption_file = os.path.join(engineeredswm_folder, "Absorption.tif")
        # WRITE THE UPDATED ARRAY TO A NEW FILE
        with rasterio.open(absorption_file, "w", **meta) as dst:
            dst.write(data, 1)

    # EFFICIENCY RASTER
    # PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY
    with rasterio.open(dem_clipped_path) as src:
        # READ THE DATA INTO A NUMPY ARRAY
        data = src.read(1)

        # SET ALL CELLS TO THE DESIRED VALUE
        data[:] = 1

        # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
        meta = src.meta.copy()
        meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

        efficiency_file = os.path.join(engineeredswm_folder, "Efficiency.tif")
        # WRITE THE UPDATED ARRAY TO A NEW FILE
        with rasterio.open(efficiency_file, "w", **meta) as dst:
            dst.write(data, 1)

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CROP THE LAND USE TIF'S
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # SHAPEFILE FILE
    ras_lu = impraster

    # READ SHAPEFILE
    gdf = gpd.read_file(basin_polygon_sub_path)

    gdf["count"] = [i + 1 for i, _ in enumerate(gdf.geometry)]
    outputfile = os.path.join(engineeredswm_folder, "basin_selected_count.shp")
    gdf.to_file(outputfile)

    gdf1 = gpd.read_file(outputfile)

    # GDF.CRS
    names = [x for x in gdf1["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

    # READ SHAPEFILE
    gdf3 = gpd.read_file(outputfile)

    # GDF.CRS
    names = [x for x in gdf3["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

    # READ RASTER DATA
    ras_data = rasterio.open(ras_lu)

    # OUTPUT PATH
    out_raster_lu = os.path.join(
        engineeredswm_folder, "LURasterIntermediateSteps"
    )
    if not os.path.exists(out_raster_lu):
        os.makedirs(out_raster_lu)

    # SAVE OUTPUT FILES AS PER SHAPEFILE FEATURES
    for i in range(len(gdf3)):
        geom = []
        coord = shapely.geometry.mapping(gdf3)["features"][i]["geometry"]
        geom.append(coord)
        with rasterio.open(ras_lu) as src:
            out_image, out_transform = rasterio.mask.mask(src, geom, crop=True)
            out_meta = src.meta
            out_meta.update(
                {
                    "driver": "GTiff",
                    "height": out_image.shape[1],
                    "width": out_image.shape[2],
                    "transform": out_transform,
                }
            )
            # FILE_PATH = ''
            with rasterio.open(
                f"{out_raster_lu}/{names[i]}.tif", "w", **out_meta
            ) as dest:
                dest.write(out_image)

    luraster_files = glob.glob(out_raster_lu + "/*.tif")
    mergedfilelu = os.path.join(engineeredswm_folder, "Final_LU_Raster.tif")

    # OPEN INPUT RASTERS
    src_files_to_mosaic2 = []
    # PRINT(SRC_FILES_TO_MOSAIC2)

    for file in luraster_files:
        src = rasterio.open(file)
        src_files_to_mosaic2.append(src)

    # MERGE INPUT RASTERS INTO ONE
    mosaic, out_trans = rasterio.merge.merge(src_files_to_mosaic2)

    # CLOSE ALL INPUT RASTER FILES
    for src in src_files_to_mosaic2:
        src.close()

    # UPDATE METADATA FOR OUTPUT RASTER
    out_meta = src.meta.copy()
    out_meta.update(
        {
            "driver": "GTiff",
            "height": mosaic.shape[1],
            "width": mosaic.shape[2],
            "transform": out_trans,
        }
    )

    # WRITE OUTPUT RASTER TO DISK
    with rasterio.open(mergedfilelu, "w", **out_meta) as dest:
        dest.write(mosaic)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## REPROJECT THE FILES
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    reprojectedfiles = os.path.join(engineeredswm_folder, "WhiteBoxToolFiles")
    if not os.path.exists(reprojectedfiles):
        os.makedirs(reprojectedfiles)

    # DEFINE EPSG CODE
```

```
        epsg_code = dem_espg2int

        # CREATE SPATIALREFERENCE OBJECT FROM EPSG CODE
        srs = osr.SpatialReference()
        srs.ImportFromEPSG(int(epsg_code))

        # CONVERT SPATIALREFERENCE OBJECT TO WKT STRING
        new_crs_wkt = srs.ExportToWkt()

        # COPY THE DEMS
        # OPEN INPUT RASTER FILE
        input_raster = gdal.Open(dem_clipped_path, gdal.GA_Update)

        # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL
SIZE
        existing_geotransform = input_raster.GetGeoTransform()
        new_geotransform = (
            existing_geotransform[0],
            existing_geotransform[1],
            0,
            existing_geotransform[3],
            0,
            existing_geotransform[5],
        )

        # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
        input_raster.SetGeoTransform(new_geotransform)
        input_raster.SetProjection(new_crs_wkt)

        # CREATE OUTPUT RASTER FILE
        driver = gdal.GetDriverByName("GTiff")
        outputdemreproj = os.path.join(reprojectedfiles, "Reprojected_DEM.tif")
        output_raster = driver.CreateCopy(outputdemreproj, input_raster)

        # CLOSE INPUT AND OUTPUT RASTER FILES
        input_raster = None
        output_raster = None

        # COPY THE ABSORPTION
        # OPEN INPUT RASTER FILE
        input_raster1 = gdal.Open(absorption_file, gdal.GA_Update)

        driver = gdal.GetDriverByName("GTiff")
        outputabsreproj = os.path.join(reprojectedfiles, "Copy_Absorption.tif")
        output_raster1 = driver.CreateCopy(outputabsreproj, input_raster1)

        # CLOSE INPUT AND OUTPUT RASTER FILES
        input_raster1 = None
        output_raster1 = None

        # COPY THE EFFICIENCY
        # OPEN INPUT RASTER FILE
        input_raster2 = gdal.Open(efficiency_file, gdal.GA_Update)

        # CREATE OUTPUT RASTER FILE
        driver = gdal.GetDriverByName("GTiff")
        outputeffreproj = os.path.join(reprojectedfiles, "Copy_Efficiency.tif")
        output_raster2 = driver.CreateCopy(outputeffreproj, input_raster2)

        # CLOSE INPUT AND OUTPUT RASTER FILES
        input_raster2 = None
        output_raster2 = None

        # COPY THE LAND USE
        # OPEN INPUT RASTER FILE
        input_raster3 = gdal.Open(mergedfilelu, gdal.GA_Update)

        # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL
SIZE
        existing_geotransform3 = input_raster3.GetGeoTransform()
        new_geotransform3 = (
            existing_geotransform3[0],
            existing_geotransform3[1],
            0,
            existing_geotransform3[3],
            0,
            existing_geotransform3[5],
        )

        # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
        input_raster3.SetGeoTransform(new_geotransform3)
        input_raster3.SetProjection(new_crs_wkt)

        # CREATE OUTPUT RASTER FILE
        driver = gdal.GetDriverByName("GTiff")
        outputlureproj = os.path.join(reprojectedfiles, "Reprojected_LandUse.tif")
        output_raster3 = driver.CreateCopy(outputlureproj, input_raster3)

        # CLOSE INPUT AND OUTPUT RASTER FILES
        input_raster3 = None
        output_raster3 = None

        # COPY THE FLOWACCUM
        # OPEN INPUT RASTER FILE
        input_raster4 = gdal.Open(flow_accum, gdal.GA_Update)

        # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL
SIZE
        existing_geotransform4 = input_raster4.GetGeoTransform()
        new_geotransform4 = (
            existing_geotransform4[0],
            existing_geotransform4[1],
            0,
            existing_geotransform4[3],
            0,
            existing_geotransform4[5],
        )

        # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
        input_raster4.SetGeoTransform(new_geotransform4)
        input_raster4.SetProjection(new_crs_wkt)
        # CREATE OUTPUT RASTER FILE
        driver = gdal.GetDriverByName("GTiff")
        outputflowaccumreproj = os.path.join(reprojectedfiles, "Reprojected_FlowAccum.tif")
        output_raster4 = driver.CreateCopy(outputflowaccumreproj, input_raster4)

        # CLOSE INPUT AND OUTPUT RASTER FILES
        input_raster4 = None
        output_raster4 = None
    ## ------------------------------------------------------------------------------------------------------------------------------------------------------
    ## CHECK SPATIAL EXTENTS OF THE FILES
    ## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # OPEN DEM RASTER
        small_raster = gdal.Open(outputdemreproj)
        small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

        # DICTIONARY TO STORE PROCESSED RASTERS
        raster_dict = {}

        # LOOP THROUGH ALL GEOTIFF FILES IN THE FOLDER
        for file_name in os.listdir(reprojectedfiles):
            if file_name.endswith(".tif"):
                file_path = os.path.join(reprojectedfiles, file_name)

                # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
                raster = gdal.Open(file_path, gdal.GA_Update)
                raster_gt = raster.GetGeoTransform()
                raster_proj = raster.GetProjection()
                raster_size = (raster.RasterXSize, raster.RasterYSize)
```

```
            # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
            if (
                raster_gt == small_raster.GetGeoTransform()
                and raster_proj == small_raster.GetProjection()
                and raster_size == small_size
            ):
                print(
                    f"{file_name} is already in the correct CRS, resolution, and size."
                )
            else:
                # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
                raster_update_path = os.path.join(
                    reprojectedfiles, f"{file_name.split('.')[0]}_update.tif"
                )
                new_raster = gdal.Translate(
                    raster_update_path,
                    raster,
                    format="GTiff",
                    width=small_size[0],
                    height=small_size[1],
                )
                new_raster.SetGeoTransform(small_raster.GetGeoTransform())
                new_raster.SetProjection(small_raster.GetProjection())
                print(f"{file_name} has been updated.")

            # CLOSE THE RASTER FILE
            raster = None
## ----------------------------------------------------------------------------------------------------------------------------------------------------------
## LOOP THROUGH THE FOLDER AND CHECK IF THERE ARE UPDATE FILES THEN ASSIGN IT VARIABLES
## IF THERE IS NO CHANGE IN CRS/RES/EXTENTS USE THE OLD TIF
## ----------------------------------------------------------------------------------------------------------------------------------------------------------
        wbt_demfile = None
        for filename in os.listdir(reprojectedfiles):
            if filename.startswith("Reprojected_DEM_update") and filename.endswith(".tif"):
                wbt_demfile = os.path.join(reprojectedfiles, filename)
                break
        if wbt_demfile is not None:
            print("Update file found:", wbt_demfile)
        else:
            wbt_demfile = outputdemreproj
            print("Using the old file:", wbt_demfile)

        wbt_absfile = None
        for filename in os.listdir(reprojectedfiles):
            if filename.startswith("Reprojected_Absorption_update") and filename.endswith(
                ".tif"
            ):
                wbt_absfile = os.path.join(reprojectedfiles, filename)
                break
        if wbt_absfile is not None:
            print("Update file found:", wbt_absfile)
        else:
            wbt_absfile = outputabsreproj
            print("Using the old file:", wbt_absfile)

        wbt_efffile = None
        for filename in os.listdir(reprojectedfiles):
            if filename.startswith("Reprojected_Efficiency_update") and filename.endswith(
                ".tif"
            ):
                wbt_efffile = os.path.join(reprojectedfiles, filename)
                break
        if wbt_efffile is not None:
            print("Update file found:", wbt_efffile)
        else:
            wbt_efffile = outputeffreproj
            print("Using the old file:", wbt_efffile)
        wbt_lufile = None
        for filename in os.listdir(reprojectedfiles):
            if filename.startswith("Reprojected_LandUse_update") and filename.endswith(
                ".tif"
            ):
                wbt_lufile = os.path.join(reprojectedfiles, filename)
                break
        if wbt_lufile is not None:
            print("Update file found:", wbt_lufile)
        else:
            wbt_lufile = outputlureproj
            print("Using the old file:", wbt_lufile)

        dividing_flowaccum = None
        for filename in os.listdir(reprojectedfiles):
            if filename.startswith("Reprojected_FlowAccum_update") and filename.endswith(
                ".tif"
            ):
                dividing_flowaccum = os.path.join(reprojectedfiles, filename)
                break
        if dividing_flowaccum is not None:
            print("Update file found:", dividing_flowaccum)
        else:
            dividing_flowaccum = outputflowaccumreproj
            print("Using the old file:", dividing_flowaccum)
## ----------------------------------------------------------------------------------------------------------------------------------------------------------
## USE WBT TO OUTPUT THE D8 MASSFLUX
## ----------------------------------------------------------------------------------------------------------------------------------------------------------
        # BRING IN WHITEBOX TOOLS
        wbt = whitebox.WhiteboxTools()

        # DEFINE THE OUTPUT_FILE
        dmassoutput = os.path.join(engineeredswm_folder, "D8MassFluxOutput.tif")

        # USE D8MASSFLUX TOOL AND OUTPUT A FILE
        wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_efffile, wbt_absfile, dmassoutput)
        #WBT.D8_MASS_FLUX(DEM_CLIPPED_PATH, MERGEDFILELU, EFFICIENCY_FILE, ABSORPTION_FILE, DMASSOUTPUT)
## ----------------------------------------------------------------------------------------------------------------------------------------------------------
## SAVE IN LOG SCALE
## ----------------------------------------------------------------------------------------------------------------------------------------------------------
        with rasterio.open(dmassoutput) as src:
            # READ THE RASTER DATA INTO A NUMPY ARRAY
            data = src.read(1)
            # APPLY LOG TRANSFORMATION
            data_log = np.log(data + 1)

            # CREATE NEW RASTER FILE
            profile = src.profile  # META DATA COPY
            profile.update(nodata=np.nan)

            LogScale_D8 = os.path.join(
                engineeredswm_folder, "LogScale_D8MassFlux.tif"
            )
            with rasterio.open(LogScale_D8, "w", **profile) as dst:
                dst.write(data_log, 1)


        # OPTION 1 D8 MASS FLUX FOR THE PONDS WITH EFFICIENCY OF ZERO
        wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_engswmfile, wbt_absfile, engineereddmassoutput) # THIS ONE SETS THE OUTLET TO
ZERO

        #WBT.D8_MASS_FLUX(WBT_DEMFILE, WBT_LUFILE, WBT_EFFFILE, WBT_ABSFILE, ENGINEEREDDMASSOUTPUT) #
THIS ONE DOES NOT CHANGE THE EFFICIENCY

        # DIVIDING!
        # OPEN THE TWO RASTER FILES
        src5 = gdal.Open(engineereddmassoutput)
        src4 = gdal.Open(flow_accum) # DIVIDING BY THE NEW FLOW ACCUMULATION RASTER GENERATED THAT TAKES INTO
ACCOUNT THE PONDS

        # THE FLOW ACCUMUALTION RASTER TAKES INTO ACCOUNT THE PONTS, SO IT IS NOT NECESSARY TO DEFINE THE
RELATIONSHIP BETWEEN
        # THE CONTRIBUTING DRAINAGE AREA AND THE TOTAL DRAINAGE AREA.
```

```
# GET THE SIZE OF THE RASTERS
cols = src5.RasterXSize
rows = src5.RasterYSize

# CREATE AN OUTPUT RASTER WITH THE SAME SIZE AND PROJECTION AS THE INPUT RASTERS
driver = gdal.GetDriverByName("GTiff")
out_raster1 = driver.Create(output_impervious1, cols, rows, 1, gdal.GDT_Float32)
out_raster1.SetProjection(src5.GetProjection())
out_raster1.SetGeoTransform(src5.GetGeoTransform())

# READ THE DATA FROM BOTH RASTERS INTO NUMPY ARRAYS
data3 = src5.ReadAsArray()
data2 = src4.ReadAsArray()

# PERFORM ELEMENT-WISE DIVISION OF THE TWO ARRAYS
result1 = np.divide(data3, data2)

# SET THE RANGE OF VALUES TO WRITE TO THE OUTPUT RASTER
data_min = 0
data_max = 1

# CLIP THE RESULT ARRAY TO THE SPECIFIED RANGE
result_clipped1 = np.clip(result1, data_min, data_max)

# WRITE THE RESULT TO THE OUTPUT RASTER
out_raster1.GetRasterBand(1).WriteArray(result_clipped1)

# CLOSE THE RASTERS
src5 = None
src4 = None
out_raster1 = None

print("Divided Rasters Engineered SWM, Imperviousness Found")

with rasterio.open(output_impervious1) as src:
    imp = src.read(1)

# PLOT THE RASTER
plt.figure(figsize=(10, 10))
rasterio.plot.show(imp, transform=src.transform, cmap="terrain")
plt.show()

# OPEN DEM RASTER
small_raster = gdal.Open(dem_clipped_path)
small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

# OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
clippedengimperviousraster = gdal.Open(output_impervious1, gdal.GA_Update)
rasterimp_gt1 = clippedengimperviousraster.GetGeoTransform()
rasterimp_proj1 = clippedengimperviousraster.GetProjection()
rasterimp_size1 = (
    clippedengimperviousraster.RasterXSize,
    clippedengimperviousraster.RasterYSize,
)

# CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
if (
    rasterimp_gt1 == small_raster.GetGeoTransform()
    and rasterimp_proj1 == small_raster.GetProjection()
    and rasterimp_size1 == small_size
):
    print("Imperviousness is already in the correct CRS, resolution, and size.")
else:
    # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
    raster_imp_update1 = os.path.join(
        engineeredswm_folder, "Final_Eng_Imperviousness_update.tif"
    )
    new_raster2 = gdal.Translate(
        raster_imp_update1,
        clippedengimperviousraster,
        format="GTiff",
        width=small_size[0],
        height=small_size[1],
    )
    new_raster2.SetGeoTransform(small_raster.GetGeoTransform())
    new_raster2.SetProjection(small_raster.GetProjection())
    print("Imperviousness has been updated.")

# CLOSE THE RASTER FILES
raster = None

divided_imp1 = None
for filename in os.listdir(engineeredswm_folder):
    if filename.startswith("Final_Eng_Imperviousness_update") and filename.endswith(
        ".tif"
    ):
        divided_imp1 = os.path.join(engineeredswm_folder, filename)
        break
if divided_imp1 is not None:
    print("Update file found:", divided_imp1)
else:
    divided_imp1 = output_impervious1
    print("Using the old file:", divided_imp1)

extract_pts1 = os.path.join(
    engineeredswm_folder, "Urban_Stream_Points_EngSWM_Imperviousness.shp"
)

# READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
gdf = gpd.read_file(extract_pts)

gdf["SWMImp"] = np.nan
gdf["SWMD8Flux"] = np.nan
gdf["SWMLUFloAc"] = np.nan

# OPEN THE RASTER FILE FOR READING
with rasterio.open(divided_imp1) as src4, rasterio.open(
    engineereddmassoutput
) as src5, rasterio.open(flow_accum) as src6:
    # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
    for index, row in gdf.iterrows():
        # GET THE X AND Y COORDINATES OF THE POINT
        point1 = row.geometry
        x, y = point1.x, point1.y

        # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
        engimpvalue = list(src4.sample([(x, y)]))
        endmassfluxvalue = list(src5.sample([(x, y)]))
        flowaccumvalue = list(src6.sample([(x, y)]))

        # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
        gdf.loc[index, "SWMImp"] = engimpvalue[0][0]
        gdf.loc[index, "SWMD8Flux"] = endmassfluxvalue[0][0]
        gdf.loc[index, "SWMLUFloAc"] = flowaccumvalue[0][0]

# WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
gdf.to_file(extract_pts1)

print("Impervious Values for Eng SWM Scenario Extracted to Match Stream Points")

# NOW MERGE THE VALUES BASED ON THE COLUMN
points_gdf1 = gpd.read_file(extract_pts1)
lines_gdf3 = gpd.read_file(copy_strm_net)

engurban_segments = os.path.join(
    engineeredswm_folder, "Urban_Eng_SWM_Imperviousness_Attachment.shp"
)

lines_gdf3.to_file(engurban_segments)
```

```
lines_gdf4 = gpd.read_file(engurban_segments)

# POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
# THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
common_column = "pXY"

# MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
merged_gdf1 = lines_gdf2.set_index(common_column).join(
    points_gdf1.set_index(common_column)[["SWMImp", "SWMD8Flux", "SWMLUFloAc"]]
)
print(merged_gdf1.columns)

# ATTACH THE DRAINAGE AREAS FROM THE OUTLETS TO THE STREAM SEGMENTS
# DEFINE THE PROXIMITY THRESHOLD
proximity_threshold = 10  # THIS IS THE THRESHOLD TO SEARCH FOR WHERE THE POUR POINTS ARE

# ADD A NEW FIELD FOR DRAINAGE AREA IN SQUARE KILOMETERS
merged_gdf1['SWMDA'] = None

snappedpourpoints = os.path.join(engineeredswm_folder,"SnappedPourPoints.shp")
# STREAMS = OS.PATH.JOIN(WATERSHED_OUTPUT,"EXTRACT_STREAM.TIF")
# WBT.JENSON_SNAP_POUR_POINTS(POUR_POINTS_FILE1, STREAMS, SNAPPEDPOURPOINTS, PROXIMITY_THRESHOLD)

# READ THE POUR POINTS SHAPEFILE
pour_points = gpd.read_file(pour_points_file1)

# INITIALIZE THE CUMULATIVE DRAINAGE AREA COLUMN IF IT DOES NOT EXIST
if 'SWMDA' not in merged_gdf1.columns:
    merged_gdf1['SWMDA'] = 0

# INITIALIZE CUMULATIVE DRAINAGE AREA VALUE
cumulative_swmda = 0

# INITIALIZE A DICTIONARY TO TRACK SEGMENTS WITH POUR POINTS
pour_point_segments = {}

# CREATE A LIST TO STORE THE SNAPPED POUR POINT LOCATIONS
snapped_locations = []

# SNAP POUR POINTS TO THE NEAREST STREAM SEGMENT AND UPDATE POUR_POINT_SEGMENTS DICTIONARY
for idx, pour_point in pour_points.iterrows():
    pour_point_geometry = pour_point.geometry
    distances = merged_gdf1.geometry.apply(lambda line: line.distance(pour_point_geometry))
    nearest_stream_idx = distances.idxmin()

    # CALCULATE THE SNAPPED POINT LOCATION ON THE NEAREST STREAM SEGMENT
    nearest_stream_geom = merged_gdf1.loc[nearest_stream_idx].geometry
    snapped_location = nearest_stream_geom.interpolate(nearest_stream_geom.project(pour_point_geometry))
    snapped_locations.append({'geometry': snapped_location, 'original_index': idx, 'nearest_stream_index': nearest_stream_idx})

    if nearest_stream_idx in pour_point_segments:
        pour_point_segments[nearest_stream_idx] += pour_point['SWMDA']
    else:
        pour_point_segments[nearest_stream_idx] = pour_point['SWMDA']

# ITERATE OVER EACH SEGMENT IN THE STREAM NETWORK
for index, segment_row in merged_gdf1.iterrows():
    # CHECK IF THE CURRENT SEGMENT HAS A POUR POINT
    if index in pour_point_segments:
        cumulative_swmda += pour_point_segments[index]
        #print(f"Segment {index}: Cumulative SWMDA updated to {cumulative_swmda}")
    else:
        print(f"Segment {index}: Cumulative SWMDA carried forward as {cumulative_swmda}")

    # ASSIGN THE CUMULATIVE DRAINAGE AREA VALUE TO THE SEGMENT
    merged_gdf1.at[index, 'SWMDA'] = cumulative_swmda

# CREATE A GEODATAFRAME FOR THE SNAPPED POUR POINT LOCATIONS
snapped_locations_gdf = gpd.GeoDataFrame(snapped_locations, crs=merged_gdf1.crs)

    # SAVE THE SNAPPED POUR POINT LOCATIONS TO A NEW SHAPEFILE
    snapped_locations_gdf.to_file(snappedpourpoints)

    # PRINT A MESSAGE INDICATING COMPLETION
    print("Drainage Area Values Extracted and Assigned to engurban_segments")
    print("Snapped pour points saved")

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf1.to_file(engurban_segments)

    print("Impervious, D8 Mass Flux, Flow Accumulation Values Appended to Stream Segments")
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE THE DISCHARGE 2 YR
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    calculation1 = gpd.read_file(engurban_segments)

    # CREATES NEW COLUMN AND SETS DEFAULT VALUE TO 0
    calculation1["SWMLUQ2m3_s"] = 0
    calculation1["SWMLUSP2"] = 0
    calculation1["Wg_Nperm3"] = 9810
    calculation1["SWMLUSSP2"] = 0

    calculation1["SWMLUQ2m3_s"] = pd.to_numeric(calculation1["SWMLUQ2m3_s"], errors="coerce")
    calculation1["SlopeAVGLU"] = pd.to_numeric(calculation1["SlopeAVGLU"], errors="coerce")
    calculation1["Wg_Nperm3"] = pd.to_numeric(calculation1["Wg_Nperm3"], errors="coerce")
    calculation1["RurWidth"] = pd.to_numeric(calculation1["RurWidth"], errors="coerce")
    calculation1["RuralSSP"] = pd.to_numeric(calculation1["RuralSSP"], errors="coerce")

    # ENGINEERED SWM DISCHARGE EQUATION
    # FORMULA FOR DISCHARGE IS Q=CA^X(IA^B) WHERE C, X AND B ARE COEFFICCIENTS, A IS THE DRAINAGE AREA
AND IA IS TOTAL IMPERVIOUSNESS PERCENTAGE
    # A IS IN THE COLUMN TITLED DA_KM2
    # IA IS IN COLUMN TITLED IMP

    a = rQ_a
    b = rQ_b

    c = u_Qs_c
    f = u_Qs_d

    # FUNCTION TO APPLY THE CALCULATION WITH DEBUGGING
    def calculate_flow(row):
        # EXTRACT VALUES
        DA_km2 = row["DA_km2"]
        SWMImp = row["SWMImp"]

        # CHECK FOR MISSING VALUES
        if pd.isna(DA_km2) or pd.isna(SWMImp):
            print("Missing values detected, skipping row.")
            return None  # OR SOME DEFAULT VALUE

        # PRINT ROW VALUES
        #print(f"Processing row: DA_km2={DA_km2}, SWMImp={SWMImp}")

        # CALCULATE INTERMEDIATE RESULTS
        pervious_contribution = a * (DA_km2 * (1 - SWMImp)) ** b
        impervious_contribution = c * (DA_km2 * SWMImp) ** f

        # PRINT INTERMEDIATE RESULTS
        #print(f"Pervious contribution: {pervious_contribution}")
        #print(f"Impervious contribution: {impervious_contribution}")

        # FINAL CALCULATION
        result = pervious_contribution + impervious_contribution
        #print(f"Result: {result}")

        return result

    # APPLY THE CALCULATION
    calculation1["SWMLUQ2m3_s"] = calculation1.apply(calculate_flow, axis=1)
```

```
    ## CALCULATE TOTAL POWER
    calculation1["SWMLUSP2"] = calculation1.apply(
        lambda row: (row["SWMLUQ2m3_s"]) * (row["SlopeAVGLU"]) * (row["Wg_Nperm3"]), axis=1
    )

    # ORIGINAL METHOD
    ## CALCULATE SPECIFIC STREAM POWER
    calculation1["SWMLUSSP2"] = calculation1.apply(
        lambda row: row["SWMLUSP2"] / row["RurWidth"], axis=1 #USING PHILLIPS-DESLOGES WIDTH
    )

    # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON)
    pos_val = calculation1["RuralSSP"] > 0
    if pos_val.any():
        calculation1["SWMLUSSPR2"] = calculation1.apply(
            lambda row: row["SWMLUSSP2"] / row["RuralSSP"] if row["RuralSSP"] != 0 else 0,
            axis=1,
        )
    else:
        calculation1["SWMLUSSPR2"] = 0

    # SPECIFIC STREAM POWER
    ### NEWWIDTHURBAN-CALIBRATION
    # ENSURE THAT THE DIFFERENCE IS NON-NEGATIVE
    positive_diff1 = np.maximum(0, (calculation1["SWMLUQ2m3_s"] - calculation1["Q_m3pers"])/calculation1["Q_m3pers"])

    # CALCULATE E5URBWIDTH USING THE NON-NEGATIVE DIFFERENCE
    calculation1["SWMUrbWidth"] = calculation1["RurWidth"]*(1 + urb_width*(positive_diff1**0.50))

    # CONVERT COLUMNS TO NUMERIC TYPE
    calculation1["SWMUrbWidth"] = pd.to_numeric(calculation1["SWMUrbWidth"], errors="coerce")

    # DISPLAY THE UPDATED DATAFRAME
    print(calculation1)

    pos_val6 = calculation1["SWMUrbWidth"] > 0

    if pos_val6.any():
        calculation1["SWMULSSP2"] = calculation1.apply(
            lambda row: row["SWMLUSP2"] / row["SWMUrbWidth"] if row["SWMUrbWidth"] != 0 else 0,
            axis=1,
        )
    else:
        calculation1["SWMULSSP2"] = 0

    # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON) ASHLUSSP2/RURSSPASH
    pos_val7 = calculation1["RuralSSP"] > 0
    if pos_val7.any():
        calculation1["SWMULSPR2"] = calculation1.apply(
            lambda row: row["SWMULSSP2"] / row["RuralSSP"]
            if row["RuralSSP"] != 0
            else 0,
            axis=1,
        )
    else:
        calculation1["SWMULSPR2"] = 0

    output_file1 = os.path.join(
        watershed_output, "Final_Land_Use_EngSWM_Stream_Power_Output.shp"
    )
    calculation1.to_file(output_file1, driver="ESRI Shapefile")

print(counter)

# DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
file_pattern = os.path.join(watershed_output, 'stream_points_fields_renamed.*')
file_list = glob.glob(file_pattern)
for file_path in file_list:
    os.remove(file_path)

lu_analysis_file = None

## END OF SINGLE LU ANALYSIS SCRIPT
end1 = time.time()
print("The time of execution of Land Use Single is :", (end1 - start1), "seconds")
gc.collect()
###############################################################################################################################
###############################################
###############################################################################################################################
###############################################
## SINGLE WATERSHED AND HECRAS SELECTED
start2 = time.time()

if check_boundary == 0 and hec_analysis == "Y":
    counter == 0

if counter == 0 and hec_analysis == "Y":
    counter += 1
    print("SINGLE WATERSHED, THIS WILL ONLY LOOP ONCE")

    for watershed in os.listdir(hec_watershed):
        watershed_path = os.path.join(hec_watershed, watershed)
        watershed_output = os.path.join(data_dir, "HECRAS Analysis")
        if os.path.exists(watershed_output):
            print("FOLDER CREATED")
        else:
            intermediate_dir = os.makedirs(watershed_output)

        watershed_outputs_id = os.path.join(watershed_output, watershed)
        if os.path.exists(watershed_outputs_id):
            print("FOLDER CREATED")
        else:
            intermediate_dir = os.makedirs(watershed_outputs_id)

        wbt.set_working_dir(watershed_outputs_id)

        if os.path.isdir(
            watershed_path
        ):  # IF THE SUBFOLDER PATH EXISTS THEN PERFORM THE SCRIPT
## ---------------------------------------------------------------------------------------------------------------------------------
## REDEFINE THE VARIABLES
## ---------------------------------------------------------------------------------------------------------------------------------
            hec_csv = os.path.join(
                watershed_path, "hec_csv.csv"
            )  # HECRAS STATIONS DATA
            hec_point_raw = os.path.join(
                watershed_path, "hec_shp.shp"
            )  # HECRAS AS POINTS SHAPEFILE

            bridge_hec_csv = os.path.join(
                watershed_path, "bridge_hec_csv.csv"
            )  # HECRAS BRIDGE DATA
            bridge_point_raw = os.path.join(
                watershed_path, "bridge_point_raw.shp"
            )  # HECRAS BRIDGE DATA AS SHAPEFILES

            culv_hec_csv = os.path.join(
                watershed_path, "culv_hec_csv.csv"
            )  # CULVERT DATA
            culv_point_raw = os.path.join(
                watershed_path, "culv_point_raw.shp"
            )  # CULVERT DATA AS A SHAPEFILE

            # OPEN THE DEM RASTER FILE
            with rasterio.open(dem) as src:
                # GET THE CRS OF THE RASTER FILE
                crs = src.crs
```

```
    # USE PYPROJ TO DETECT THE EPSG CODE OF THE CRS
    dem_espg2int = pyproj.CRS.from_string(crs.to_string()).to_epsg()

    print(f"EPSG code: {dem_espg2int}")

# READ THE HEC-RAS POINTS
hecpoint_src = gpd.read_file(hec_point_raw)

# GET THE CRS OF THE HEC-RAS POINTS
hecpoint_src_prj = hecpoint_src.crs

# GET THE EPSG CODE OF THE CRS
hecpoint_src_prj_epsg_int = hecpoint_src_prj.to_epsg()
print(f"EPSG code: {hecpoint_src_prj_epsg_int}")

if dem_espg2int != hecpoint_src_prj_epsg_int:
    print("Reprojecting river stations")
    hecpoint_src_copy = hecpoint_src.to_crs(epsg=dem_espg2int)

    hecpoint_prj_path = os.path.join(watershed_outputs_id,"hecpoint_prj.shp")
    hecpoint_src_copy.to_file(hecpoint_prj_path, driver="ESRI Shapefile")
    hec_point = hecpoint_prj_path

    # SET THE OUTPUT CRS FOR THE FINAL RESULTS
    output_crs = dem_espg2int
else:
    print("Will not reproject river stations")
    hec_point = hec_point_raw

    # SET THE OUTPUT CRS FOR THE FINAL RESULTS
    output_crs = dem_espg2int

# BRIDGE ANALYSIS - REPROJECT POINT FEATURES
try:
    if bridge_analysis == "Y":
        bridgept_src = geopandas.read_file(bridge_point_raw)
        bridgept_src_prj = bridgept_src.crs

        # ELSE:
        print("Will not reproject bridge stations")
        bridge_point = bridge_point_raw
except:
    print("Bridge analysis fails", file=sys.stderr)

# CULVERT ANALYSIS
try:
    if culvert_analysis == "Y":
        culvpt_src = geopandas.read_file(culv_point_raw)
        culvpt_src_prj = culvpt_src.crs

        # ELSE:
        print("Will not reproject culvert stations")
        culv_point = culv_point_raw
except:
    print("Culvert analysis fails", file=sys.stderr)

# FOR SUBWATERSHED ANALYSIS ONLY
if subwatershed_analysis == "Y":
    subwshed_src = geopandas.read_file(subwshed_raw)
    subwshed_src_prj = subwshed_src.crs
    subwshed_src_prj_epsg_str = subwshed_src_prj["init"].split(":")[1]
    subwshed_src_prj_epsg_int = int(subwshed_src_prj_epsg_str)

    if dem_obj_prj_espg_str != subwshed_src_prj_epsg_str:
        print("Reprojecting subwatershed")
        # PYEPSG.GET(DEM_OBJ_PRJ_ESPG_STR)
        subwshed_src_copy = subwshed_src.to_crs(epsg=dem_espg2int)
        subwshed_prj_path = str(data_dir) + str("/subwatershed_prj.shp")
        subwshed_src_copy.to_file(
            subwshed_prj_path, driver="ESRI Shapefile"
        )

        subwshed_poly = subwshed_prj_path

    else:
        print("Will not reproject subwatershed")
        subwshed_poly = subwshed_raw

print("HEC-RAS STARTING FOR SINGLE WATERSHED")
if hec_analysis == "Y" or bridge_analysis == "Y" or culvert_analysis == "Y":
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CLIP DEM TO BASIN
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # MAKE COPY OF HEC POINTS
    wbt.set_working_dir(watershed_outputs_id)
    wbt.verbose = False

    if not os.path.exists(watershed_outputs_id):
        os.makedirs(watershed_outputs_id)

    hec_analysis_file = os.path.join(watershed_outputs_id, "HECRAS_Analysis.gpkg")

    hec_point_forbasin = geopandas.read_file(hec_point)
    basin_hecpts_path = os.path.join(watershed_outputs_id, "basin_hecpts.shp")
    hec_point_forbasin = hec_point_forbasin.loc[
        :, hec_point_forbasin.columns.isin(["REACH","RIVER_STA", "geometry"])
    ]
    hec_point_forbasin.to_file(basin_hecpts_path)
    layer_name = "basin_hecpts"
    hec_point_forbasin.to_file(hec_analysis_file, layer=layer_name, driver="GPKG")

    basin_raster_path = os.path.join(data_dir, "basins.tif")
    # EXTRACT ALL RASTER VALUES TO STREAM POINTS
    print("Extracting raster values")
    wbt.verbose = True
    wbt.extract_raster_values_at_points(basin_raster_path, basin_hecpts_path)
    wbt.verbose = False

    # OPEN HECRAS POINTS FILE
    basin_hecpts_df = geopandas.read_file(basin_hecpts_path)
    before_last = len(basin_hecpts_df.columns) - 2
    name_map = {basin_hecpts_df.columns[before_last]: "BasinValue"}
    basin_hecpts_df.rename(columns=name_map, inplace=True)

    # GET MOST FREQUEST BASIN VALUE
    basin_valuemode = basin_hecpts_df["BasinValue"].mode()
    basin_valuemode_int = basin_valuemode[0]
    print("basin_value_mod_int: ", basin_valuemode_int)

    # CONVERT BASIN TO POLYGON
    basin_polygon_path = os.path.join(
        watershed_outputs_id, "basins_polygon.shp"
    )

    mask = None
    with rasterio.Env():  # UPDATED 17/6/2019 CHANGED RASTERIO.DRIVERS() TO RASTERIO.ENV()
        with rasterio.open(basin_raster_path) as src:
            image = src.read(1)  # FIRST BAND
            results = (
                {"properties": {"Value": v}, "geometry": s}
                for i, (s, v) in enumerate(
                    shapes(image, mask=mask, transform=src.transform)
                )
            )  # UPDATED 17/6/2019 CHANGED SRC.AFFINE TO SRC.TRANSFORM
    geoms = list(results)
    polygonized_basin_df = geopandas.GeoDataFrame.from_features(geoms)
    polygonized_basin_df.to_file(basin_polygon_path, driver="ESRI Shapefile")

    layer_name = "basins_polygon"
    polygonized_basin_df.to_file(hec_analysis_file, layer=layer_name, driver="GPKG")
```

```
        # SELECT BASIN
        basin_polygon_sub_path = os.path.join(
            watershed_outputs_id, "basin_selected.shp"
        )
        polygonized_basin_sub_df = polygonized_basin_df.loc[
            polygonized_basin_df["Value"] == float(basin_valuemode_int)
        ]
        polygonized_basin_sub_df.crs = output_crs
        polygonized_basin_sub_df.to_file(
            basin_polygon_sub_path, driver="ESRI Shapefile"
        )

        layer_name = "basin_selected"
        polygonized_basin_sub_df.to_file(hec_analysis_file, layer=layer_name, driver="GPKG")

        # CLIP DEM
        dem_clipped_path = os.path.join(
            watershed_outputs_id, "breached_dem_clipped.tif"
        )
        overallbreached_dem = os.path.join(data_dir, "breached_dem.tif")

        with rasterio.open(overallbreached_dem) as src:
            out_image, out_transform = rasterio.mask.mask(src, polygonized_basin_sub_df.geometry.values, crop=True)
            out_meta = src.meta.copy()
            out_meta.update({
                "height": out_image.shape[1],
                "width": out_image.shape[2],
                "transform": out_transform
            })
            with rasterio.open(dem_clipped_path, "w", **out_meta) as dst:
                dst.write(out_image)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'basins_polygon.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## GENERAL WORKFLOW GET FLOW DIRECTION (D8 METHOD)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        gen_flow_dir = os.path.join(watershed_outputs_id, "flow_dir.tif")
        wbt.d8_pointer(dem_clipped_path, gen_flow_dir)

        # GET FLOW ACCUMULATION
        gen_flow_accum = os.path.join(watershed_outputs_id, "flow_accum.tif")
        wbt.d8_flow_accumulation(dem_clipped_path, gen_flow_accum, out_type="cells")

        # EXTRACT STREAMS BASED ON DRAINAGE THRESHOLD
        if pruningda == None:
            gen_extr_strm = os.path.join(watershed_outputs_id, "extract_stream_initial.tif")
            wbt.extract_streams(gen_flow_accum, gen_extr_strm, initial_dathres)
        else:
            gen_extr_strm = os.path.join(watershed_outputs_id, "extract_stream_usprune.tif")
            wbt.extract_streams(gen_flow_accum, gen_extr_strm, (pruningda/cell_size))

        # IF THE USER DOESNT GIVE A DRAINAGE AREA VALUE

        # GET STREAM VECTOR FROM RASTER STREAM
        gen_strm_vec = os.path.join(watershed_outputs_id, "stream_vector_initial.shp")
        wbt.raster_streams_to_vector(gen_extr_strm, gen_flow_dir, gen_strm_vec)

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_vector_initial'
        gdf = gpd.read_file(gen_strm_vec)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        initial_dem_points_path = os.path.join(watershed_outputs_id, "dem_points_initial.shp")
        wbt.raster_to_vector_points(gen_extr_strm, initial_dem_points_path)
        # SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
        initial_dem_points = geopandas.read_file(initial_dem_points_path)

        # WRITE TO A GEOPACKAGE
        print("Writing DEM Points Initial Output")
        layer_name = 'dem_points_initial'
        initial_dem_points.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        initial_stream_point_path = os.path.join(
            watershed_outputs_id, "stream_points_initial.shp"
        )
        initial_dem_points[initial_dem_points["VALUE"] == 1].to_file(
            initial_stream_point_path, driver="ESRI Shapefile"
        )
        # WRITE TO A GEOPACKAGE
        print("Writing Stream Points Initial Output")
        layer_name = 'stream_points_initial'
        gdf = gpd.read_file(initial_stream_point_path)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## STREAM PRUNING: REDEFINE STREAM THRES BASED ON MIN FLOW ACCUM OF HEC POINTS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        #IF INITIAL_DATHRES_PARAM == 1000:
        # EXTRACT FLOW ACCUMULATION TO STREAM POINTS
        wbt.extract_raster_values_at_points("flow_accum.tif", initial_stream_point_path)

        strm_point_faccum_gdf = geopandas.read_file(initial_stream_point_path)
        before_last = len(strm_point_faccum_gdf.columns) - 2
        name_map = {strm_point_faccum_gdf.columns[before_last]: "Faccum"}
        strm_point_faccum_gdf.rename(columns=name_map, inplace=True)

        # OPEN HEC RAS POINTS TO CREATE BUFFER AROUND THEM
        hec_pts_buffer_gdf = geopandas.read_file(basin_hecpts_path)
        hec_pts_buffer_gdf["geometry"] = hec_pts_buffer_gdf.geometry.buffer(50)

        hec_pts_buffer_gdf = hec_pts_buffer_gdf.rename(columns={"FID": "RIVER_STA"})

        buffers_hec_path = os.path.join(watershed_outputs_id, "hecpt_buffers50m.shp")
        hec_pts_buffer_gdf.to_file(buffers_hec_path, driver="ESRI Shapefile")
        print("Writing HECPT Buffers Output")

        # WRITE TO A GEOPACKAGE
        layer_name = 'hecpt_buffers50m'
        gdf = gpd.read_file(buffers_hec_path)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # JOIN POINTS TO BUFFER
        point_in_buffer = geopandas.sjoin(hec_pts_buffer_gdf,strm_point_faccum_gdf, how='left')

        # DISPLAY THE COLUMN NAMES
        column_names = point_in_buffer.columns.tolist()
        print("Column names:", column_names)

        # IF USER GIVES AN ACTUAL THRESHOLD THEY WANT, SKIP BRANCHIGNG AND REPLACE FACCUM_THRES_25PCNT
WITH THEIR VALUE

        ## BRANCHING
        # GROUP BY "RIVER_STA" AND CALCULATE THE MAXIMUM "FACCUM" AT EACH RIVER STATION
        point_in_buffer_max = point_in_buffer.groupby("RIVER_STA")["Faccum"].apply(lambda x: x.max())

        if pruningda == None:
            #PLOT FOR ALL OF THE POINTS NOT SEPERATED BY REACH ID
            point_in_buffer = point_in_buffer.dropna()
            point_in_buffer = point_in_buffer[point_in_buffer['Faccum'] != 0]

            # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA
            x_coordinates = point_in_buffer['RIVER_STA']
            flow_accumulation = point_in_buffer['Faccum']

            # PLOT THE X-COORDINATE VS. FLOW ACCUMULATION
            plt.scatter(x_coordinates, flow_accumulation)
```

```
# PERFORM LINEAR REGRESSION
slope, intercept, r_value, p_value, std_err = linregress(x_coordinates, flow_accumulation)
# ADD A REGRESSION LINE TO THE PLOT
plt.plot(x_coordinates, intercept + slope * x_coordinates, color='red', label='Regression Line')

print(intercept + slope * x_coordinates)
print("Intercept:", intercept)
print(x_coordinates, flow_accumulation)

# CUSTOMIZE THE PLOT
plt.xlabel('River Station')
plt.ylabel('Flow Accumulation')
#PLT.TEXT(0.1, 0.9, F'INTERCEPT: {INTERCEPT}', TRANSFORM=PLT.GCA().TRANSAXES, FONTSIZE=7)
plt.title(f'X-coordinate vs. Flow Accumulation for all Points')
plt.legend()

# SAVE THE PLOT AS AN IMAGE FILE
plot_filename = os.path.join(watershed_output,"HECAll_Reaches.tif")
plt.savefig(plot_filename, format="tif")

# SHOW OR SAVE THE PLOT
plt.show()

# # CREATE A LIST TO STORE DATA FOR LATER DATAFRAME CREATION
# RESULT_DATA = []

# CREATE AN EMPTY DATAFRAME TO STORE THE MAXIMUM FLOW ACCUMULATIONS
min_flow_table = pd.DataFrame(columns=['REACH', 'Min_Flow_Accumulation'])

# PLOTS FOR EACH INDIVIDUAL REACH ID
# EXTRACT UNIQUE REACH NUMBERS FROM YOUR DATA
unique_reach_numbers = point_in_buffer['REACH'].unique()

# PRINT THE UNIQUE REACH NUMBERS FOR REFERENCE
print("Unique REACH Numbers:", unique_reach_numbers)

# LOOP THROUGH EACH UNIQUE REACH NUMBER
for reach_number in unique_reach_numbers:
    # FILTER DATA BASED ON THE CURRENT REACH NUMBER
    filtered_data = point_in_buffer[point_in_buffer['REACH'] == reach_number]

    # FILTER OUT DATA POINTS WITH ZERO FLOW ACCUMULATION
    filtered_data = filtered_data[filtered_data['Faccum'] != 0]

    # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING
    if not filtered_data.empty:

        # CALCULATE THE MAXIMUM FLOW ACCUMULATION WITHIN THE FILTERED DATA FOR EACH RIVER_STA
        max_flow_per_station = filtered_data.groupby("RIVER_STA")["Faccum"].max().reset_index()

    # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING WITHIN EACH RIVER_STA GROUP
        if not max_flow_per_station.empty:
            # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA FROM MAX_FLOW_PER_STATION
            x_coordinates = max_flow_per_station['RIVER_STA'].values.reshape(-1, 1)
            flow_accumulation = max_flow_per_station['Faccum']

            # CREATE A NEW PLOT FOR EACH REACH NUMBER
            plt.figure()
            plt.scatter(x_coordinates, flow_accumulation)

            # CREATE A LINEAR REGRESSION MODEL
            regr = linear_model.LinearRegression()
            regr.fit(x_coordinates, flow_accumulation)

            # MAKE PREDICTIONS USING THE X FOR LINEAR LINE
            Y_pred = regr.predict(x_coordinates)

            # ASSIGN THE Y_PRED VALUES TO A NEW COLUMN 'Y_PRED' IN MAX_FLOW_PER_STATION
            max_flow_per_station['Y_pred'] = Y_pred

            # REPLACE NEGATIVE VALUES IN COLUMN 'Y_PRED' WITH VALUES FROM COLUMN 'FACCUM'
            mask = max_flow_per_station['Y_pred'] < 0
            max_flow_per_station.loc[mask, 'Y_pred'] = max_flow_per_station.loc[mask, 'Faccum']

            # PLOT FLOW ACCUMULATION AGAINST RIVER STATION
            plt.scatter(x_coordinates, flow_accumulation, color="black")
            # ADD A REGRESSION LINE TO THE PLOT
            plt.plot(x_coordinates, Y_pred, color='red', label='Regression Line')

            # CUSTOMIZE THE PLOT
            plt.xlabel('River Station')
            plt.ylabel('Flow Accumulation')
            plt.title(f'X-coordinate vs. Flow Accumulation for REACH {reach_number}')
            plt.legend()

            # SAVE THE PLOT AS AN IMAGE FILE
            plot_filename = os.path.join(watershed_output, f"hecreach_{reach_number}_plot.tif")
            plt.savefig(plot_filename, format="tif")

            # SHOW THE CURRENT PLOT
            plt.show()

        #MINIMUM FLOW ACCUMULATION OF THE REACH
        print(f'{reach_number}',max_flow_per_station)
        min_flow_accumulation = max_flow_per_station['Y_pred'].min()
        print(f'{reach_number}', min_flow_accumulation)

        # APPEND THE RESULT TO THE MAX_FLOW_TABLE DATAFRAME USING CONCAT
        min_flow_table = pd.concat([min_flow_table, pd.DataFrame({'REACH': [reach_number],'Min_Flow_Accumulation':
[min_flow_accumulation]})], ignore_index=True)

    # FIND THE MINIMUM FACCUM_MIN_YPRED
    min_faccum_min_ypred = min_flow_table["Min_Flow_Accumulation"].min()

    # CALCULATE FACCUM_THRES_25PCNT
    if min_faccum_min_ypred < 0:
        faccum_thres_25pcnt = initial_dathres_param
    else:
        faccum_thres_25pcnt = int(min_faccum_min_ypred)

    # PRINT THE RESULTS
    print("Result Table:")
    print(min_flow_table)
    # ASSUMING MIN_FLOW_TABLE IS A PANDAS DATAFRAME
    path = os.path.join(watershed_outputs_id,"min_flow_table.csv")
    min_flow_table.to_csv(path, index=False)
    print("Minimum faccum_min_ypred:", min_faccum_min_ypred)
    print("faccum_thres_25pcnt:", faccum_thres_25pcnt)

else:
    faccum_thres_25pcnt = round(pruningda/cell_size)
    print(faccum_thres_25pcnt)

# EXTRACT STREAM
wbt.extract_streams("flow_accum.tif", "extract_stream.tif", faccum_thres_25pcnt)
print("Streams have been extracted with a threshold applied")

# GET POINTS FROM RASTER STREAM BASED ON VALUE = 1
wbt.raster_to_vector_points("extract_stream.tif", "dem_points.shp")

# SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
dem_points_path = os.path.join(watershed_outputs_id,"dem_points.shp")
dem_points = geopandas.read_file(dem_points_path)

# WRITE TO A GEOPACKAGE
layer_name = 'dem_points'
gdf = gpd.read_file(gen_strm_vec)
gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

new_stream_point_path = os.path.join(watershed_outputs_id,"stream_points.shp")
```

```
        dem_points[dem_points["VALUE"] == 1].to_file(
            new_stream_point_path, driver="ESRI Shapefile"
        )
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_points'
        gdf = gpd.read_file(new_stream_point_path)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # GET STREAM VECTOR FROM RASTER STREAM
        wbt.raster_streams_to_vector(
            "extract_stream.tif", "flow_dir.tif", "stream_vector_pruned.shp"
        )

        # CLIP STREAM VECTOR TO SUBWATERSHED ANALYSIS
        if subwatershed_analysis == "Y":
            subwshed_df = geopandas.read_file(subwshed_poly)

            stream_vector_pruned_path = os.path.join(watershed_outputs_id,"stream_vector_pruned.shp")
            stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
            subwshed_stream_df = geopandas.sjoin(
                stream_vector_pruned_df, subwshed_df, op="intersects"
            )
            subwshed_stream_list = subwshed_stream_df["FID"].tolist()
            # SUBSET BY LIST OF LINES ONLY WITHIN SUBWATERSHED
            subwshed_stream_selected_df = stream_vector_pruned_df[
                stream_vector_pruned_df["FID"].isin(subwshed_stream_list)
            ]
            stream_vector_path = os.path.join(watershed_outputs_id,"stream_vector.shp")
            subwshed_stream_selected_df.to_file(stream_vector_path, driver="ESRI Shapefile")

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_vector'
            gdf = gpd.read_file(gen_strm_vec)
            gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')


        else:
            stream_vector_pruned_path = os.path.join(watershed_outputs_id,"stream_vector_pruned.shp")
            stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
            stream_vector_path = os.path.join(watershed_outputs_id,"stream_vector.shp")
            stream_vector_pruned_df.to_file(stream_vector_path, driver="ESRI Shapefile")

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_vector_pruned'
            gdf = gpd.read_file(stream_vector_pruned_path)
            gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'stream_vector_pruned.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
                os.remove(file_path)

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_vector'
            gdf = gpd.read_file(stream_vector_path)
            gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## EXTRACT ALL RASTER VALUES TO STREAM POINTS
## GET STREAM LINK CLASS
## CLASSIFICATION KEY
## VALUE  CLASS
## 1      EXTERIOR LINK
## 2      INTERIOR LINK
## 3      SOURCE NODE (HEADWATER)
## 4      LINK NODE (CONFLUENCE)
## 5      SINK NODE (OUTLET)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        wbt.stream_link_class(
            "flow_dir.tif", "extract_stream.tif", "stream_link_class.tif"
        )

        # GET UNIQUE ID FOR EACH STREAM LINK CLASS
        wbt.stream_link_identifier(
            "flow_dir.tif", "extract_stream.tif", "stream_link_id.tif"
        )

        # GET LENGTH FOR EACH STREAM LINK CLASS
        wbt.stream_link_length(
            "flow_dir.tif", "stream_link_id.tif", "stream_link_length.tif"
        )

        # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
        wbt.tributary_identifier(
            "flow_dir.tif", "extract_stream.tif", "tributary_id.tif"
        )

        # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
        wbt.strahler_stream_order(
            "flow_dir.tif", "extract_stream.tif", "strahler_order.tif"
        )

        # GET DISTANCE TO OUTLET
        wbt.distance_to_outlet(
            "flow_dir.tif", "extract_stream.tif", "distance_to_outlet.tif"
        )

        # GET THE DISTANCE TO THE FURTHEST UPSTREAM CHANNEL HEAD FOR EACH STREAM CELL.
        wbt.farthest_channel_head(
            "flow_dir.tif", "extract_stream.tif", "distance_to_headwater.tif"
        )

        # GET STREAM SLOPE ALONG RASTER STREAM
        # WBT.STREAM_SLOPE_CONTINUOUS("FLOW_DIR.TIF", "STREAM_LINK_ID.TIF", DEM , "STREAM_SLOPE.TIF")

        # EXTRACT ALL RASTER VALUES TO SELECTED STREAM POINTS
        wbt.extract_raster_values_at_points(

"distance_to_headwater.tif;distance_to_outlet.tif;stream_link_class.tif;stream_link_id.tif;stream_link_length.tif;tributary_id.tif;strahler_order.tif;flow_accum.tif",
            new_stream_point_path,
        )

        # EXTRACT RASTER DEM VALUES TO STREAM POINTS
        src_filename = dem
        shp_filename = new_stream_point_path

        src_ds = gdal.Open(src_filename)
        gt = src_ds.GetGeoTransform()
        rb = src_ds.GetRasterBand(1)

        ds = ogr.Open(shp_filename, 1)
        lyr = ds.GetLayer()
        lyr.CreateField(ogr.FieldDefn("Elev", ogr.OFTReal))
        for feat in lyr:
            geom = feat.GetGeometryRef()
            mx, my = geom.GetX(), geom.GetY()  # COORD IN MAP UNITS

            # CONVERT FROM MAP TO PIXEL COORDINATES.
            # ONLY WORKS FOR GEOTRANSFORMS WITH NO ROTATION.
            px = int((mx - gt[0]) / gt[1])  # X PIXEL
            py = int((my - gt[3]) / gt[5])  # Y PIXEL

            intval = rb.ReadAsArray(px, py, 1, 1)
            elev = intval[0]
            for value in elev:
                v = float(value)
                # PRINT(V)

            # UPDATE NEW FIELD
```

```
            lyr.SetFeature(feat)
            feat.SetField("Elev", v)
            lyr.SetFeature(feat)

        # RENAME FIELDS
        stream_point = geopandas.read_file(new_stream_point_path)

        name_map = {
            stream_point.columns[2]: "DistUp",
            stream_point.columns[3]: "DistDown",
            stream_point.columns[4]: "LinkClass",
            stream_point.columns[5]: "LinkID",
            stream_point.columns[6]: "LinkLength",
            stream_point.columns[7]: "TribID",
            stream_point.columns[8]: "Strahler",
            stream_point.columns[9]: "FlowAccum",
            stream_point.columns[10]: "Elev",
        }
        stream_point.rename(columns=name_map, inplace=True)

        # GET XY COORDINATES
        stream_point["X"] = stream_point.geometry.x
        stream_point["Y"] = stream_point.geometry.y
        stream_point["pXY"] = (
            stream_point["X"].map(str) + "," + stream_point["Y"].map(str)
        )

        # SAVE AS SHAPEFILE
        stream_point_field_rename_path = os.path.join(watershed_outputs_id,"stream_points_fields_renamed.shp")
        stream_point.to_file(stream_point_field_rename_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_points_fields_renamed'
        gdf = gpd.read_file(stream_point_field_rename_path)
        stream_point.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## STREAM SEGMENTATION
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        print("Stream segmentation beginning")

        stream_point = os.path.join(watershed_outputs_id,"stream_points.shp")
        stream_vector = os.path.join(watershed_outputs_id,"stream_vector.shp")

        # CREATE FUNCTION TO SPLIT LINE AT POINTS
        def split_line_at_points(line, point):
            gdf_line = line
            gdf_points = point

            # JOIN ALL LINE AND POINTS
            line = gdf_line.geometry.unary_union
            coords = gdf_points.geometry.unary_union

            # SNAP AND SPLIT COORDS ON LINE
            # RETURNS GEOMETRYCOLLECTION
            split_line = shapely.ops.split(line, coords)

            # CHECK IF SPLIT_LINE IS A GEOMETRYCOLLECTION
            if isinstance(split_line, shapely.geometry.collection.GeometryCollection):
                # ITERATE OVER INDIVIDUAL GEOMETRIES IN THE COLLECTION
                segments = [feature for feature in split_line.geoms]
            else:
                segments = [split_line]

            # TRANSFORM GEOMETRY COLLECTION TO GEODATAFRAME
            gdf_segments = geopandas.GeoDataFrame(
                list(range(len(segments))), geometry=segments
            )
            gdf_segments.columns = ["index", "geometry"]

            # SAVE SEGMENTS
            stream_segments = os.path.join(watershed_outputs_id,"stream_segments.shp")
            gdf_segments.to_file(stream_segments, driver="ESRI Shapefile")

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_segments'
            gdf = gpd.read_file(stream_segments)
            gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

            # RETURN GDF_SEGMENTS

        split_line_at_points(gpd.read_file(stream_vector), gpd.read_file(stream_point))
        # 3 MINS PROCESSING TIME
        print("segmentation finished")
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_vector'
        gdf = gpd.read_file(stream_vector)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_vector.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_points'
        gdf = gpd.read_file(stream_point)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## GET NODES OF SEGMENTS AND JOIN POINTS' ATTRIBUTES
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # GET LINE NODES (START AND END) USING WHITEBOX TO MATCH POINT NODES
        stream_segments = os.path.join(watershed_outputs_id,"stream_segments.shp")
        stream_segment_nodes = os.path.join(watershed_outputs_id,"stream_segment_nodes.shp")
        wbt.extract_nodes(stream_segments, stream_segment_nodes)

        # CREATE JOIN IDENTIFIER - FIELD "INDEX" (PLUS 1) OF STREAM SEGMENT IS FIELD "PARENTID" OF STREAM SEGMENT NODE.
        segment_attribute = ogr.Open(stream_segments, 1)
        segment_lyr = segment_attribute.GetLayer()
        segment_lyr.CreateField(ogr.FieldDefn("JoinID", ogr.OFTInteger))

        for feat in segment_lyr:
            # GET INDEX VALUE
            feat_v = feat.GetField("index")
            new_v = feat_v + 1
            # UPDATE NEW FIELD
            segment_lyr.SetFeature(feat)
            feat.SetField("JoinID", new_v)
            segment_lyr.SetFeature(feat)

        # JOIN STREAM POINTS DATA TO SEGMENTS' NODES
        stream_point_field_rename_df = geopandas.read_file(stream_point_field_rename_path)
        stream_segment_node_df = geopandas.read_file(stream_segment_nodes)

        stream_point_join2_segment_node = geopandas.sjoin(
            stream_segment_node_df,
            stream_point_field_rename_df,
            how="inner",
            op="intersects",
        )
        stream_segment_join2_point_path = os.path.join(watershed_outputs_id,"stream_point_join2_segment_node.shp"
        )
        stream_point_join2_segment_node.to_file(
            stream_segment_join2_point_path, driver="ESRI Shapefile"
        )

        # USE DISTANCE FROM OUTLET TO FIND THE COORDINATES OF START NODES AND END NODES OF SEGMENTS. SAVE THE END COORDINATES IN THE SAME ROW AS THE START COORDINATES.
        stream_segment_join2_point_df = geopandas.read_file(stream_segment_join2_point_path)
```

```
# WRITE TO A GEOPACKAGE
layer_name = 'stream_point_join2_segment_node'
stream_segment_join2_point_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

# DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
file_pattern = os.path.join(watershed_outputs_id, 'stream_point_join2_segment_node.*')
file_list = glob.glob(file_pattern)
for file_path in file_list:
    os.remove(file_path)

# WRITE TO A GEOPACKAGE
layer_name = 'stream_segment_nodes'
stream_segment_node_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

# DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
file_pattern = os.path.join(watershed_outputs_id, 'stream_segment_nodes.*')
file_list = glob.glob(file_pattern)
for file_path in file_list:
    os.remove(file_path)

stream_segment_join2_point_df["X"] = stream_segment_join2_point_df.geometry.x
stream_segment_join2_point_df["Y"] = stream_segment_join2_point_df.geometry.y
stream_segment_join2_point_df["StartXY"] = (
    stream_segment_join2_point_df["X"].map(str)
    + ","
    + stream_segment_join2_point_df["Y"].map(str)
)

# FIND MIN DISTANCE FROM OUTLET (I.E. LOWER NODE) AND UPDATE NEW COLUMN WITH ENDXY COORDINATES.
stream_segment_join2_point_df[
    "DistDown_min"
] = stream_segment_join2_point_df.groupby(["PARENT_ID"])["DistDown"].transform(min)

# SUBSET DATA BY EXTRACTING START NODES
stream_segment_join2_point_df_startnodes = stream_segment_join2_point_df[
    stream_segment_join2_point_df["DistDown_min"]
    != stream_segment_join2_point_df["DistDown"]
]
stream_segment_join2_point_df_endnodes = stream_segment_join2_point_df[
    stream_segment_join2_point_df["DistDown_min"]
    == stream_segment_join2_point_df["DistDown"]
]
stream_segment_join2_point_df_endnodes.loc[
    :, "EndXY"
] = stream_segment_join2_point_df_endnodes["StartXY"]

stream_segment_join2_point_df_subset = pd.merge(
    stream_segment_join2_point_df_startnodes,
    stream_segment_join2_point_df_endnodes[["PARENT_ID", "EndXY"]],
    how="left",
    on="PARENT_ID",
)

# MATCH THE ELEV VALUE FROM THE STREAM POINTS FOR BOTH NODES OF STREAM SEGMENTS
stream_segment_join2_point_df_subset[
    "EndElev"
] = stream_segment_join2_point_df_subset["EndXY"].map(
    stream_point_field_rename_df.set_index("pXY")["Elev"]
)
stream_segment_join2_point_df_subset[
    "StartElev"
] = stream_segment_join2_point_df_subset["StartXY"].map(
    stream_point_field_rename_df.set_index("pXY")["Elev"]
)

stream_segment_join2_point_df_subset_path = os.path.join(watershed_outputs_id,"stream_point_join2_segment_node_subset.shp")
stream_segment_join2_point_df_subset.to_file(
    stream_segment_join2_point_df_subset_path, driver="ESRI Shapefile"
)
```

```
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## JOIN STREAM SEGMENT NODES TO STREAM SEGMENT BASED ON PARENT ID
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        stream_segments_df = geopandas.read_file(stream_segments)
        stream_segment_join2_point_df_subset_df = geopandas.read_file(
            stream_segment_join2_point_df_subset_path
        )

        # MUST DROP GEOMETRY OF POINT TO ALLOW FILE TO BE SAVED IN THE FUTURE
        stream_segment_join2_point_df_subset_df = (
            stream_segment_join2_point_df_subset_df.drop("geometry", axis=1)
        )

        # STREAM_SEGMENTS_ATTRIBUTE_DF =
STREAM_SEGMENTS_DF.JOIN(STREAM_SEGMENT_JOIN2_POINT_DF_SUBSET.SET_INDEX('PARENT_ID'), ON='JOINID',
RSUFFIX = "_P")
        stream_segments_attribute_df = stream_segments_df.merge(
            stream_segment_join2_point_df_subset_df,
            left_on="JoinID",
            right_on="PARENT_ID",
            how="left",
        )

        # 1 MIN
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_point_join2_segment_node_subset'
        stream_segment_join2_point_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_point_join2_segment_node_subset.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments'
        gdf = gpd.read_file(stream_point)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE DEM SLOPE, DRAINAGE AREA, DISCHARGE AND WIDTH (RURAL SCENARIO)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # CALCULATE LENGTH OF SEGMENTS
        stream_segments_attribute_df["Length"] = stream_segments_attribute_df.length

        # CALCULATE SLOPE USING DEM ELEVATION
        stream_segments_attribute_df["Slope"] = (
            (
                stream_segments_attribute_df["EndElev"]
                - stream_segments_attribute_df["StartElev"]
            )
            / (stream_segments_attribute_df["Length"])
        ) * -1

        # CALCULATE DRAINAGE AREA (DA = (FLOW ACCUMULATION/1000000)*100
        stream_segments_attribute_df["DA_km2"] = (
            stream_segments_attribute_df["FlowAccum"] / 1000000
        ) * cell_size

        # CALCULATE RURAL DISCHARGE USING Q=CA^X WHERE C AND X ARE COEFFICCIENTS AND A IS THE DRAINAGE
AREA
        stream_segments_attribute_df["Q_m3pers"] = rQ_a * (
            stream_segments_attribute_df["DA_km2"] ** rQ_b
        )

        # CALCULATE WIDTH (EMPIRICAL - DA WIDTH)
        stream_segments_attribute_df["RurWidth"] = rW_a * (
            stream_segments_attribute_df["DA_km2"] ** rW_b
        )
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## 4) SMOOTH DEM SLOPE
## ------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
        # SET INDEX FROM HIGH ELEVATION TO LOW, I.E. BY DECREASING DISTANCE DOWNSTREAM
        stream_segments_attribute_df = stream_segments_attribute_df.sort_values(
            by=["DistDown"], ascending=False
        )
        stream_segments_attribute_df["SID"] = np.arange(
            len(stream_segments_attribute_df)
        )
        stream_segments_attribute_df.set_index("SID", inplace=True)
        stream_segments_attribute_df["index"] = np.arange(
            len(stream_segments_attribute_df)
        )

        # ADD NEW FIELDS TO STORE RESULTS
        stream_segments_attribute_df["deltaDA"] = None
        stream_segments_attribute_df["SavgID"] = None
        stream_segments_attribute_df["SavgRID"] = None
        stream_segments_attribute_df["SlopeAVG"] = None
## -----------------------------------------------------------------------------------------------------------------------------------------------
## CREATE SEGMENTS BLOCKS OVER WHICH TO SMOOTHE SLOPE
## -----------------------------------------------------------------------------------------------------------------------------------------------
        # CREATE LIST WITH ALL ELEVATION VALUES AND SORT IN DESCENDING ORDER. REMOVE ROWS WITHOUT
ATTRIBUTES
        stream_segments_attribute_df = stream_segments_attribute_df.dropna(
            axis=0, subset=["PARENT_ID"]
        )

        elev_set = set(stream_segments_attribute_df.apply(lambda row: (row["index"], row["EndElev"]), axis=1))

        elev_list_u = list(elev_set)
        elev_list = sorted(elev_list_u, key=lambda x: x[0])  # ACTIVATED

        # START FROM THE SEGMENT WITH HIGHEST ELEVATION, ASSIGN THE SAME SEGMENT ID WHILE TRACING
DOWNSTREAM OF THE REACH UNTIL THE DIFFERENCE RATIO IN DRAINAGE AREA IS >=0.1.
        seg_id = 0
        elev_list_count = elev_list

        while len(elev_list_count) != 0:
            elev_pair = elev_list_count[0]
            max_elev = elev_pair[1]
            max_id = elev_pair[0]
            seg_id = seg_id + 1
            del elev_list_count[0]

            # MAX_ELEV AND MAX_ID ONLY HAS ONE POSSIBLE MATCH, SKIP THE LOOP AND USE EXISTING INDEXES
            seg_r_id = 0
            start_endxy = stream_segments_attribute_df.loc[max_id, "EndXY"]
            start_da = stream_segments_attribute_df.loc[max_id, "DA_km2"]
            stream_segments_attribute_df.loc[max_id, "SavgID"] = seg_id
            stream_segments_attribute_df.loc[max_id, "SavgRID"] = seg_r_id

            if start_da is None or start_da == 0:
                start_da = 0.0

            # RATHER THAN LOOPING THROUGH MULTIPLE TIMES, QUERY THE DATAFRAME FOR THE MAIN CONDITION
            ssaIdxs = stream_segments_attribute_df.query(
                "@start_endxy == StartXY"
            ).index.tolist()
            # NOW LOOP THROUGH THE INITIAL MATCHES AND DO MATH
            while len(ssaIdxs) != 0:
                index = ssaIdxs[0]
                del ssaIdxs[0]
                r_da = stream_segments_attribute_df.loc[index, "DA_km2"]
                r_seg_id = stream_segments_attribute_df.loc[index, "SavgID"]
                if r_da == 0 or r_da == None:
                    delta_da = 0.0
                else:
                    diff_da = r_da - start_da
                    delta_da = (diff_da) / r_da
                if delta_da < thres_da and r_seg_id == None:
                    seg_r_id = seg_r_id + 1
                    stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
                    stream_segments_attribute_df.loc[index, "SavgRID"] = seg_r_id
                    stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
                    stream_segments_attribute_df.loc[index, "deltaDA"] = delta_da
                    start_endxy = stream_segments_attribute_df.loc[index, "EndXY"]
                    # FIND MATCHES FOR THE NEW START_ENDXY
                    ssaIdxs = stream_segments_attribute_df.query(
                        "@start_endxy == StartXY"
                    ).index.tolist()
                    # DELETE PAIR IN LIST IF EXIST
                    r_elev = stream_segments_attribute_df.loc[index, "EndElev"]
                    r_id = index
                    pair = (r_id, r_elev)
                    # PRINT PAIR
                    if pair in elev_list_count:
                        elev_list_count.remove(pair)
## -----------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE ROLLING MEAN OF SEGMENT USING 25 REACHES UPSTREAM AND 25 REACHES DOWNSTREAM IN A
SEGMENT
## -----------------------------------------------------------------------------------------------------------------------------------------------
        # OPTIMIZED CODE
        seg_id_l_unique = stream_segments_attribute_df["SavgID"].unique().tolist()

        # LOOP THROUGH TABLE AND COLLECT REACHIDS
        avg_window = {}

        for seg_id in seg_id_l_unique:
            # DONT LOOP AND CHECK CONDITIONS, GET INDEXES THAT MATCH CONDITION INSTEAD THEN LOOP
            ssaIdxs = stream_segments_attribute_df.query(
                "@seg_id == SavgID"
            ).index.tolist()
            for index in ssaIdxs:
                seg_r_id = stream_segments_attribute_df.loc[index, "SavgRID"]
                slope_raw = stream_segments_attribute_df.loc[index, "Slope"]
                avg_window[seg_r_id] = slope_raw

            avg_window_df = pd.Series(avg_window)
            roll_avg_df = avg_window_df.rolling(
                window=Savg_cell_window, min_periods=1, center=True
            ).mean()

            # LOOP THROUGH SET AND CALCULATE ROLLING MEAN
            for i, j in roll_avg_df.items():
                seg_r_id_avg = i
                avg_value = j
                ssaIdxs = stream_segments_attribute_df.query(
                    "(@seg_id == SavgID) & (@seg_r_id_avg == SavgRID)"
                ).index.tolist()
                for index in ssaIdxs:
                    stream_segments_attribute_df.loc[index, "SlopeAVG"] = abs(avg_value)

        # MUST FILL NULL SLOPE WITH 0
        stream_segments_attribute_df.fillna(value=0, inplace=True)

        stream_segments_smoothedslope_df_path = os.path.join(watershed_outputs_id,"stream_segments_SAVGslope.shp")
        stream_segments_attribute_df.to_file(
            stream_segments_smoothedslope_df_path, driver="ESRI Shapefile"
        )

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments_SAVGslope'
        stream_segments_attribute_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_segments_SAVGslope.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
## -----------------------------------------------------------------------------------------------------------------------------------------------
## 5) CALCULATE RURAL STREAM POWER
```

```
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # CALCULATE TOTAL STREAM POWER (SP) AND SPECIFIC STREAM POWER (SSP)
        stream_segments_attribute_df["RuralSP"] = pd.to_numeric(
            9810
            * stream_segments_attribute_df["SlopeAVG"]
            * stream_segments_attribute_df["Q_m3pers"],
            errors='coerce'
        )

        # RURAL SSP (EMPIRICAL FORMUL - DA SSP)
        stream_segments_attribute_df["RuralSSP"] = pd.to_numeric(
            stream_segments_attribute_df["RuralSP"] / stream_segments_attribute_df["RurWidth"],
            errors='coerce'
        )

        # FIND THE MAXIMUM RURAL WIDTH OLD METHOD
        max_val_pdmethod = stream_segments_attribute_df["RurWidth"].max()

        # # CALCULATE D84
        # # EQUATION 16 FROM FERGUSON (2005) ASSUMPTION: DI = DB
        # # CONSTANTS FOR D84 PREDICTION
        gamma = 9790 # N/M3
        kappa = 0.41 # CONSTANT FROM FERGUSON 2005 - WORKS FOR D IN MM, POWER IN W/M2
        R = 1.65 # SUBMERGED SPECIFIC GRAVITY
        rho = 1000 # DENSITY OF WATER
        theta_cb = 0.045 # SHEAR STRESS THRESHOLD
        m =2.80 # ROUGHNESS MULTIPLIER FOR D84 FROM LOPEZ AND BARRANGAN E.G. HEY 1979
        gee = 9.81 #ACCELERATION BY GRAVITY

        # C = LOG10(30*THETA_CB*R/(E*M*!S_MPERM!))"
        # EXPRESSIONDB =
"((KAPPA*!SPOWER_WPERM!/(2.30*RHO*LOG10(30*THETA_CB*R/(EXP*M*!S_MPERM!))))**(2/3))/(THETA_CB*R*GEE)"
        stream_segments_attribute_df["SlopeAVG"] = pd.to_numeric(stream_segments_attribute_df["SlopeAVG"], errors='coerce')
        stream_segments_attribute_df["eqlogC"] = np.log10(30 * theta_cb * R / (np.exp(1) * m *
np.abs(stream_segments_attribute_df["SlopeAVG"])))
        stream_segments_attribute_df["D84_mm"] = ((((kappa * stream_segments_attribute_df["RuralSSP"]) / (2.30 * rho *
stream_segments_attribute_df["eqlogC"])) ** (2.0 / 3)) / (theta_cb * R * gee)) * rho

        # SAVE FILE, MUST FILL NULL WITH 0 OR THE COLUMN WILL BE IMPORTED AS A STRING
        stream_segments_attribute_df.fillna(value=0, inplace=True)

        stream_segments_attribute_df_path = os.path.join(watershed_outputs_id,"stream_segments_attribute.shp")
        stream_segments_attribute_df.to_file(
            stream_segments_attribute_df_path, driver="ESRI Shapefile"
        )

        # SAVE A COPY OF SEGMENTS FOR MANIPULATION
        stream_segments_attribute_df_path_c = os.path.join(watershed_outputs_id,"stream_segments_attribute_edited.shp")
        stream_segments_attribute_df.to_file(
            stream_segments_attribute_df_path_c, driver="ESRI Shapefile"
        )

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments_attribute'
        stream_segments_attribute_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments_attribute_edited'
        stream_segments_attribute_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # END OF RURAL STREAM POWER CALCULATION
        # READ ORIGINAL
        gdf_strm = gpd.read_file(
            stream_segments_attribute_df_path_c
        )  # RENAME TO RURAL STREAM POWER FINAL
        copy_strm_net = os.path.join(watershed_outputs_id,"Final_Rural_Stream_Power_Output.shp")
        # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
        gdf_strm.to_file(copy_strm_net, driver="ESRI Shapefile")
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        ## WRITE SHAPEFILES TO A GEOPACKAGE TO SAVE SPACE\
```

```
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'basin_hecpts.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'basin_selected.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_points_initial.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_vector_initial.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'dem_points.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_vector.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'hecpt_buffers50m_join.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'dem_points_initial.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'hecpt_buffers50m.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT HECRAS DATA TO SEGMENTS
## TRANSFORM HECRASFILE FOR FLOOD EVENTS (.CSV) INTO COMPATIBLE FORMAT
## READ FILE AS DATAFRAME (DF) BY SKIPPING 9 ROWS AND SPECIFY COLUMN HEADERS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        if hec_analysis == "Y":
            print("Reading csv\n")
            hec_df = pd.read_csv(hec_csv, header=[0, 1])

            print("Reached csv read point\n")
            # REMOVE UNNAMED CELLS IN COLUMN NAMES
            # FOR NEW VERSION OF PANDAS 0.21 AND OLD
            for i, columns in enumerate(hec_df.columns.levels):
                columns_new = columns.tolist()
                for j, row in enumerate(columns_new):
                    if "Unnamed: " in row:
                        columns_new[j] = ""
                if (
                    pd.__version__ < "0.21.0"
                ):  # HTTPS://STACKOVERFLOW.COM/A/48186976/716469
                    hec_df.columns.set_levels(columns_new, level=i, inplace=True)
                else:
                    hec_df = hec_df.rename(
                        columns=dict(zip(columns.tolist(), columns_new)), level=i
```

```
            )

        # COMBINE BOTH ROWS AS HEADERS
        col_raw = hec_df.columns
        col_raw.tolist()
        # NEW_COL = PD.INDEX([E[0] + E[1] FOR E IN COL_RAW.TOLIST()]) FOR GDB CLASS, LONG NAMES ARE ALLOWED
(I.E. "QCHANNEL_M3PERS")
        new_col = pd.Index(
            [e[0] for e in col_raw.tolist()]
        )  # ONLY NAME, NOT METRIC FOR SHAPEFILE
        hec_df.columns = new_col

        # TRANSFORM FIELD NAME BY REPLACE SPACE AND SYMBOLS WITH UNDERSCORE
        hec_df.columns = hec_df.columns.str.replace(" ", "")
        hec_df.columns = hec_df.columns.str.replace("(", "")
        hec_df.columns = hec_df.columns.str.replace(")", "")
        hec_df.columns = hec_df.columns.str.replace("/", "per")
        hec_df.columns = hec_df.columns.str.replace(".", "")
        hec_df.columns = hec_df.columns.str.replace("_", "")
        hec_df.columns = hec_df.columns.str.lower()

        # SELECT ONLY NEEDED COLUMNS
        hec_df = hec_df[
            [
                "riversta",
                "profile",
                "qtotal",
                "topwidth",
                "qchannel",
                "topwchnl",
                "minchel",
            ]
        ]

        # SHORTEN COLUMN NAME : ONLY FOR SHAPEFILE
        hec_df.rename(
            columns={
                "riversta": "RSta",
                "profile": "Profile",
                "qtotal": "QTotal",
                "topwidth": "TopW",
                "qchannel": "QCh",
                "topwchnl": "TopWCh",
                "minchel": "ChEl",
            },
            inplace=True,
        )

        # TRANSFORM COLUMNS WITH PROFILE - KEEP ONLY NUMBER
        # HEC_DF["PROFILE"] = HEC_DF["PROFILE"].STR.REPLACE(R'\D', R'') # NOT IN USE
        hec_df["Profile"].replace(
            regex=True, inplace=True, to_replace=r"\D", value=r""
        )

        # REPLACE ALL SPACE IN HEC_DF
        hec_df = hec_df.replace(" ", "", regex=True)

        # DROP ROWS WITH NONE
        hec_df = hec_df.dropna(axis=0, how="all")

        # DROP ROWS WITH NONE UNDER PROFILE
        hec_df = hec_df.dropna(subset=["Profile"], axis=0, how="all")

        # REPLACE ALL SPACE IN HEC_DF
        hec_df = hec_df.replace("_", "", regex=True)

        # SAVE THE LIST OF CULVERT IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
        hec_profile_list = list(hec_df["Profile"])
        hec_profile_list_drop = list(dict.fromkeys(hec_profile_list))

        # PIVOT TABLE BY MAKING THE PROFILE (I.E. STORM EVENT) DATA INTO COLUMNS
        # HEC_DF_PIV = HEC_DF.PIVOT(INDEX='RSTA', COLUMNS = 'PROFILE')
        hec_df_piv = pd.pivot_table(
            hec_df, index="RSta", columns="Profile"
        )  # OLD CODE

        # GROUP STACKED COLUMNS
        hec_df_piv.columns = hec_df_piv.columns.map("".join)

        # SAVE TRANSFORMED HEC RAS FILE
        hec_csv_f = str(watershed_outputs_id) + "/hecras_csv_formatted.csv"
        hec_df_piv.to_csv(hec_csv_f)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT RIVER STATION POINTS TO TRANSFORMED HECRAS DATA
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # READ HEC POINTS SHAPEFILE
        hec_point_df = geopandas.read_file(hec_point)

        # SELECT ONLY NEEDED ROWS
        hec_point_df = hec_point_df[["RIVER_STA", "geometry"]]

        # READ HEC FORMATTED DATA
        hec_csv_df = pd.read_csv(hec_csv_f)

        # JOIN TRANSFORMED HECRAS DATA TO HEC POINTS SHAPEFILE
        hec_pts_joined_data = pd.merge(
            hec_point_df,
            hec_csv_df,
            how="inner",
            left_on=["RIVER_STA"],
            right_on=["RSta"],
        )

        # SAVING AS SHAPEFILE
        hec_pts_joined_data_path = os.path.join(watershed_outputs_id,"hec_points_joined2data.shp")
        hec_pts_joined_data.to_file(
            hec_pts_joined_data_path, driver="ESRI Shapefile"
        )

        # WRITE TO A GEOPACKAGE
        layer_name = 'hec_points_joined2data'
        hec_pts_joined_data.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT RIVER STATION POINTS WITH HECRAS DATA TO STREAM REACH
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # OPEN LINE AND POINTS DATA
        gdf_segments = geopandas.read_file(stream_segments_attribute_df_path)
        gdf_hecpoint = geopandas.read_file(hec_pts_joined_data_path)

        # SAVE A COPY OF SEGMENTS FOR MANIPULATION
        gdf_segments_c = geopandas.read_file(
            stream_segments_attribute_df_path_c
        )

        # UNION ALL GEOMETRIES
        shply_line = gdf_segments.geometry.unary_union
        shply_point = gdf_hecpoint.geometry.unary_union

        # SELECT POINTS WITHIN 45 M OF RIVER BY CREATING POLYGON
        buffer_45 = shply_line.buffer(45)
        buff_df = geopandas.GeoDataFrame(geometry=[buffer_45])

        # JOIN POINTS TO BUFFER
        point_in_buffer = geopandas.sjoin(gdf_hecpoint, buff_df, how="left")

        # DROP POINTS OUTSIDE OF BUFFER
        point_45 = point_in_buffer.dropna(axis=0, subset=["index_right"])

        # SAVE POINTS
```

```
point_45_path = os.path.join(watershed_outputs_id,"hec_points_within45m.shp")
point_45.to_file(point_45_path, driver="ESRI Shapefile")

gdf_hecpoint_45m = geopandas.read_file(point_45_path)

# WRITE TO A GEOPACKAGE
layer_name = 'hec_points_within45m'
gdf_hecpoint_45m.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')


# GET CENTROID OF SEGMENTS AND SAVE AS NEW FILE
gdf_segments_c["geometry"] = gdf_segments_c["geometry"].centroid
seg_centroid_path = os.path.join(watershed_outputs_id,"segments_centroid.shp")
gdf_segments_c.to_file(seg_centroid_path, driver="ESRI Shapefile")

# OPEN CENTROID FILE
gdf_centroid = geopandas.read_file(seg_centroid_path)

# WRITE TO A GEOPACKAGE
layer_name = 'segments_centroid'
gdf_centroid.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

# FUNCTION TO FIND NEAREST SEGMENT'S CENTROID TO HECDATA POINT
def nearest(
    row,
    geom_union,
    df1,
    df2,
    geom1_col="geometry",
    geom2_col="geometry",
    src_column=None,
):
    """Find the nearest point and return the corresponding
    value from specified column."""
    # FIND THE GEOMETRY THAT IS CLOSEST
    nearest = (
        df2[geom2_col] == nearest_points(row[geom1_col], geom_union)[1]
    )
    # GET THE CORRESPONDING VALUE FROM DF2 (MATCHING IS BASED ON THE GEOMETRY)
    value = df2[nearest][src_column].values[
        0
    ]  # GET VALUES IS DEPREACIATED THE NEW ONE IS CALLED VALUES
    return value

# CREATE MULTIPOINT: CENTROID OF SEGMENTS
unary_union = gdf_centroid.unary_union

# FIND ID OF LINE CLOSEST TO HEC POINT
gdf_hecpoint_45m["NearSegID"] = gdf_hecpoint_45m.apply(
    nearest,
    geom_union=unary_union,
    df1=gdf_hecpoint_45m,
    df2=gdf_centroid,
    src_column="JoinID",
    axis=1,
)

# ONLY 1 HEC POINT CAN BE ATTACHED TO A SEGMENT. CHOOSE ONLY 1 POINT PER SEGMENT.
gdf_hecpoint_45m_sub = gdf_hecpoint_45m.drop_duplicates(
    subset="NearSegID", keep="first", inplace=False
)

# JOIN HEC POINTS TO STREAM SEGMENTS, MUST DROP FIRST GEOMETRY COLUMN
gdf_hecpoint_45m_sub = gdf_hecpoint_45m_sub.drop(columns="geometry")
hec45m_2_seg = pd.merge(
    gdf_segments,
    gdf_hecpoint_45m_sub,
    how="left",
    left_on=["JoinID"],
    right_on=["NearSegID"],
    )

    hec45m_2_seg_path = os.path.join(watershed_outputs_id,"hec45m_join2_segments.shp")
    hec45m_2_seg.to_file(hec45m_2_seg_path, driver="ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'segments_centroid'
    gdf_centroid.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

    print("Completed saving", hec45m_2_seg_path)
## ---------------------------------------------------------------------------------------------------------------------------
## INTERPOLATE HECRAS DATA FOR SEGMENTS WITHOUT DATA
## ---------------------------------------------------------------------------------------------------------------------------
    # IMPORT SHAPEFILE CONTAINING STORM EVENT DATA AS DATAFRAME
    strm_hec_path = os.path.join(watershed_outputs_id,"hec45m_join2_segments.shp")
    strm_hec_df = geopandas.read_file(strm_hec_path)

    # WRITE TO A GEOPACKAGE
    layer_name = 'hec45m_join2_segments'
    strm_hec_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

    print("Read the file:", strm_hec_df)

    # RESET UNIQUE INDEX OF DATAFRAME USING DISTANCE DOWNSTREAM IN DESCENDING ORDER SO THAT THE
REACHES ARE ORDERED FROM THEIR UPSTREAM LOCATION TO DOWNSTREAM LOCATION
    strm_hec_df["iRID"] = np.arange(len(strm_hec_df))
    strm_hec_df = strm_hec_df.sort_values(by=["DistDown"], ascending=False)
    strm_hec_df.set_index("iRID", inplace=True)

    # REPLACE NAN WITH NONE
    strm_hec_df = strm_hec_df.replace({np.nan: None})

    # FIELDS
    river_sta_id = "RSta" # RIVER_STA
    unique_id = "JoinID" # REACH ID
    dist_value = "Length" # STREAM LENGTH

    #ONLY REQUIRED FIELDS FOR STREAM POWER WILL BE INTERPOLATED
    req_fields = ['QCh','TopWCh', 'ChEl']

    raw_value_split = []

    for fld in req_fields:
        for p in hec_profile_list_drop:
            val = str(fld) + str(p)
            raw_value_split.append(val)

    # LOOP THROUGH EACH FIELD FOR INTERPOLATION AND CREATE NEW FIELD NAMES
    calc_name_list = []
    for r in raw_value_split:
        raw_value = str(r)
        calc_name = str("i") + str(r)
        calc_name_list.append(calc_name)
        # ADD FIELD TO IDENTIFY REACHES THAT WILL BE INTERPOLATED BETWEEN TWO SELECTED RIVER STATIONS
        strm_hec_df["iID"] = None
        strm_hec_df["iLENGTH"] = None
        # NEW FIELD FOR CALCULATED VALUE
        strm_hec_df[calc_name] = None
        # CREATE LIST WITH ALL RIVER STATION IDS WHERE DATA EXISTS.
        sta_list = []
        sta_list_c = []
        for index, row in strm_hec_df.iterrows():
            r_sta = row[river_sta_id]
            r_id = row[unique_id]
            raw_v = row[raw_value]
            if raw_v is not None:
                pair = (r_id, r_sta)
                sta_list.append(pair)
                sta_list_c.append(pair)
        #PRINT(STA_LIST)
```

```
        # LOOP THROUGH TABLE, GET FIRST STATION IN LIST OF STATION WITH EXISTING DATA, ASSIGN A CUMULATIVE LENGTH (INTP_LENGTH) OF 0 TO THE FIRST STATION, SAVE THE VALUE FOR THE FIELD
        # TO BE INTERPOLATED IN Y_LIST AND SAVE THE CUMULATIVE DISTANCE IN THE X_LIST.
        y_list = []
        x_list = []
        x_intp_list = []

        intp_id = 0

        while len(sta_list_c) != 0:
            x_list=[]
            y_list=[]
            x_intp_list = []
            # BEGIN LOOP TO GET FIRST STATION
            first_sta = sta_list_c[0][1]
            first_rid = sta_list_c[0][0]
            #PRINT(FIRST_STA, FIRST_RID)
            intp_id = intp_id + 1
            del sta_list_c[0]
            #PRINT("PROCESSING STATION " + STR(FIRST_STA))
            for index, row in strm_hec_df.iterrows():
                r_sta = row[river_sta_id]
                r_id = row[unique_id]
                if first_sta == r_sta and first_rid == r_id:
                    #PRINT("R_STA " + STR(R_STA))
                    #PRINT("R_ID " + STR(R_ID))
                    dist_sum = 0
                    #print(dist_sum, "In if loop")
                    y_start = 0
                    start_endxy = row["EndXY"] # GET END OF REACH
                    #PRINT("STARTSTA: " + STR())
                    first_rawv = row[raw_value] # GET THE VALUE TO AVERAGE
                    x_list.append(y_start)    # ADD FIRST X=0 TO X LIST
                    y_list.append(first_rawv) # ADD FIRST ELEV TO Y LIST
                    strm_hec_df.loc[index,calc_name] = first_rawv
                    strm_hec_df.loc[index,"iID"] = intp_id
                    strm_hec_df.loc[index,"iLENGTH"] = dist_sum
                    break

            dist_sum
            #print(dist_sum)
            # TRACE DOWNSTREAM USING END XY COORDINATES OF FIRST REACH TO FIND START XY COORDINATES OF SUBSEQUENT REACH
            # IF BOTH RAW VALUE FIELD (FIELD TO BE INTERPOLATED) AND INTERPOLATED FIELD (CALC_NAME)ARE NONE, THEN UPDATE
            # THE REACH'S CUMULENGTH LENGTH (INTP_LENGTH), IDENTIFIER (INTP_ID) AND SAVE THE REACH ID. IF RAW VALUE EXISTS, BREAK LOOP.
            # E.G. LET'S INTERPOLATE ELEVATION VALUES
            # REACH ID = [1,2,3,4]  STATIONS 2 AND 3 ARE LOCATED BETWEEN 1 AND 4. 1 AND 4 HAVE HECRAS DATA.
            # INTP_LENGTH = [0,10,20,30]
            # INTP_ID = [1,1,1,1]
            # RAW VALUE = [120, NA, NA, 30]
            # X_LIST = [0,10,20,30] # SAME AS CUMULATIVE LENGTH
            # Y_LIST = [120, NA, NA, 30]
            # INTERPOLATED RAW VALUE = [120, 90, 60,30]
            for index, row in strm_hec_df.iterrows():
                r_startxy = row["StartXY"]
                if r_startxy == start_endxy:
                    distv = row[dist_value] # GET THE DISTANCE
                    rawv = row[raw_value] # GET THE ELEV
                    intp_v = row[calc_name]
                    #PRINT DISTV
                    #PRINT RAWV
                    #PRINT INTP_V
                    if rawv == None or np.isnan(rawv) and intp_v == None:
                        dist_sum = dist_sum + distv
                        #PRINT("SUM DISTANCE " + STR(DIST_SUM))
                        x_intp_list.append(dist_sum)
                        strm_hec_df.loc[index,"iLENGTH"] = dist_sum
                        strm_hec_df.loc[index,"iID"] = intp_id
                        start_endxy = row["EndXY"]

                        #DELETE PAIR IN LIST IF EXIST
                        r_sta = row[river_sta_id]
                        r_id = row[unique_id]
                        pair = (r_id,r_sta)
                        #PRINT(PAIR)
                        if pair in sta_list_c:
                            sta_list_c.remove(pair)
                    elif rawv is not None:
                        dist_sum = dist_sum + distv
                        y_list.append(rawv)
                        x_list.append(dist_sum)
                        break
            # USE X_LIST (CUMULATIVE LENGTH) AND Y_LIST (RAW VALUES)
            for index, row in strm_hec_df.iterrows():
                intp_id_tbl = row["iID"]
                y_sum_tbl = row["iLENGTH"]
                rawv_tbl = row[calc_name]
                if intp_id == intp_id_tbl and rawv_tbl == None:
                    intp_rawv = np.interp(y_sum_tbl, x_list, y_list)
                    if len(x_list) == 1:
                        strm_hec_df.loc[index,calc_name] = None   #### MUST CHECK THIS. IF DIFFERENCE IS TOO SMALL, MIN ELEV MAY STAY THE SAME. (SOLVED)
                    else:
                        strm_hec_df.loc[index,calc_name] = intp_rawv

        # SAVE FILE, REPLACE NONE WITH 0
        strm_hec_df.fillna(value=np.nan, inplace=True)
        #STRM_HEC_DF.FILLNA(0, INPLACE=TRUE)
        strm_hec_df.replace(to_replace= np.nan, value=0, inplace=True)


        strm_hec_gdf_i_path = os.path.join(watershed_outputs_id, "segment_dem_hec_connected_intp.shp")
        strm_hec_df.to_file(strm_hec_gdf_i_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'segment_dem_hec_connected_intp'
        strm_hec_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        print("Saved file", strm_hec_gdf_i_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## 8) CALCULATE STREAM POWER (URBAN SCENARIO)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # CREATE NEW FIELDS TO STORE TOTAL AND SPECIFIC STREAM POWER
        strm_hec_intp_gdf = geopandas.read_file(strm_hec_gdf_i_path)

        # CHECK IF ITOPWCH2 (2 YR RETURN PERIOD) IS LESS THAN RURAL WIDTH, IF LESS THAN EMPIRICAL USE THE RURAL WIDTH AT GIVEN POINT
        # LOOP THROUGH THE ROWS AND UPDATE 'MODW' BASED ON THE CURRENT WIDTH CALC
        # DO WIDTHS FOR EVERY STORM

        for p in hec_profile_list_drop:
            check_field = "iTopWCh" + str(p)
            new_emp_width = "ModW" + str(p)
            #NEW_ASH_WIDTH = "MODWASH" + STR(P)

            strm_hec_intp_gdf[new_emp_width] = None

            # UPDATE NEW_EMP_WIDTH COLUMN BASED ON CONDITIONS
            strm_hec_intp_gdf[check_field] = pd.to_numeric(strm_hec_intp_gdf[check_field], errors='coerce').astype(float)
            mask = (strm_hec_intp_gdf[check_field] >= 0) & (strm_hec_intp_gdf[check_field] >= strm_hec_intp_gdf["RurWidth"])
            strm_hec_intp_gdf.loc[mask, new_emp_width] = strm_hec_intp_gdf.loc[mask, check_field]
            strm_hec_intp_gdf.loc[~mask, new_emp_width] = strm_hec_intp_gdf.loc[~mask, "RurWidth"]

            # CONVERT NEW_EMP_WIDTH COLUMN TO FLOAT
            strm_hec_intp_gdf[new_emp_width] = pd.to_numeric(strm_hec_intp_gdf[new_emp_width], errors="coerce").astype(float)

            # CONVERT COLUMNS TO NUMERIC DATA TYPES
            strm_hec_intp_gdf[check_field] = pd.to_numeric(strm_hec_intp_gdf[check_field], errors="coerce")
```

```
# SAVE THE UPDATED SHAPEFILE
strm_hec_intp_gdf.to_file(strm_hec_gdf_i_path)

for p in hec_profile_list_drop:

    sp_field = "SP" + str(p)
    disch_field = "iQCh" + str(p)
    strm_hec_intp_gdf["SlopeAVG"] = pd.to_numeric(strm_hec_intp_gdf["SlopeAVG"], errors="coerce")
    strm_hec_intp_gdf[disch_field] = pd.to_numeric(strm_hec_intp_gdf[disch_field], errors="coerce")
    strm_hec_intp_gdf[sp_field] = None
    strm_hec_intp_gdf[sp_field] = (
        9810
        * strm_hec_intp_gdf["SlopeAVG"]
        * strm_hec_intp_gdf[disch_field]
    )

# CALCULATE SPECIFIC STREAM POWER (SSP) = SP/WIDTH
for p in hec_profile_list_drop:
    ssp_field = "SSP" + str(p)
    w_field = "iTopWCh" + str(p)
    sp_field = "SP" + str(p)
    strm_hec_intp_gdf[ssp_field] = None
    strm_hec_intp_gdf[ssp_field] = (
        strm_hec_intp_gdf[sp_field] / strm_hec_intp_gdf[w_field]
    )

# CALCULATE SSPR = URBAN SSP/ RURAL SSP: IN THIS CASE SSP2 REPRESENTS RURAL CONDITION
if "SSP2" in strm_hec_intp_gdf.columns:
    for p in hec_profile_list_drop:
        rural_ssp_field = "SSP2"
        urban_ssp_field = "SSP" + str(p)
        sspr_field = "SSPR" + str(p)
        strm_hec_intp_gdf[sspr_field] = None
        strm_hec_intp_gdf[sspr_field] = (
            strm_hec_intp_gdf[urban_ssp_field]
            / strm_hec_intp_gdf[rural_ssp_field]
        )

# CALCULATE SSPR = SSP2/RURALSSP AND COMPARE 2-YEAR URBAN TO RURAL
if "SSP2" in strm_hec_intp_gdf.columns and "RuralSSP" in strm_hec_intp_gdf.columns:
    sspr_q2_field = "SSPR"
    strm_hec_intp_gdf[sspr_q2_field] = None
    strm_hec_intp_gdf["SSP2"] = strm_hec_intp_gdf["SSP2"].astype(float)
    strm_hec_intp_gdf["RuralSSP"] = strm_hec_intp_gdf["RuralSSP"].astype(float)
    strm_hec_intp_gdf.loc[:, sspr_q2_field] = strm_hec_intp_gdf["SSP2"] / strm_hec_intp_gdf["RuralSSP"]

# THIS IS WITH THE TEST WHERE IF ITOPWCH IS LESS THAN EMPIRICAL THEN CREATE A NEW COLUMN
# CALCULATE SPECIFIC STREAM POWER (SSP) = SP/WIDTH;
for p in hec_profile_list_drop:
    ssp_field = "ModSSP" + str(p)
    w_field = "ModW" + str(p)
    sp_field = "SP" + str(p)
    strm_hec_intp_gdf[ssp_field] = None

    mask = strm_hec_intp_gdf[w_field] > 0
    strm_hec_intp_gdf.loc[mask, ssp_field] = (
        strm_hec_intp_gdf.loc[mask, sp_field]
        / strm_hec_intp_gdf.loc[mask, w_field]
    )
    strm_hec_intp_gdf.loc[~mask, ssp_field] = 0

    strm_hec_intp_gdf[ssp_field] = pd.to_numeric(
        strm_hec_intp_gdf[ssp_field], errors="coerce"
    ).astype(float)

# CALCULATE SSPR = URBAN SSP/ RURAL SSP: IN THIS CASE SSP2 REPRESENTS RURAL CONDITION
# COMPARE EACH STM EVENT WITH RURAL SSP, 2 YR RURAL WITH 2 YR URBAN
# COMPARE THE SSP FOR ALL EVENTS TO THE 2 YR AND COMPARE 2YR TO THE RURAL
if "ModSSP2" in strm_hec_intp_gdf.columns:
    for p in hec_profile_list_drop:
        rural_ssp_field = "ModSSP2"  # MEANS COMPARING TO THE NORMAL CONDITION (LE MOST FREQUENT 2 YR)
        urban_ssp_field = "ModSSP" + str(p)
        sspr_field = "ModSSPR" + str(p)
        strm_hec_intp_gdf[sspr_field] = None

        mask2 = strm_hec_intp_gdf[rural_ssp_field] > 0
        strm_hec_intp_gdf.loc[mask2, sspr_field] = (
            strm_hec_intp_gdf.loc[mask2, urban_ssp_field]
            / strm_hec_intp_gdf.loc[mask2, rural_ssp_field]
        )
        strm_hec_intp_gdf.loc[~mask2, sspr_field] = 0

        strm_hec_intp_gdf[sspr_field] = pd.to_numeric(
            strm_hec_intp_gdf[sspr_field], errors="coerce"
        ).astype(float)

    # CALCULATE SSPR = SSP2/SSPR,  AND COMPARE 2YR URBAN TO THE RURAL
    if "ModSSP2" in strm_hec_intp_gdf.columns:
        sspr_q2_field = "ModSSPR"
        strm_hec_intp_gdf[sspr_q2_field] = None
        strm_hec_intp_gdf[sspr_q2_field] = (
            strm_hec_intp_gdf["ModSSP2"] / strm_hec_intp_gdf["RuralSSP"]
        )

        strm_hec_intp_gdf[sspr_q2_field] = pd.to_numeric(
            strm_hec_intp_gdf[sspr_q2_field], errors="coerce"
        ).astype(float)

        # SET PRJ AND SAVE FILE
        strm_hec_intp_gdf.crs = output_crs
        strm_hec_intp_gdf_calc_path = os.path.join(watershed_outputs_id,"segment_dem_hec_streampow_prj.shp")
        strm_hec_intp_gdf.to_file(
            strm_hec_intp_gdf_calc_path, driver="ESRI Shapefile"
        )

        # WRITE TO A GEOPACKAGE
        layer_name = 'segment_dem_hec_streampow_prj'
        strm_hec_intp_gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## RENAME TO HECRAS STREAM POWER FINAL
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        gdf_hecfinal = gpd.read_file(
            strm_hec_intp_gdf_calc_path
        )  # READ THE ORIGINAL SHAPEFILE
        copy_hecfinal = (
            str(watershed_outputs_id) + "\Final_HECRAS_Stream_Power_Output.shp"
        )  # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
        gdf_hecfinal.to_file(copy_hecfinal, driver="ESRI Shapefile")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## END OF HECRAS CROSS SECTION ANALYSIS
## CONNECT HECRAS BRIDGE DATA TO SEGMENTS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    if bridge_analysis == "Y":
        # TRANSFORM HEC CSV INTO COMPATIBLE FORMAT
        # READ FILE AS DATAFRAME BY SKIPPING 9 ROWS AND SPECIFY COLUMN HEADERS
        df_bridge = pd.read_csv(bridge_hec_csv, header=[0, 1])

        # REMOVE UNNAMED CELLS IN COLUMN NAMES
        # FOR NEW VERSION OF PANDAS 0.21 AND OLD
        for i, columns in enumerate(df_bridge.columns.levels):
            columns_new = columns.tolist()
            for j, row in enumerate(columns_new):
                if "Unnamed: " in row:
                    columns_new[j] = ""
            if (
                pd.__version__ < "0.21.0"
            ):  # HTTPS://STACKOVERFLOW.COM/A/48186976/716469
                df_bridge.columns.set_levels(columns_new, level=i, inplace=True)
            else:
```

```
            df_bridge = df_bridge.rename(
                columns=dict(zip(columns.tolist(), columns_new)), level=i
            )

        # COMBINE BOTH ROWS AS HEADERS
        col_raw = df_bridge.columns
        col_raw.tolist()
        # NEW_COL = PD.INDEX([E[0] + E[1] FOR E IN COL_RAW.TOLIST()]) FOR GDB CLASS, LONG NAMES ARE ALLOWED
(I.E. "QCHANNEL_M3PERS")
        new_col = pd.Index(
            [e[0] for e in col_raw.tolist()]
        )  # ONLY NAME, NOT METRIC FOR SHAPEFILE
        df_bridge.columns = new_col

        # TRANSFORM FIELD NAME BY REPLACE SPACE AND SYMBOLS WITH UNDERSCORE
        df_bridge.columns = df_bridge.columns.str.replace(" ", "")
        df_bridge.columns = df_bridge.columns.str.replace("(", "")
        df_bridge.columns = df_bridge.columns.str.replace(")", "")
        df_bridge.columns = df_bridge.columns.str.replace("/", "per")
        df_bridge.columns = df_bridge.columns.str.replace(".", "")
        df_bridge.columns = df_bridge.columns.str.replace("_", "")
        df_bridge.columns = df_bridge.columns.str.lower()

        # SELECT COLUMNS NEEDED
        df_bridge = df_bridge[
            [
                "reach",
                "riversta",
                "profile",
                "qtotal",
                "qchannelupstream",
                "qbridge",
                "deckwidth",
                "topwchnlupstream",
                "scourtype",
            ]
        ]

        # SHORTEN COLUMN NAME : ONLY FOR SHAPEFILE
        df_bridge.rename(
            columns={
                "reach": "Reach",
                "riversta": "RSta",
                "profile": "Profile",
                "qtotal": "QTot",
                "qchannelupstream": "QChUp",
                "qbridge": "QBr",
                "deckwidth": "DeckW",
                "topwchnlupstream": "TopWChUp",
                "scourtype": "scour"
            },
            inplace=True,
        )

        # TRANSFORM COLUMNS WITH PROFILE - KEEP ONLY NUMBER
        df_bridge["Profile"].replace(
            regex=True, inplace=True, to_replace=r"\D", value=r""
        )

        # DELETE SPACE IN DATAFRAME WITH UNDERSCORE
        df_bc = df_bridge.replace(" ", "", regex=True)

        # DROP ROWS WITH NONE
        df_bc = df_bc.dropna(axis=0, how="all")

        # DROP ROWS WITH NONE UNDER PROFILE
        df_bc = df_bc.dropna(subset=["Profile"], axis=0, how="all")

        # SAVE THE LIST OF BRIDGE IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
        bridge_profile_list = list(df_bc["Profile"])
        bridge_profile_list_drop = list(dict.fromkeys(bridge_profile_list))

        # PIVOT TABLE TO TRANSFORM PROFILE FROM ROW WISE TO COLUMN WISE
        df_piv_b = pd.pivot_table(df_bc, index="RSta", columns="Profile")

        # GROUP STACKED COLUMNS
        df_piv_b.columns = df_piv_b.columns.map("".join)

        # SAVE TRANSFORMED DATA WITH NEW FIELDS
        hec_csv_bf_path = os.path.join(watershed_outputs_id,"hecras_bridge_formatted.csv")
        df_piv_b.to_csv(hec_csv_bf_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT TRANSFORMED BRIDGE DATA TO GDB TABLE. TO CALCULATE BRIDGE SCOUR AND BRIDGE OVERTOPPING,
THE DATA FOR THE UPSTREAM RIVER STATION AND AT THE BRIDGE ARE REQUIRED.
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # CONVERT CSV TO GDB TABLE
        gdf_bridge = pd.read_csv(hec_csv_bf_path)

        # ADD NEW COLUMNS
        bridge_scour_label_list = []
        bridge_overtop_label_list = []

        for y in bridge_profile_list_drop:
            new_lab = "SBr" + str(y)
            bridge_scour_label_list.append(new_lab)
            print(new_lab)
            # CREATE EMPTY FIELD
            gdf_bridge[new_lab] = ""
            # APPLY EQUATION
            q_br_lab = "QBr" + str(y)
            print(q_br_lab)
            qchUp_br_lab = "QChUp" + str(y)
            print(qchUp_br_lab)
            topWChUp_br_lab = "TopWChUp" + str(y)
            print(topWChUp_br_lab)
            deck_lab = "DeckW" + str(y)
            print(deck_lab)
            # CHECK THE SPECIFIC COLUMN (E.G., "SCOURTYPE_COLUMN") FOR THE VALUE 1 BEFORE PERFORMING THE
CALCULATION
            scourtype_column = "scour" + str(y) # REPLACE THIS WITH THE ACTUAL COLUMN NAME FOR SCOURTYPE

            # APPLY THE CALCULATION ONLY WHEN SCOURTYPE IS EQUAL TO 1
            scourtype_1_mask = gdf_bridge[scourtype_column] == 1
            gdf_bridge.loc[scourtype_1_mask, new_lab] = (
                (gdf_bridge.loc[scourtype_1_mask, q_br_lab] / gdf_bridge.loc[scourtype_1_mask, qchUp_br_lab]) ** (6 / 7)
            ) * ((gdf_bridge.loc[scourtype_1_mask, topWChUp_br_lab] / gdf_bridge.loc[scourtype_1_mask, deck_lab]) ** 0.59)

        for y in bridge_profile_list_drop:
            new_lab = "OBr" + str(y)
            bridge_overtop_label_list.append(new_lab)
            print(new_lab)
            gdf_bridge[new_lab] = ""
            # APPLY EQUATION
            qtot_lab = "QTot" + str(y)
            qbridge_lab = "QBr" + str(y)
            # CALCULATE
            gdf_bridge[new_lab] = gdf_bridge[qtot_lab] / gdf_bridge[qbridge_lab]

        # SUBSET ONLY CALCULATED VALUES, ADD RIVER STA AND SAVE
        merged_bridge_ratios_lab = (
            bridge_scour_label_list + bridge_overtop_label_list
        )
        merged_bridge_ratios_lab.insert(0, "RSta")
        merged_bridge_ratios_df = gdf_bridge[merged_bridge_ratios_lab]

        # CREATE A COPY OF TABLE
        gdf_bridge_path = os.path.join(watershed_outputs_id,"bridge_scour_overtop_ratios_only.csv")
        merged_bridge_ratios_df.to_csv(gdf_bridge_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
```

```
## CONNECT BRIDGE CALCULATIONS TO A COPY OF RIVER STATIONS POINTS FILE
## ------------------------------------------------------------------------------------------------------------------------------------------
        # MAKE A COPY OF POINTS FILE
        bridge_point_df = geopandas.read_file(bridge_point)
        bridge_point_df = bridge_point_df[["RIVER_STA", "geometry"]]

        # JOIN BRIDGE CALCULATIONS TO HEC POINTS SHAPEFILE
        bridge_pts_joined = pd.merge(
            bridge_point_df,
            merged_bridge_ratios_df,
            how="inner",
            left_on=["RIVER_STA"],
            right_on=["RSta"],
        )
## ------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT BRIDGE POINTS TO STREAM REACH
## ------------------------------------------------------------------------------------------------------------------------------------------
        # OPEN A COPY OF SEGMENTS CENTROID FILE
        gdf_centroid_c = geopandas.read_file(seg_centroid_path)
        gdf_segments_c = geopandas.read_file(
            stream_segments_attribute_df_path_c
        )
        gdf_hec45m_2_seg = geopandas.read_file(copy_hecfinal)

        # OPEN CENTROID FILE
        gdf_centroid = geopandas.read_file(seg_centroid_path)

        # FUNCTION TO FIND NEAREST SEGMENT'S CENTROID TO HECDATA POINT
        def nearest(
            row,
            geom_union,
            df1,
            df2,
            geom1_col="geometry",
            geom2_col="geometry",
            src_column=None,
        ):
            """Find the nearest point and return the corresponding
            value from specified column."""
            # FIND THE GEOMETRY THAT IS CLOSEST
            nearest = (
                df2[geom2_col] == nearest_points(row[geom1_col], geom_union)[1]
            )
            # GET THE CORRESPONDING VALUE FROM DF2 (MATCHING IS BASED ON THE GEOMETRY)
            value = df2.loc[nearest, src_column].iloc[0]
            return value

        # CREATE MULTIPOINT: CENTROID OF SEGMENTS
        unary_union = gdf_centroid_c.unary_union

        # FIND ID OF LINE CLOSEST TO HEC POINT
        bridge_pts_joined["NearSegID"] = bridge_pts_joined.apply(
            nearest,
            geom_union=unary_union,
            df1=bridge_pts_joined,
            df2=gdf_centroid,
            src_column="JoinID",
            axis=1,
        )

        # ONLY 1 HEC POINT CAN BE ATTACHED TO A SEGMENT. CHOOSE ONLY 1 POINT PER SEGMENT.
        bridge_pts_joined_sub = bridge_pts_joined.drop_duplicates(
            subset="NearSegID", keep="first", inplace=False
        )

        # JOIN HEC POINTS TO RAW STREAM SEGMENTS, MUST DROP GEOMETRY BEFORE MERGING
        bridge_pts_joined_sub = bridge_pts_joined_sub.drop(columns="geometry")
        bridge_2_seg = pd.merge(
            gdf_segments_c,
            bridge_pts_joined_sub,
            how="left",
            left_on=["JoinID"],
            right_on=["NearSegID"],
        )

        bridge_2_seg_path = os.path.join(watershed_outputs_id,"bridge_joined2_segments.shp")
        bridge_2_seg.to_file(bridge_2_seg_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'bridge_joined2_segments'
        bridge_2_seg.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------
## JOIN BRIDGE CALCULATIONS TO FINAL STREAM SEGMENTS
## ------------------------------------------------------------------------------------------------------------------------------------------
        # JOIN ONLY BRIDGE CALCULATIONS TO FINAL RESULTS
        bridge_pts_joined_sub = bridge_pts_joined_sub.drop(columns="RIVER_STA")
        bridge_pts_joined_sub.rename(
            columns={"RSta": "RStaBr", "NearSegID": "NearSegBr"}, inplace=True
        )
        bridge_2_seg_hec = pd.merge(
            gdf_hec45m_2_seg,
            bridge_pts_joined_sub,
            how="left",
            left_on=["JoinID"],
            right_on=["NearSegBr"],
        )

        # SET PROJECTION AND SAVE
        bridge_2_seg_hec.crs = output_crs
        bridge_2_seg_hec_path = os.path.join(watershed_outputs_id,"segment_dem_hec_streampow_bridge_prj.shp")
        bridge_2_seg_hec.to_file(bridge_2_seg_hec_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'segment_dem_hec_streampow_bridge_prj'
        bridge_2_seg_hec.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------
## RENAME TO BRIDGE STREAM POWER FINAL
## ------------------------------------------------------------------------------------------------------------------------------------------
        gdf_bridgefinal = gpd.read_file(
            bridge_2_seg_hec_path
        )  # READ THE ORIGINAL SHAPEFILE
        copy_bridgefinal = os.path.join(watershed_outputs_id,"Final_Bridge_Stream_Power_Output.shp")
        # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
        gdf_bridgefinal.to_file(copy_bridgefinal, driver="ESRI Shapefile")
## ------------------------------------------------------------------------------------------------------------------------------------------
## END OF BRIDGE ANALYSIS
## CONNECT CULVERT HECRAS DATA TO SEGMENTS
## ------------------------------------------------------------------------------------------------------------------------------------------
    if culvert_analysis == "Y":
        # TRANSFORM HECRASFILE FOR FLOOD EVENTS (.CSV) INTO COMPATIBLE FORMAT
        # READ FILE AS DATAFRAME (DF) BY SKIPPING 9 ROWS AND SPECIFY COLUMN HEADERS
        cul_df = pd.read_csv(culv_hec_csv, header=[0, 1])

        # REMOVE UNNAMED CELLS IN COLUMN NAMES
        # FOR NEW VERSION OF PANDAS 0.21 AND OLD
        for i, columns in enumerate(cul_df.columns.levels):
            columns_new = columns.tolist()
            for j, row in enumerate(columns_new):
                if "Unnamed: " in row:
                    columns_new[j] = ""
            if (
                pd.__version__ < "0.21.0"
            ):  # HTTPS://STACKOVERFLOW.COM/A/48186976/716469
                cul_df.columns.set_levels(columns_new, level=i, inplace=True)
            else:
                cul_df = cul_df.rename(
                    columns=dict(zip(columns.tolist(), columns_new)), level=i
                )

        # COMBINE BOTH ROWS AS HEADERS
```

```
        col_raw = cul_df.columns
        col_raw.tolist()
        # NEW_COL = PD.INDEX([E[0] + E[1] FOR E IN COL_RAW.TOLIST()]) FOR GDB CLASS, LONG NAMES ARE ALLOWED
(I.E. "QCHANNEL_M3PERS")
        new_col = pd.Index(
            [e[0] for e in col_raw.tolist()]
        )  # ONLY NAME, NOT METRIC FOR SHAPEFILE
        cul_df.columns = new_col

        # TRANSFORM FIELD NAME BY REPLACE SPACE AND SYMBOLS WITH UNDERSCORE
        cul_df.columns = cul_df.columns.str.replace(" ", "")
        cul_df.columns = cul_df.columns.str.replace("(", "")
        cul_df.columns = cul_df.columns.str.replace(")", "")
        cul_df.columns = cul_df.columns.str.replace("/", "per")
        cul_df.columns = cul_df.columns.str.replace(".", "")
        cul_df.columns = cul_df.columns.str.replace("_", "")

        # SELECT ONLY NEEDED ROWS
        cul_df = cul_df[
            ["RiverSta", "ID", "Profile", "QCulvGroup", "CulvSpan", "QTotal"]
        ]

        # SHORTEN COLUMN NAME : ONLY FOR SHAPEFILE
        cul_df.rename(
            columns={
                "RiverSta": "RSta",
                "ID": "ID",
                "Profile": "Profile",
                "QCulvGroup": "QCul",
                "CulvSpan": "DCul",
                "QTotal": "QTot",
            },
            inplace=True,
        )

        # TRANSFORM COLUMNS WITH PROFILE - KEEP ONLY NUMBER
        # HEC_DF["PROFILE"] = HEC_DF["PROFILE"].STR.REPLACE(R'\D', R'') # NOT IN USE
        cul_df["Profile"].replace(
            regex=True, inplace=True, to_replace=r"\D", value=r""
        )

        # REPLACE ALL SPACE IN HEC_DF
        cul_df = cul_df.replace(" ", "", regex=True)

        # DROP ROWS WITH NONE
        cul_df = cul_df.dropna(axis=0, how="all")

        # DROP ROWS WITH NONE UNDER PROFILE
        cul_df = cul_df.dropna(subset=["Profile"], axis=0, how="all")

        # REPLACE ALL SPACE IN HEC_DF
        cul_df = cul_df.replace("_", "", regex=True)

        # SAVE THE LIST OF CULVERT IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
        culv_profile_list = list(cul_df["Profile"])
        culv_profile_list_drop = list(dict.fromkeys(culv_profile_list))

        # PIVOT TABLE BY MAKING THE PROFILE (I.E. STORM EVENT) DATA INTO COLUMNS
        # CONCATENATE THE RSTA WITH THE CULVERT ID
        cul_df["ID"] = cul_df["ID"].astype("int64")
        cul_df["CulvID"] = cul_df["RSta"].astype(str) + cul_df["ID"].astype(str)

        # SUBSET ONLY THE FIELDS NEEDED FOR PIVOTING
        cul_df_sub = cul_df[["CulvID", "Profile", "QCul", "DCul", "QTot"]]

        # HEC_DF_PIV = HEC_DF.PIVOT(INDEX='RSTA', COLUMNS = 'PROFILE')
        cul_df_piv = pd.pivot_table(
            cul_df_sub, index="CulvID", columns="Profile"
        )  # OLD CODE

        # GROUP STACKED COLUMNS
        cul_df_piv.columns = cul_df_piv.columns.map("".join)

        # FORMAT DF BY REMOVING DUPLICATES FOR JOINING TO PIVOTED TABLE
        cul_df_sub2 = cul_df[["RSta", "ID", "CulvID"]]
        cul_df_sub2_noduplicate = cul_df_sub2.drop_duplicates(
            ["RSta", "CulvID"], keep="last"
        )

        # ADD RIVER STATION TO PIVOTED DATA
        cul_df_piv_all = pd.merge(
            cul_df_piv,
            cul_df_sub2_noduplicate,
            how="left",
            left_on="CulvID",
            right_on="CulvID",
        )

        # PIVOT TABLE BY ATTACHING ID OF CULVERT (E.G. 1,2,3) TO COLUMN NAMES
        # SUBSET ONLY FIELDS NEEDED FOR PIVOTING
        cul_df_piv_all_sub = cul_df_piv_all.drop(["CulvID"], axis=1)
        cul_df_piv_all_sub = cul_df_piv_all_sub.fillna(0.0)

        # CONVERT ID TO STRING BEFORE ATTACHING TO COLUMN NAME
        cul_df_piv_all_sub["ID"] = cul_df_piv_all_sub["ID"].astype("int64")

        # SAVE THE LIST OF CULVERT IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
        culv_id_list = list(cul_df_piv_all_sub["ID"])
        culv_id_list_drop = list(dict.fromkeys(culv_id_list))

        # CONVERT COLUMN WITH CULVERT ID TO BE ATTACHED TO COLUMN NAME
        cul_df_piv_all_sub["ID"] = cul_df_piv_all_sub["ID"].astype(str)
        cul_df_piv_all_sub["ID"] = "_" + cul_df_piv_all_sub["ID"].astype(str)

        # PIVOT TABLE BASED ON ID
        cul_df_piv_all_sub_piv = pd.pivot_table(
            cul_df_piv_all_sub, index="RSta", columns="ID"
        )  # OLD CODE

        # GROUP STACKED COLUMNS
        cul_df_piv_all_sub_piv.columns = cul_df_piv_all_sub_piv.columns.map(
            "".join
        )

        # SAVE TRANSFORMED HEC RAS FILE
        cul_csv_f = os.path.join(watershed_outputs_id,"culvert_csv_formatted.csv")
        cul_df_piv_all_sub_piv.to_csv(cul_csv_f)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT TRANSFORMED CULVERT DATA TO GDB TABLE. TO CALCULATE CULVERT SCOUR AND CULVERT
OVERTOPPING, THE DATA FOR THE TOTAL DISCHARGE, DISCHARGE AT CULVERT AND CULVERT DIAMETER ARE
REQUIRED.
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # ADD NEW COLUMNS TO CALCULATE CULVERT SCOUR AND OVERTOPPING
        culv_scour_label_list = []
        culv_overtop_label_list = []

        for i in culv_id_list_drop:
            for y in culv_profile_list_drop:
                new_lab = "SCul" + str(y) + "_" + str(i)
                culv_scour_label_list.append(new_lab)
                print(new_lab)
                # CREATE EMPTY FIELD
                cul_df_piv_all_sub_piv[new_lab] = ""
                # APPLY EQUATION
                qcul_lab = "QCul" + str(y) + "_" + str(i)
                print(qcul_lab)
                dcul_lab = "DCul" + str(y) + "_" + str(i)
                print(dcul_lab)
```

```
            # CALCULATE NEW FIELD
            g = 9.81
            denom = (g**0.5) * (cul_df_piv_all_sub_piv[dcul_lab] ** 2.5)
            brac = (cul_df_piv_all_sub_piv[qcul_lab] / denom) ** 0.45
            cul_df_piv_all_sub_piv[new_lab] = 2.07 * brac

        for i in culv_id_list_drop:
            for y in culv_profile_list_drop:
                new_lab = "OCul" + str(y) + "_" + str(i)
                culv_overtop_label_list.append(new_lab)
                # PRINT(NEW_LAB)
                cul_df_piv_all_sub_piv[new_lab] = ""
                # APPLY EQUATION
                qtot_lab = "QTot" + str(y) + "_" + str(i)
                qcul_lab = "QCul" + str(y) + "_" + str(i)
                # CALCULATE
                cul_df_piv_all_sub_piv[new_lab] = (
                    cul_df_piv_all_sub_piv[qtot_lab]
                    / cul_df_piv_all_sub_piv[qcul_lab]
                )

        # SUBSET ONLY CALCULATED VALUES, ADD RIVER STA AND SAVE
        merged_culv_ratios_lab = culv_scour_label_list + culv_overtop_label_list
        merged_culv_ratios_df = cul_df_piv_all_sub_piv[merged_culv_ratios_lab]

        # ADD THE COLUMN RSTA AND CONVERT THE INDEX TO INT
        merged_culv_ratios_df["RSta"] = merged_culv_ratios_df.index
        merged_culv_ratios_df = merged_culv_ratios_df.reset_index(drop=True)

        # SAVE TRANSFORMED DATA WITH NEW FIELDS
        merged_culv_ratios_path = os.path.join(watershed_outputs_id, "culv_scour_overtop_ratios_only.csv")
        merged_culv_ratios_df.to_csv(merged_culv_ratios_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## RESHAPE POINTS FILE BY MERGING ALL STACKED POINTS (I.E. DIFFERENT CULVERTS AT 1 REACH) INTO ONE POINT
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # READ SHAPEFILE
        culv_shp = geopandas.read_file(culv_point)

        # DISSOLVE STACKED POINTS BY REACH ID
        culv_dissolved = culv_shp.dissolve(by="RIVER_STA")
        culv_dissolved["RIVER_STA"] = culv_dissolved.index

        # SAVE TRANSFORMED DATA WITH NEW FIELDS
        culv_dissolved_path = os.path.join(watershed_outputs_id,"stacked_culverts.shp")
        culv_dissolved.to_file(culv_dissolved_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'stacked_culverts'
        culv_dissolved.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT CULVERT CALCULATIONS TO A COPY OF RIVER STATIONS POINTS FILE
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # MAKE A COPY OF POINTS FILE
        culv_point_stack_df = geopandas.read_file(culv_dissolved_path)
        culv_point_stack_df = culv_point_stack_df[["RIVER_STA", "geometry"]]

        # JOIN CULVERT CALCULATIONS TO CULVERT POINTS SHAPEFILE
        culv_pts_joined = pd.merge(
            culv_point_stack_df,
            merged_culv_ratios_df,
            how="inner",
            left_on=["RIVER_STA"],
            right_on=["RSta"],
        )
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT CULVERT POINTS TO STREAM REACH
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # OPEN A COPY OF SEGMENTS CENTROID FILE
        gdf_centroid_c = geopandas.read_file(seg_centroid_path)
        gdf_segments_c = geopandas.read_file(
            stream_segments_attribute_df_path_c
        )
        gdf_hec45m_2_seg = geopandas.read_file(copy_bridgefinal)

        # FUNCTION TO FIND NEAREST SEGMENT'S CENTROID TO HECDATA POINT
        def nearest(
            row,
            geom_union,
            df1,
            df2,
            geom1_col="geometry",
            geom2_col="geometry",
            src_column=None,
        ):
            """Find the nearest point and return the corresponding
            value from specified column."""
            # FIND THE GEOMETRY THAT IS CLOSEST
            nearest = (
                df2[geom2_col] == nearest_points(row[geom1_col], geom_union)[1]
            )
            # GET THE CORRESPONDING VALUE FROM DF2 (MATCHING IS BASED ON THE GEOMETRY)
            value = df2.loc[nearest, src_column].iloc[0]
            return value

        # CREATE MULTIPOINT: CENTROID OF SEGMENTS
        unary_union = gdf_centroid_c.unary_union

        # FIND ID OF LINE CLOSEST TO HEC POINT
        culv_pts_joined["NearSegID"] = culv_pts_joined.apply(
            nearest,
            geom_union=unary_union,
            df1=culv_pts_joined,
            df2=gdf_centroid_c,
            src_column="JoinID",
            axis=1,
        )

        # ONLY 1 HEC POINT CAN BE ATTACHED TO A SEGMENT. CHOOSE ONLY 1 POINT PER SEGMENT.
        culv_pts_joined_sub = culv_pts_joined.drop_duplicates(
            subset="NearSegID", keep="first", inplace=False
        )

        # JOIN HEC POINTS TO RAW STREAM SEGMENTS, MUST DROP GEOMETRY BEFORE MERGING
        culv_pts_joined_sub = culv_pts_joined_sub.drop(columns="geometry")
        culv_2_seg = pd.merge(
            gdf_segments_c,
            culv_pts_joined_sub,
            how="left",
            left_on=["JoinID"],
            right_on=["NearSegID"],
        )

        culv_2_seg_path = os.path.join(watershed_outputs_id,"culv_joined2_segments.shp")
        culv_2_seg.to_file(culv_2_seg_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'culv_joined2_segments'
        culv_2_seg.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## JOIN CULVERT TO FINAL STREAM SEGMENTS
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # JOIN ONLY CULVERT CALCULATIONS TO STREAM SEGMENTS WITH OTHER CALCULATIONS
        culv_pts_joined_sub = culv_pts_joined_sub.drop(columns="RIVER_STA")
        culv_pts_joined_sub.rename(
            columns={"RSta": "RStaCul", "NearSegID": "NearSegCul"}, inplace=True
        )
        culv_2_seg_hec = pd.merge(
            gdf_hec45m_2_seg,
            culv_pts_joined_sub,
            how="left",
```

```
            left_on=["JoinID"],
            right_on=["NearSegCul"],
        )

        culv_2_seg_hec_path = os.path.join(watershed_outputs_id,"segment_dem_hec_bridge_culvert_prj.shp")
        culv_2_seg_hec.to_file(culv_2_seg_hec_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'segment_dem_hec_bridge_culvert_prj'
        culv_2_seg_hec.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## RENAME TO CULVERT STREAM POWER FINAL
## END OF CULVERT ANALYSIS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        gdf_culfinal = gpd.read_file(
            culv_2_seg_hec_path
        )  # READ THE ORIGINAL SHAPEFILE
        copy_culfinal = os.path.join(watershed_outputs_id,"Final_Culvert_Stream_Power_Output.shp")
        # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
        gdf_culfinal.to_file(copy_culfinal, driver="ESRI Shapefile")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'hec_points_joined2data.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'hec_points_within45m.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'hec45m_join2_segments.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'segment_dem_hec_connected_intp.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'segment_dem_hec_streampow_prj.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'segments_centroid.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'stream_segments_attribute.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'stream_segments_attribute_edited.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CREATING ONE SHAPEFILE OUT WITH ALL OF THE HECRAS FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # DEFINE THE FILE PATHS OF THE SHAPEFILES TO BE MERGED
    file_paths = [copy_strm_net]

    if hec_analysis == "Y":
        file_paths.append(copy_hecfinal)

    if bridge_analysis == "Y":
        file_paths.append(copy_bridgefinal)

    if culvert_analysis == "Y":
        file_paths.append(copy_culfinal)

    # READ IN THE FIRST SHAPEFILE TO INITIALIZE THE MERGED GEODATAFRAME
    merged_gdf = gpd.read_file(file_paths[0])

    # LOOP THROUGH EACH FILE PATH AND MERGE THE SHAPEFILE DATA
    for path in file_paths[1:]:
        if os.path.isfile(path):
            try:
                gdf = gpd.read_file(path)
                merged_gdf = pd.concat([merged_gdf, gdf], ignore_index=True, sort=False)
            except:
                print(f"Error reading file: {path}")
        else:
            print(f"File not found: {path}")

    # WRITE THE MERGED GEODATAFRAME TO A NEW SHAPEFILE
    allanalysisdata = str(watershed_outputs_id) + "\HECRAS_Data_Merged.shp"
    merged_gdf.to_file(allanalysisdata)
    print("Merged all HEC-RAS Outputs")

    # END1 = TIME.TIME()
    # PRINT("THE TIME OF EXECUTION OF HECRAS SINGLE IS :",(END1-START1), "SECONDS")
    print(counter)
    hec_analysis_file = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## END OF SINGLE HECRAS ANALYSIS
end2 = time.time()
print("The time of execution of HECRAS Single is :", (end2 - start2), "seconds")
gc.collect()
#################################################################################################################################
#################################################################################################################################
## MULTIPLE WATERSHED AND LAND USE SELECTED
start3 = time.time()

if check_boundary > 0 and lu_analysis == "Y":
    counter == 2

if counter == 2 and lu_analysis == "Y":
    print("LU STARTING FOR MULTIPLE WATERSHEDS")

    with rasterio.open(dem) as src:
        crs = src.crs

    # THIS IS THE CRS THAT EVERYTHING SHOULD BE
    proj_crs = crs

    ## CLIP DEM TO BASIN
    watershed_output = os.path.join(data_dir, "LU_Analysis")
    if os.path.exists(watershed_output):
        print("FOLDER CREATED")
    else:
        intermediate_dir = os.makedirs(watershed_output)

    wbt.set_working_dir(watershed_output)
    wbt.verbose = False

    if not os.path.exists(watershed_output):
        os.makedirs(watershed_output)

    lu_analysis_file = os.path.join(watershed_output, "LU_Analysis.gpkg")

    # CREATE A SHAPEFILE OUT OF THE USER PROVIDED SAMPLE POINTS
    # THIS VALUE IS USED IN THE EXTRACTION OF THE BASIN AND THE
```

```
# THRESHOLD VALUE DETERMINATION LATER ON
lupoints = pd.read_csv(lu_streampts)

geometry = [Point(xy) for xy in zip(lupoints['X'], lupoints['Y'])]
lupoints['geometry'] = geometry
lugdf = gpd.GeoDataFrame(lupoints, geometry='geometry')
lugdf.crs = crs
lusamplepoints = os.path.join(watershed_output, "LU_Userinput_Points.shp")
lugdf.to_file(lusamplepoints)

print("created a shapefile for the user provided location points")

# CHECK IF THE GEOPACKAGE ALREADY EXISTS
if os.path.exists(lu_analysis_file):
    print("GeoPackage already exists")
else:
    print("GeoPackage created:", lu_analysis_file)

# CONVERT BASIN TO POLYGON
basin_raster_path = str(data_dir) + str("/basins.tif")
basin_polygon_path = str(watershed_output) + str("/basins_polygon.shp")

mask = None
with rasterio.Env():  # UPDATED 17/6/2019 CHANGED RASTERIO.DRIVERS() TO RASTERIO.ENV()
    with rasterio.open(basin_raster_path) as src:
        image = src.read(1)  # FIRST BAND
        results = (
            {"properties": {"Value": v}, "geometry": s}
            for i, (s, v) in enumerate(
                shapes(image, mask=mask, transform=src.transform)
            )
        )  # UPDATED 17/6/2019 CHANGED SRC.AFFINE TO SRC.TRANSFORM
geoms = list(results)
polygonized_basin_df = geopandas.GeoDataFrame.from_features(geoms)
polygonized_basin_df.to_file(basin_polygon_path, driver="ESRI Shapefile")

overallbreached_dem = os.path.join(data_dir, "breached_dem.tif")

# READ THE BASIN SHAPEFILE
basin_gdf = gpd.read_file(basin_polygon_path)

# ENSURE CRS IS SET FOR BASIN_GDF
if basin_gdf.crs is None:
    print("CRS for basin_gdf is not set. Setting default CRS.")
    basin_gdf.set_crs(default_crs, inplace=True)
else:
    print(f"CRS for basin_gdf is set to {basin_gdf.crs}")


# FILTER THE BASINS BASED ON A SPECIFIC FIELD VALUE
filtered_basin_gdf = basin_gdf[basin_gdf["Value"] > 0]
print(f"Filtered basin count: {len(filtered_basin_gdf)}")

# PERFORM INTERSECTION WITH THE OTHER POLYGON SHAPEFILE
clippingboundary = gpd.read_file(boundary)
print(f"Clipping boundary count: {len(clippingboundary)}")


# ENSURE BOTH GEODATAFRAMES USE THE SAME CRS
if filtered_basin_gdf.crs != clippingboundary.crs:
    clippingboundary = clippingboundary.to_crs(filtered_basin_gdf.crs)

# PERFORM INTERSECTION
intersecting_basin_gdf = gpd.overlay(filtered_basin_gdf, clippingboundary, how='intersection')
print(f"Intersecting basin count: {len(intersecting_basin_gdf)}")

# CHECK IF INTERSECTING_BASIN_GDF IS EMPTY
if intersecting_basin_gdf.empty:
    raise ValueError("No intersecting basins found. Please check your input data.")

    # DISSOLVE THE FEATURES IN THE BASIN SHAPEFILE
    dissolved_basin = intersecting_basin_gdf.unary_union

    # CHECK IF DISSOLVED_BASIN IS NONE
    if dissolved_basin is None or dissolved_basin.is_empty:
        raise ValueError("Dissolved basin is empty. No valid intersections found.")

    # CREATE A GEODATAFRAME WITH THE DISSOLVED BASIN
    dissolved_basin_gdf = gpd.GeoDataFrame(
        geometry=[dissolved_basin], crs=intersecting_basin_gdf.crs
    )

    # OUTPUT PATH FOR THE DISSOLVED BASIN SHAPEFILE
    basin_polygon_sub_path = os.path.join(watershed_output, "basin_selected.shp")

    # SAVE THE DISSOLVED BASIN SHAPEFILE
    dissolved_basin_gdf.to_file(basin_polygon_sub_path, "ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'basin_selected'
    dissolved_basin_gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    dem_clipped_path = os.path.join(watershed_output, "breached_dem_clipped.tif")

    with rasterio.open(overallbreached_dem) as src:
        # READ THE RASTER DATA
        raster_data = src.read()

        # OPEN THE SHAPEFILE OR POLYGON FILE
        with fiona.open(basin_polygon_sub_path, "r") as shapefile:
            # EXTRACT THE GEOMETRY OF THE POLYGON
            geometry = [feature["geometry"] for feature in shapefile]

            # PERFORM THE CLIPPING OPERATION
            clipped_data, clipped_transform = rasterio.mask.mask(src, geometry, crop=True)

            # UPDATE THE METADATA
            clipped_meta = src.meta
            clipped_meta.update({"transform": clipped_transform, "height": clipped_data.shape[1], "width": clipped_data.shape[2], "count":
clipped_data.shape[0]})

            # WRITE THE CLIPPED DATA TO A NEW RASTER FILE
            with rasterio.open(dem_clipped_path, "w", **clipped_meta) as dst:
                dst.write(clipped_data)

    # CHECK IF THE OUTPUT FILE WAS CREATED
    if os.path.isfile(dem_clipped_path):
        print("Saved the Breached DEM Clipped")
    else:
        print("Failed to clip the raster to the polygon.")
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## GENERAL WORKFLOW GET FLOW DIRECTION (D8 METHOD)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    gen_flow_dir = os.path.join(watershed_output, "flow_dir.tif")
    wbt.d8_pointer(dem_clipped_path, gen_flow_dir)

    # GET FLOW ACCUMULATION
    gen_flow_accum = os.path.join(watershed_output, "flow_accum.tif")
    wbt.d8_flow_accumulation(dem_clipped_path, gen_flow_accum, out_type="cells")

    # EXTRACT STREAMS BASED ON DRAINAGE THRESHOLD
    print("Pruning Threshold:",pruningda)

    if pruningda == None:
        gen_extr_strm = os.path.join(watershed_output, "extract_stream_initial.tif")
        wbt.extract_streams(gen_flow_accum, gen_extr_strm, initial_dathres)
    else:
        gen_extr_strm = os.path.join(watershed_output, "extract_stream_usprune.tif")
        wbt.extract_streams(gen_flow_accum, gen_extr_strm, (pruningda/cell_size))
```

```
# GET STREAM VECTOR FROM RASTER STREAM
gen_strm_vec = os.path.join(watershed_output, "stream_vector_initial.shp")
wbt.raster_streams_to_vector(gen_extr_strm, gen_flow_dir, gen_strm_vec)

# WRITE TO A GEOPACKAGE
layer_name = 'stream_vector_initial'
gdf = gpd.read_file(gen_strm_vec)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

initial_dem_points_path = os.path.join(watershed_output, "dem_points_initial.shp")
wbt.raster_to_vector_points(gen_extr_strm, initial_dem_points_path)
# SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
initial_dem_points = geopandas.read_file(initial_dem_points_path)

# WRITE TO A GEOPACKAGE
print("Writing DEM Points Initial Output")
layer_name = 'dem_points_initial'
initial_dem_points.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

initial_stream_point_path = os.path.join(
    watershed_output, "stream_points_initial.shp"
)
initial_dem_points[initial_dem_points["VALUE"] == 1].to_file(
    initial_stream_point_path, driver="ESRI Shapefile"
)
# WRITE TO A GEOPACKAGE
print("Writing Stream Points Initial Output")
layer_name = 'stream_points_initial'
gdf = gpd.read_file(initial_stream_point_path)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## STREAM PRUNING: REDEFINE STREAM THRES BASED ON MIN FLOW ACCUM OF HEC POINTS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
# EXTRACT FLOW ACCUMULATION TO STREAM POINTS
wbt.extract_raster_values_at_points("flow_accum.tif", initial_stream_point_path)

strm_point_faccum_gdf = geopandas.read_file(initial_stream_point_path)
before_last = len(strm_point_faccum_gdf.columns) - 2
name_map = {strm_point_faccum_gdf.columns[before_last]: "Faccum"}
strm_point_faccum_gdf.rename(columns=name_map, inplace=True)

# OPEN LU SAMPLE POINTS TO CREATE BUFFER AROUND IT
lu_pts_buffer_gdf = geopandas.read_file(lusamplepoints)
lu_pts_buffer_gdf["geometry"] = lu_pts_buffer_gdf.geometry.buffer(50)

lu_pts_buffer_gdf = lu_pts_buffer_gdf.rename(columns={"FID": "RIVER_STA"})

buffers_lu_path = os.path.join(watershed_output, "lupt_buffers50m.shp")
lu_pts_buffer_gdf.to_file(buffers_lu_path, driver="ESRI Shapefile")
print("Writing LU Buffers Output")

# WRITE TO A GEOPACKAGE
layer_name = 'lupt_buffers50m'
gdf = gpd.read_file(buffers_lu_path)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

print("Spatial Join")

chunk_size = 500  # ADJUST THE CHUNK SIZE AS PER YOUR MEMORY LIMITATIONS

# DIVIDE THE GEODATAFRAMES INTO SMALLER SUBSETS AND PERFORM THE SPATIAL JOIN
point_in_buffer = gpd.GeoDataFrame()

for i, chunk_left in enumerate(lu_pts_buffer_gdf.groupby(np.arange(len(lu_pts_buffer_gdf)) // chunk_size)):
    chunk_right = strm_point_faccum_gdf  # ASSUMING RIGHT_DF IS THE OTHER GEODATAFRAME FOR THE JOIN

    # PERFORM SPATIAL JOIN ON THE CURRENT SUBSET
    chunk_join = gpd.sjoin(chunk_left[1], chunk_right, how='inner')

    # CONCATENATE THE CURRENT JOIN RESULT TO THE FINAL RESULT
    point_in_buffer = gpd.GeoDataFrame(pd.concat([point_in_buffer, chunk_join], ignore_index=True))

# DISPLAY THE COLUMN NAMES
column_names = point_in_buffer.columns.tolist()
print("Column names:", column_names)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## BRANCHING
## ------------------------------------------------------------------------------------------------------------------------------------------------------
# GROUP BY "RIVER_STA" AND CALCULATE THE MAXIMUM "FACCUM" AT EACH RIVER STATION
point_in_buffer_max = point_in_buffer.groupby("RIVER_STA")["Faccum"].apply(lambda x: x.max())

if pruningda == None:
    #PLOT FOR ALL OF THE POINTS NOT SEPERATED BY REACH ID
    point_in_buffer = point_in_buffer.dropna()
    point_in_buffer = point_in_buffer[point_in_buffer['Faccum'] != 0]

    # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA
    x_coordinates = point_in_buffer['RIVER_STA']
    flow_accumulation = point_in_buffer['Faccum']

    # PLOT THE X-COORDINATE VS. FLOW ACCUMULATION
    plt.scatter(x_coordinates, flow_accumulation)

    # PERFORM LINEAR REGRESSION
    slope, intercept, r_value, p_value, std_err = linregress(x_coordinates, flow_accumulation)
    # ADD A REGRESSION LINE TO THE PLOT
    plt.plot(x_coordinates, intercept + slope * x_coordinates, color='red', label='Regression Line')

    print(intercept + slope * x_coordinates)
    print("Intercept:", intercept)
    print(x_coordinates, flow_accumulation)

    # CUSTOMIZE THE PLOT
    plt.xlabel('River Station')
    plt.ylabel('Flow Accumulation')
    #PLT.TEXT(0.1, 0.9, F'INTERCEPT: {INTERCEPT}', TRANSFORM=PLT.GCA().TRANSAXES, FONTSIZE=7)
    plt.title(f'X-coordinate vs. Flow Accumulation for all Points')
    plt.legend()

    # SAVE THE PLOT AS AN IMAGE FILE
    plot_filename = os.path.join(watershed_output,"LUAll_Reaches.tif")
    plt.savefig(plot_filename, format="tif")

    # SHOW OR SAVE THE PLOT
    plt.show()

    # CREATE AN EMPTY DATAFRAME TO STORE THE MAXIMUM FLOW ACCUMULATIONS
    min_flow_table = pd.DataFrame(columns=['REACH', 'Min_Flow_Accumulation'])

    # PLOTS FOR EACH INDIVIDUAL REACH ID
    # EXTRACT UNIQUE REACH NUMBERS FROM YOUR DATA
    unique_reach_numbers = point_in_buffer['REACH'].unique()

    # PRINT THE UNIQUE REACH NUMBERS FOR REFERENCE
    print("Unique REACH Numbers:", unique_reach_numbers)

    # LOOP THROUGH EACH UNIQUE REACH NUMBER
    for reach_number in unique_reach_numbers:
        # FILTER DATA BASED ON THE CURRENT REACH NUMBER
        filtered_data = point_in_buffer[point_in_buffer['REACH'] == reach_number]

        # FILTER OUT DATA POINTS WITH ZERO FLOW ACCUMULATION
        filtered_data = filtered_data[filtered_data['Faccum'] != 0]

        # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING
        if not filtered_data.empty:

            # CALCULATE THE MAXIMUM FLOW ACCUMULATION WITHIN THE FILTERED DATA FOR EACH RIVER_STA
            max_flow_per_station = filtered_data.groupby("RIVER_STA")["Faccum"].max().reset_index()
```

```
        # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING WITHIN EACH RIVER_STA GROUP
            if not max_flow_per_station.empty:
                # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA FROM MAX_FLOW_PER_STATION
                x_coordinates = max_flow_per_station['RIVER_STA'].values.reshape(-1, 1)
                flow_accumulation = max_flow_per_station['Faccum']

                # CREATE A NEW PLOT FOR EACH REACH NUMBER
                plt.figure()
                plt.scatter(x_coordinates, flow_accumulation)

                # CREATE A LINEAR REGRESSION MODEL
                regr = linear_model.LinearRegression()
                regr.fit(x_coordinates, flow_accumulation)

                # MAKE PREDICTIONS USING THE X FOR LINEAR LINE
                Y_pred = regr.predict(x_coordinates)

                # ASSIGN THE Y_PRED VALUES TO A NEW COLUMN 'Y_PRED' IN MAX_FLOW_PER_STATION
                max_flow_per_station['Y_pred'] = Y_pred

                # REPLACE NEGATIVE VALUES IN COLUMN 'Y_PRED' WITH VALUES FROM COLUMN 'FACCUM'
                mask = max_flow_per_station['Y_pred'] < 0
                max_flow_per_station.loc[mask, 'Y_pred'] = max_flow_per_station.loc[mask, 'Faccum']

                # PLOT FLOW ACCUMULATION AGAINST RIVER STATION
                plt.scatter(x_coordinates, flow_accumulation, color="black")
                # ADD A REGRESSION LINE TO THE PLOT
                plt.plot(x_coordinates, Y_pred, color='red', label='Regression Line')

                # CUSTOMIZE THE PLOT
                plt.xlabel('River Station')
                plt.ylabel('Flow Accumulation')
                plt.title(f'X-coordinate vs. Flow Accumulation for REACH {reach_number}')
                plt.legend()

                # SAVE THE PLOT AS AN IMAGE FILE
                plot_filename = os.path.join(watershed_output, f"lureach_{reach_number}_plot.tif")
                plt.savefig(plot_filename, format="tif")

                # SHOW THE CURRENT PLOT
                plt.show()

            #MINIMUM FLOW ACCUMULATION OF THE REACH
            print(f'{reach_number}',max_flow_per_station)
            min_flow_accumulation = max_flow_per_station['Y_pred'].min()
            print(f'{reach_number}', min_flow_accumulation)

            # APPEND THE RESULT TO THE MAX_FLOW_TABLE DATAFRAME USING CONCAT
            min_flow_table = pd.concat([min_flow_table, pd.DataFrame({'REACH': [reach_number],'Min_Flow_Accumulation':
[min_flow_accumulation]})], ignore_index=True)

        # FIND THE MINIMUM FACCUM_MIN_YPRED
        min_faccum_min_ypred = min_flow_table["Min_Flow_Accumulation"].min()

        # CALCULATE FACCUM_THRES_25PCNT
        if min_faccum_min_ypred < 0:
            faccum_thres_25pcnt = initial_dathres_param
        else:
            faccum_thres_25pcnt = int(min_faccum_min_ypred)

        # PRINT THE RESULTS
        print("Result Table:")
        print(min_flow_table)
        path = os.path.join(watershed_output,"min_flow_table.csv")
        min_flow_table.to_csv(path, index=False)
        print("Minimum faccum_min_ypred:", min_faccum_min_ypred)
        print("faccum_thres_25pcnt:", faccum_thres_25pcnt)

    else:
        faccum_thres_25pcnt = round(pruningda/cell_size)
    print(faccum_thres_25pcnt)

# EXTRACT STREAM
wbt.extract_streams("flow_accum.tif", "extract_stream.tif", faccum_thres_25pcnt)
print("Streams have been extracted with a threshold applied")

# GET POINTS FROM RASTER STREAM BASED ON VALUE = 1
wbt.raster_to_vector_points("extract_stream.tif", "dem_points.shp")

# SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
dem_points_path = os.path.join(watershed_output,"dem_points.shp")
dem_points = geopandas.read_file(dem_points_path)

# WRITE TO A GEOPACKAGE
layer_name = 'dem_points'
gdf = gpd.read_file(gen_strm_vec)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

new_stream_point_path = os.path.join(watershed_output,"stream_points.shp")
dem_points[dem_points["VALUE"] == 1].to_file(
    new_stream_point_path, driver="ESRI Shapefile"
)
# WRITE TO A GEOPACKAGE
layer_name = 'stream_points'
gdf = gpd.read_file(new_stream_point_path)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

# GET STREAM VECTOR FROM RASTER STREAM
wbt.raster_streams_to_vector(
    "extract_stream.tif", "flow_dir.tif", "stream_vector_pruned.shp"
)

# CLIP STREAM VECTOR TO SUBWATERSHED ANALYSIS
if subwatershed_analysis == "Y":
    subwshed_df = geopandas.read_file(subwshed_poly)

    stream_vector_pruned_path = os.path.join(watershed_output,"stream_vector_pruned.shp")
    stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
    subwshed_stream_df = geopandas.sjoin(
        stream_vector_pruned_df, subwshed_df, op="intersects"
    )
    subwshed_stream_list = subwshed_stream_df["FID"].tolist()
    # SUBSET BY LIST OF LINES ONLY WITHIN SUBWATERSHED
    subwshed_stream_selected_df = stream_vector_pruned_df[
        stream_vector_pruned_df["FID"].isin(subwshed_stream_list)
    ]
    stream_vector_path = os.path.join(watershed_output,"stream_vector.shp")
    subwshed_stream_selected_df.to_file(stream_vector_path, driver="ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_vector'
    gdf = gpd.read_file(gen_strm_vec)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

else:
    stream_vector_pruned_path = os.path.join(watershed_output,"stream_vector_pruned.shp")
    stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
    stream_vector_path = os.path.join(watershed_output,"stream_vector.shp")
    stream_vector_pruned_df.to_file(stream_vector_path, driver="ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_vector_pruned'
    gdf = gpd.read_file(stream_vector_pruned_path)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_vector_pruned.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
```

```
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_vector'
        gdf = gpd.read_file(stream_vector_path)
        gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## EXTRACT ALL RASTER VALUES TO STREAM POINTS
## GET STREAM LINK CLASS
## CLASSIFICATION KEY
## VALUE  CLASS
## 1      EXTERIOR LINK
## 2      INTERIOR LINK
## 3      SOURCE NODE (HEADWATER)
## 4      LINK NODE (CONFLUENCE)
## 5      SINK NODE (OUTLET)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    wbt.stream_link_class("flow_dir.tif", "extract_stream.tif", "stream_link_class.tif")

    # GET UNIQUE ID FOR EACH STREAM LINK CLASS
    wbt.stream_link_identifier(
        "flow_dir.tif", "extract_stream.tif", "stream_link_id.tif"
    )

    # GET LENGTH FOR EACH STREAM LINK CLASS
    wbt.stream_link_length(
        "flow_dir.tif", "stream_link_id.tif", "stream_link_length.tif"
    )

    # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
    wbt.tributary_identifier("flow_dir.tif", "extract_stream.tif", "tributary_id.tif")

    # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
    wbt.strahler_stream_order(
        "flow_dir.tif", "extract_stream.tif", "strahler_order.tif"
    )

    # GET DISTANCE TO OUTLET
    wbt.distance_to_outlet(
        "flow_dir.tif", "extract_stream.tif", "distance_to_outlet.tif"
    )

    # GET THE DISTANCE TO THE FURTHEST UPSTREAM CHANNEL HEAD FOR EACH STREAM CELL.
    wbt.farthest_channel_head(
        "flow_dir.tif", "extract_stream.tif", "distance_to_headwater.tif"
    )

    # GET STREAM SLOPE ALONG RASTER STREAM
    # WBT.STREAM_SLOPE_CONTINUOUS("FLOW_DIR.TIF", "STREAM_LINK_ID.TIF", DEM , "STREAM_SLOPE.TIF")

    # EXTRACT ALL RASTER VALUES TO SELECTED STREAM POINTS
    wbt.extract_raster_values_at_points(

"distance_to_headwater.tif;distance_to_outlet.tif;stream_link_class.tif;stream_link_id.tif;stream_link_length.tif;tributary_id.tif;strahler_order.tif;f
low_accum.tif",
        new_stream_point_path,
    )

    # EXTRACT RASTER DEM VALUES TO STREAM POINTS
    src_filename = dem
    shp_filename = new_stream_point_path

    src_ds = gdal.Open(src_filename)
    gt = src_ds.GetGeoTransform()
    rb = src_ds.GetRasterBand(1)

    ds = ogr.Open(shp_filename, 1)
    lyr = ds.GetLayer()
    lyr.CreateField(ogr.FieldDefn("Elev", ogr.OFTReal))
    for feat in lyr:
        geom = feat.GetGeometryRef()
        mx, my = geom.GetX(), geom.GetY()  # COORD IN MAP UNITS

        # CONVERT FROM MAP TO PIXEL COORDINATES.
        # ONLY WORKS FOR GEOTRANSFORMS WITH NO ROTATION.
        px = int((mx - gt[0]) / gt[1])  # X PIXEL
        py = int((my - gt[3]) / gt[5])  # Y PIXEL

        intval = rb.ReadAsArray(px, py, 1, 1)
        elev = intval[0]
        for value in elev:
            v = float(value)
            # PRINT(V)

        # UPDATE NEW FIELD
        lyr.SetFeature(feat)
        feat.SetField("Elev", v)
        lyr.SetFeature(feat)

    # RENAME FIELDS
    stream_point = geopandas.read_file(new_stream_point_path)

    name_map = {
        stream_point.columns[2]: "DistUp",
        stream_point.columns[3]: "DistDown",
        stream_point.columns[4]: "LinkClass",
        stream_point.columns[5]: "LinkID",
        stream_point.columns[6]: "LinkLength",
        stream_point.columns[7]: "TribID",
        stream_point.columns[8]: "Strahler",
        stream_point.columns[9]: "FlowAccum",
        stream_point.columns[10]: "Elev",
    }
    stream_point.rename(columns=name_map, inplace=True)

    # GET XY COORDINATES
    stream_point["X"] = stream_point.geometry.x
    stream_point["Y"] = stream_point.geometry.y
    stream_point["pXY"] = stream_point["X"].map(str) + "," + stream_point["Y"].map(str)

    # SAVE AS SHAPEFILE
    stream_point_field_rename_path = os.path.join(watershed_output,"stream_points_fields_renamed.shp")
    stream_point.to_file(stream_point_field_rename_path, driver="ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_points_fields_renamed'
    gdf = gpd.read_file(stream_point_field_rename_path)
    stream_point.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')
    print("stream points fields renamed shapefile created ")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## STREAM SEGMENTATION
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    print("stream segmentation starting")
    stream_point = os.path.join(watershed_output,"stream_points.shp")
    stream_vector = os.path.join(watershed_output,"stream_vector.shp")

    # CREATE FUNCTION TO SPLIT LINE AT POINTS
    def split_line_at_points(line, point):
        gdf_line = line
        gdf_points = point

        # JOIN ALL LINE AND POINTS
        line = gdf_line.geometry.unary_union
        coords = gdf_points.geometry.unary_union

        # SNAP AND SPLIT COORDS ON LINE
        # RETURNS GEOMETRYCOLLECTION
        split_line = shapely.ops.split(line, coords)

        # CHECK IF SPLIT_LINE IS A GEOMETRYCOLLECTION
```

```
    if isinstance(split_line, shapely.geometry.collection.GeometryCollection):
        # ITERATE OVER INDIVIDUAL GEOMETRIES IN THE COLLECTION
        segments = [feature for feature in split_line.geoms]
    else:
        segments = [split_line]

    # TRANSFORM GEOMETRY COLLECTION TO GEODATAFRAME
    gdf_segments = geopandas.GeoDataFrame(
        list(range(len(segments))), geometry=segments
    )
    gdf_segments.columns = ["index", "geometry"]

    # SAVE SEGMENTS
    stream_segments = os.path.join(watershed_output,"stream_segments.shp")
    gdf_segments.to_file(stream_segments, driver="ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments'
    gdf = gpd.read_file(stream_segments)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # RETURN GDF_SEGMENTS

split_line_at_points(gpd.read_file(stream_vector), gpd.read_file(stream_point))
# 3 MINS PROCESSING TIME

# WRITE TO A GEOPACKAGE
layer_name = 'stream_vector'
gdf = gpd.read_file(stream_vector)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

# DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
file_pattern = os.path.join(watershed_output, 'stream_vector.*')
file_list = glob.glob(file_pattern)
for file_path in file_list:
    os.remove(file_path)

# WRITE TO A GEOPACKAGE
layer_name = 'stream_points'
gdf = gpd.read_file(stream_point)
gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## GET NODES OF SEGMENTS AND JOIN POINT ATTRIBUTES
## ------------------------------------------------------------------------------------------------------------------------------------------------------
print("getting nodes of segments and joining point attributes")
# GET LINE NODES (START AND END) USING WHITEBOX TO MATCH POINT NODES
stream_segments = os.path.join(watershed_output,"stream_segments.shp")
stream_segment_nodes = os.path.join(watershed_output,"stream_segment_nodes.shp")
wbt.extract_nodes(stream_segments, stream_segment_nodes)

# CREATE JOIN IDENTIFIER - FIELD "INDEX" (PLUS 1) OF STREAM SEGMENT IS FIELD "PARENTID" OF STREAM SEGMENT
NODE.
segment_attribute = ogr.Open(stream_segments, 1)
segment_lyr = segment_attribute.GetLayer()
segment_lyr.CreateField(ogr.FieldDefn("JoinID", ogr.OFTInteger))

for feat in segment_lyr:
    # GET INDEX VALUE
    feat_v = feat.GetField("index")
    new_v = feat_v + 1
    # UPDATE NEW FIELD
    segment_lyr.SetFeature(feat)
    feat.SetField("JoinID", new_v)
    segment_lyr.SetFeature(feat)

# JOIN STREAM POINTS DATA TO SEGMENTS' NODES
stream_point_field_rename_df = geopandas.read_file(stream_point_field_rename_path)
stream_segment_node_df = geopandas.read_file(stream_segment_nodes)
stream_point_join2_segment_node = geopandas.sjoin(
    stream_segment_node_df,
    stream_point_field_rename_df,
    how="inner",
    op="intersects",
)
stream_segment_join2_point_path = os.path.join(watershed_output,"stream_point_join2_segment_node.shp"
)
stream_point_join2_segment_node.to_file(
    stream_segment_join2_point_path, driver="ESRI Shapefile"
)

# USE DISTANCE FROM OUTLET TO FIND THE COORDINATES OF START NODES AND END NODES OF SEGMENTS. SAVE
THE END COORDINATES IN THE SAME ROW AS THE START COORDINATES.
stream_segment_join2_point_df = geopandas.read_file(stream_segment_join2_point_path)

# WRITE TO A GEOPACKAGE
layer_name = 'stream_point_join2_segment_node'
stream_segment_join2_point_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

# DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
file_pattern = os.path.join(watershed_output, 'stream_point_join2_segment_node.*')
file_list = glob.glob(file_pattern)
for file_path in file_list:
    os.remove(file_path)

# WRITE TO A GEOPACKAGE
layer_name = 'stream_segment_nodes'
stream_segment_node_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

# DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
file_pattern = os.path.join(watershed_output, 'stream_segment_nodes.*')
file_list = glob.glob(file_pattern)
for file_path in file_list:
    os.remove(file_path)

stream_segment_join2_point_df["X"] = stream_segment_join2_point_df.geometry.x
stream_segment_join2_point_df["Y"] = stream_segment_join2_point_df.geometry.y
stream_segment_join2_point_df["StartXY"] = (
    stream_segment_join2_point_df["X"].map(str)
    + ","
    + stream_segment_join2_point_df["Y"].map(str)
)

# FIND MIN DISTANCE FROM OUTLET (I.E. LOWER NODE) AND UPDATE NEW COLUMN WITH ENDXY COORDINATES.
stream_segment_join2_point_df[
    "DistDown_min"
] = stream_segment_join2_point_df.groupby(["PARENT_ID"])["DistDown"].transform(min)

# SUBSET DATA BY EXTRACTING START NODES
stream_segment_join2_point_df_startnodes = stream_segment_join2_point_df[
    stream_segment_join2_point_df["DistDown_min"]
    != stream_segment_join2_point_df["DistDown"]
]
stream_segment_join2_point_df_endnodes = stream_segment_join2_point_df[
    stream_segment_join2_point_df["DistDown_min"]
    == stream_segment_join2_point_df["DistDown"]
]
stream_segment_join2_point_df_endnodes.loc[
    :, "EndXY"
] = stream_segment_join2_point_df_endnodes["StartXY"]

stream_segment_join2_point_df_subset = pd.merge(
    stream_segment_join2_point_df_startnodes,
    stream_segment_join2_point_df_endnodes[["PARENT_ID", "EndXY"]],
    how="left",
    on="PARENT_ID",
)

# MATCH THE ELEV VALUE FROM THE STREAM POINTS FOR BOTH NODES OF STREAM SEGMENTS
```

```
    stream_segment_join2_point_df_subset[
        "EndElev"
    ] = stream_segment_join2_point_df_subset["EndXY"].map(
        stream_point_field_rename_df.set_index("pXY")["Elev"]
    )
    stream_segment_join2_point_df_subset[
        "StartElev"
    ] = stream_segment_join2_point_df_subset["StartXY"].map(
        stream_point_field_rename_df.set_index("pXY")["Elev"]
    )

    stream_segment_join2_point_df_subset_path = os.path.join(watershed_output,"stream_point_join2_segment_node_subset.shp")
    stream_segment_join2_point_df_subset.to_file(
        stream_segment_join2_point_df_subset_path, driver="ESRI Shapefile"
    )
    print("stream segment join2 point complete")
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## JOIN STREAM SEGMENT NODES TO STREAM SEGMENT BASED ON PARENT ID
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    stream_segments_df = geopandas.read_file(stream_segments)
    stream_segment_join2_point_df_subset_df = geopandas.read_file(
        stream_segment_join2_point_df_subset_path
    )

    # MUST DROP GEOMETRY OF POINT TO ALLOW FILE TO BE SAVED IN THE FUTURE
    stream_segment_join2_point_df_subset_df = (
        stream_segment_join2_point_df_subset_df.drop("geometry", axis=1)
    )

    # STREAM_SEGMENTS_ATTRIBUTE_DF =
STREAM_SEGMENTS_DF.JOIN(STREAM_SEGMENT_JOIN2_POINT_DF_SUBSET.SET_INDEX('PARENT_ID'), ON='JOINID',
RSUFFIX = "_P")
    stream_segments_attribute_df = stream_segments_df.merge(
        stream_segment_join2_point_df_subset_df,
        left_on="JoinID",
        right_on="PARENT_ID",
        how="left",
    )

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_point_join2_segment_node_subset'
    stream_segment_join2_point_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_point_join2_segment_node_subset.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments'
    gdf = gpd.read_file(stream_segments)
    gdf.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE DEM SLOPE, DRAINAGE AREA, DISCHARGE, AND WIDTH (RURAL SCENARIO)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    print("calculating dem slope , drainage area, discharge, and width")
    # CALCULATE LENGTH OF SEGMENTS
    stream_segments_attribute_df["Length"] = stream_segments_attribute_df.length

    # CALCULATE SLOPE USING DEM ELEVATION
    stream_segments_attribute_df["Slope"] = (
        (
            stream_segments_attribute_df["EndElev"]
            - stream_segments_attribute_df["StartElev"]
        )
        / (stream_segments_attribute_df["Length"])
    ) * -1

    # CALCULATE DRAINAGE AREA (DA = (FLOW ACCUMULATION/1000000)*100
    stream_segments_attribute_df["DA_km2"] = (
        stream_segments_attribute_df["FlowAccum"] / 1000000
    ) * cell_size

    # CALCULATE RURAL DISCHARGE USING Q=CA^X WHERE C AND X ARE COEFFICCIENTS AND A IS THE DRAINAGE AREA
    stream_segments_attribute_df["Q_m3pers"] = rQ_a * (
        stream_segments_attribute_df["DA_km2"] ** rQ_b
    )

    # CALCULATE WIDTH (EMPIRICAL - DA WIDTH)
    stream_segments_attribute_df["RurWidth"] = rW_a * (
        stream_segments_attribute_df["DA_km2"] ** rW_b
    )
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## SMOOTH DEM SLOPE
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    print("Smoothing dem slope")

    # SET INDEX FROM HIGH ELEVATION TO LOW, I.E. BY DECREASING DISTANCE DOWNSTREAM
    stream_segments_attribute_df = stream_segments_attribute_df.sort_values(
        by=["DistDown"], ascending=False
    )
    stream_segments_attribute_df["SID"] = np.arange(len(stream_segments_attribute_df))
    stream_segments_attribute_df.set_index("SID", inplace=True)
    stream_segments_attribute_df["index"] = np.arange(len(stream_segments_attribute_df))

    # ADD NEW FIELDS TO STORE RESULTS
    stream_segments_attribute_df["deltaDA"] = None
    stream_segments_attribute_df["SavgID"] = None
    stream_segments_attribute_df["SavgRID"] = None
    stream_segments_attribute_df["SlopeAVGLU"] = None

    # CREATE SEGMENTS BLOCKS OVER WHICH TO SMOOTH SLOPE
    # CREATE LIST WITH ALL ELEVATION VALUES AND SORT IN DESCENDING ORDER. REMOVE ROWS WITHOUT
ATTRIBUTES
    stream_segments_attribute_df = stream_segments_attribute_df.dropna(
        axis=0, subset=["PARENT_ID"]
    )

    elev_set = set(stream_segments_attribute_df.apply(lambda row: (row["index"], row["EndElev"]), axis=1))

    elev_list_u = list(elev_set)
    elev_list = sorted(elev_list_u, key=lambda x: x[0])  # ACTIVATED

    # START FROM THE SEGMENT WITH HIGHEST ELEVATION, ASSIGN THE SAME SEGMENT ID WHILE TRACING
DOWNSTREAM OF THE REACH UNTIL THE DIFFERENCE RATIO IN DRAINAGE AREA IS >=0.1.
    seg_id = 0
    elev_list_count = elev_list

    while len(elev_list_count) != 0:
        elev_pair = elev_list_count[0]
        max_elev = elev_pair[1]
        max_id = elev_pair[0]
        seg_id = seg_id + 1
        del elev_list_count[0]

        # MAX_ELEV AND MAX_ID ONLY HAS ONE POSSIBLE MATCH, SKIP THE LOOP AND USE EXISTING INDEXES
        seg_r_id = 0
        start_endxy = stream_segments_attribute_df.loc[max_id, "EndXY"]
        start_da = stream_segments_attribute_df.loc[max_id, "DA_km2"]
        stream_segments_attribute_df.loc[max_id, "SavgID"] = seg_id
        stream_segments_attribute_df.loc[max_id, "SavgRID"] = seg_r_id

        if start_da is None or start_da == 0:
            start_da = 0.0

        # RATHER THAN LOOPING THROUGH MULTIPLE TIMES, QUERY THE DATAFRAME FOR THE MAIN CONDITION
        ssaIdxs = stream_segments_attribute_df.query(
            "@start_endxy == StartXY"
        ).index.tolist()
        # NOW LOOP THROUGH THE INITIAL MATCHES AND DO MATH
```

```
    while len(ssaIdxs) != 0:
        index = ssaIdxs[0]
        del ssaIdxs[0]
        r_da = stream_segments_attribute_df.loc[index, "DA_km2"]
        r_seg_id = stream_segments_attribute_df.loc[index, "SavgID"]
        if r_da == 0 or r_da == None:
            delta_da = 0.0
        else:
            diff_da = r_da - start_da
            delta_da = (diff_da) / r_da
        if delta_da < thres_da and r_seg_id == None:
            seg_r_id = seg_r_id + 1
            stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
            stream_segments_attribute_df.loc[index, "SavgRID"] = seg_r_id
            stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
            stream_segments_attribute_df.loc[index, "deltaDA"] = delta_da
            start_endxy = stream_segments_attribute_df.loc[index, "EndXY"]
            # FIND MATCHES FOR THE NEW START_ENDXY
            ssaIdxs = stream_segments_attribute_df.query(
                "@start_endxy == StartXY"
            ).index.tolist()
            # DELETE PAIR IN LIST IF EXIST
            r_elev = stream_segments_attribute_df.loc[index, "EndElev"]
            r_id = index
            pair = (r_id, r_elev)
            # PRINT PAIR
            if pair in elev_list_count:
                elev_list_count.remove(pair)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE ROLLING MEAN OF SEGMENT USING 25 REACHES UPSTREAM AND 25 REACHES DOWNSTREAM IN A SEGMENT
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    print("calculating rolling mean")

    # OPTIMIZED CODE
    seg_id_l_unique = stream_segments_attribute_df["SavgID"].unique().tolist()

    # LOOP THROUGH TABLE AND COLLECT REACHIDS
    avg_window = {}

    for seg_id in seg_id_l_unique:
        # DONT LOOP AND CHECK CONDITIONS, GET INDEXES THAT MATCH CONDITION INSTEAD THEN LOOP
        ssaIdxs = stream_segments_attribute_df.query(
            "@seg_id == SavgID"
        ).index.tolist()
        for index in ssaIdxs:
            seg_r_id = stream_segments_attribute_df.loc[index, "SavgRID"]
            slope_raw = stream_segments_attribute_df.loc[index, "Slope"]
            avg_window[seg_r_id] = slope_raw

        avg_window_df = pd.Series(avg_window)
        roll_avg_df = avg_window_df.rolling(
            window=Savg_cell_window, min_periods=1, center=True
        ).mean()

        # LOOP THROUGH SET AND CALCULATE ROLLING MEAN
        for i, j in roll_avg_df.items():
            seg_r_id_avg = i
            avg_value = j
            ssaIdxs = stream_segments_attribute_df.query(
                "(@seg_id == SavgID) & (@seg_r_id_avg == SavgRID)"
            ).index.tolist()
            for index in ssaIdxs:
                stream_segments_attribute_df.loc[index, "SlopeAVGLU"] = abs(avg_value)

    # MUST FILL NULL SLOPE WITH 0
    stream_segments_attribute_df.fillna(value=0, inplace=True)

    stream_segments_smoothedslope_df_path = os.path.join(watershed_output,"stream_segments_SAVGslope.shp")
    stream_segments_attribute_df.to_file(
        stream_segments_smoothedslope_df_path, driver="ESRI Shapefile"
    )

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments_SAVGslope'
    stream_segments_attribute_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_segments_SAVGslope.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)

    print("SAVG Written")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE RURAL STREAM POWER
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    print("calculating rural stream power")

    # CALCULATE TOTAL STREAM POWER (SP) AND SPECIFIC STREAM POWER (SSP)
    stream_segments_attribute_df["RuralSP"] = pd.to_numeric(
        9810
        * stream_segments_attribute_df["SlopeAVGLU"]
        * stream_segments_attribute_df["Q_m3pers"],
        errors='coerce'
    )

    # RURAL SSP (EMPIRICAL FORMUL - DA SSP)
    stream_segments_attribute_df["RuralSSP"] = pd.to_numeric(
        stream_segments_attribute_df["RuralSP"] / stream_segments_attribute_df["RurWidth"],
        errors='coerce'
    )

    # FIND THE MAXIMUM RURAL WIDTH OLD METHOD
    max_val_oldmethod = stream_segments_attribute_df["RurWidth"].max()

    # # CALCULATE D84
    # # EQUATION 16 FROM FERGUSON (2005) ASSUMPTION: DI = DB
    # # CONSTANTS FOR D84 PREDICTION
    gamma = 9790 # N/M3
    kappa = 0.41 # CONSTANT FROM FERGUSON 2005 - WORKS FOR D IN MM, POWER IN W/M2
    R = 1.65 # SUBMERGED SPECIFIC GRAVITY
    rho = 1000 # DENSITY OF WATER
    theta_cb = 0.045 # SHEAR STRESS THRESHOLD
    m =2.80 # ROUGHNESS MULTIPLIER FOR D84 FROM LOPEZ AND BARRANGAN E.G. HEY 1979
    gee = 9.81 #ACCELERATION BY GRAVITY

    # C = LOG10(30*THETA_CB*R/(E*M*!S_MPERM!))"
    # EXPRESSIONDB =
"((KAPPA*!SPOWER_WPERM!/(2.30*RHO*LOG10(30*THETA_CB*R/(EXP*M*!S_MPERM!))))**(2/3))/(THETA_CB*R*GEE)"
    stream_segments_attribute_df["SlopeAVGLU"] = pd.to_numeric(stream_segments_attribute_df["SlopeAVGLU"], errors='coerce')
    stream_segments_attribute_df["eqlogC"] = np.log10(30 * theta_cb * R / (np.exp(1) * m *
np.abs(stream_segments_attribute_df["SlopeAVGLU"])))
    stream_segments_attribute_df["D84_mm"] = ((((kappa * stream_segments_attribute_df["RuralSSP"]) / (2.30 * rho *
stream_segments_attribute_df["eqlogC"])) ** (2.0 / 3)) / (theta_cb * R * gee)) * rho

    # SAVE FILE, MUST FILL NULL WITH 0 OR THE COLUMN WILL BE IMPORTED AS A STRING
    stream_segments_attribute_df.fillna(value=0, inplace=True)

    stream_segments_attribute_df_path = os.path.join(watershed_output,"stream_segments_attribute.shp")
    stream_segments_attribute_df.to_file(
        stream_segments_attribute_df_path, driver="ESRI Shapefile"
    )

    # SAVE A COPY OF SEGMENTS FOR MANIPULATION
    stream_segments_attribute_df_path_c = os.path.join(watershed_output,"stream_segments_attribute_edited.shp")
    stream_segments_attribute_df.to_file(
        stream_segments_attribute_df_path_c, driver="ESRI Shapefile"
    )
```

```
    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments_attribute'
    stream_segments_attribute_df.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_segments_attribute.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    print("Rural stream power calculation ")
    gdf_strm = gpd.read_file(
        stream_segments_attribute_df_path_c
    )  # READ THE ORIGINAL SHAPEFILE

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_segments_attribute_edited'
    gdf_strm.to_file(lu_analysis_file, layer=layer_name, driver='GPKG')

    copy_strm_net = (
        str(watershed_output) + "\Final_Rural_Stream_Power_Output.shp"
    )  # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
    gdf_strm.to_file(copy_strm_net, driver="ESRI Shapefile")

## ---------------------------------------------------------------------------------------------------------------------------------------------------
## END OF RURAL STREAM POWER CALCULATION
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## WRITE SHAPEFILES TO A GEOPACKAGE TO SAVE SPACE
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_points_initial.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_segments_attribute.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_vector_initial.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'dem_points.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_vector.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'hecpt_buffers50m_join.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'dem_points_initial.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'hecpt_buffers50m.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)
    with rasterio.open(dem) as src:
        x_res, y_res = src.res

    # CREATE A FOLDER TO STORE INTERMEDIATE FILES
    intermediate_storage_files = os.path.join(watershed_output, "LU_Outputs")
    if os.path.exists(intermediate_storage_files):
        print("FOLDER EXISTS")
    else:
        intermediate_dir = os.makedirs(intermediate_storage_files)
        print("FOLDER CREATED")

    # IMPORT FLOW ACCUMULATION
    flow_accum = os.path.join(watershed_output, "flow_accum.tif")

    # IMPORT FLOW DIRECTION
    flow_dir = os.path.join(watershed_output, "flow_dir.tif")
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## PRJ FILES FOR ALL OF THE SHAPEFILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    # DEFINE WHAT THE CRS IS
    crs = pyproj.CRS.from_epsg(int(dem_espg2int))

    # WRITE THE CRS TO A .PRJ FILE
    # ORIGINAL DISCHARGE POINTS
    with open(
        os.path.join(watershed_output, "stream_points_fields_renamed.prj"), "w"
    ) as prj_file:
        prj_file.write(crs.to_wkt())
    data1 = gpd.read_file(stream_point_field_rename_path)
    data1 = data1.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA1.CRS)

    # STREAM_NET
    with open(
        os.path.join(watershed_output, "stream_segments_attribute_edited.prj"), "w"
    ) as prj_file:
        prj_file.write(crs.to_wkt())
    data2 = gpd.read_file(stream_segments_attribute_df_path_c)
    data2 = data2.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA2.CRS)

    # COPY OF STREAM_NET
    with open(
        os.path.join(watershed_output, "Final_Rural_Stream_Power_Output.prj"), "w"
    ) as prj_file:
        prj_file.write(crs.to_wkt())
    data3 = gpd.read_file(copy_strm_net)
    data3 = data3.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA3.CRS)

    with open(os.path.join(watershed_output, "basin_selected.prj"), "w") as prj_file:
        prj_file.write(crs.to_wkt())
    data4 = gpd.read_file(basin_polygon_sub_path)
    data4 = data4.set_crs("EPSG:" + str(dem_espg2int))
    # PRINT(DATA4.CRS)

    if lushapefileanalysis == "Y":
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## PREPROCESS THE INPUT RASTERS
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # RESOLUTION EXTRACTED AT TOP
        with rasterio.open(dem_clipped_path) as src:
            # GET THE AFFINE TRANSFORM FROM THE METADATA
            transform = src.transform

            # THE RESOLUTION IS STORED IN THE CELLSIZE OF THE AFFINE TRANSFORM
            x_resolution = transform.a
            y_resolution = transform.e

            # THE CELLSIZE IS THE ABSOLUTE VALUE OF THE RESOLUTION, AS IT MAY BE NEGATIVE IF THE RASTER IS
ROTATED
            x_cellsize = abs(x_resolution)
```

```
        y_cellsize = abs(y_resolution)

    crs = rasterio.crs.CRS.from_epsg(int(dem_espg2int))

    # ABSORPTION RASTER
    # PREPROCESS THE ABSORPTION RASTER, WE WANT TO HAVE ZERO ABSORPTION
    # OPEN THE RASTER FILE
    with rasterio.open(dem_clipped_path) as src:
        # READ THE DATA INTO A NUMPY ARRAY
        data = src.read(1)

        # SET ALL CELLS TO THE DESIRED VALUE
        data[:] = 0

        # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
        meta = src.meta.copy()
        meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

        absorption_file = os.path.join(intermediate_storage_files, "Absorption.tif")
        # WRITE THE UPDATED ARRAY TO A NEW FILE
        with rasterio.open(absorption_file, "w", **meta) as dst:
            dst.write(data, 1)

    # EFFICIENCY RASTER
    # PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY
    with rasterio.open(dem_clipped_path) as src:
        # READ THE DATA INTO A NUMPY ARRAY
        data = src.read(1)

        # SET ALL CELLS TO THE DESIRED VALUE
        data[:] = 1

        # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
        meta = src.meta.copy()
        meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

        efficiency_file = os.path.join(intermediate_storage_files, "Efficiency.tif")
        # WRITE THE UPDATED ARRAY TO A NEW FILE
        with rasterio.open(efficiency_file, "w", **meta) as dst:
            dst.write(data, 1)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CROP LAND USE SHAPEFILE TO DEM OUTLINE - TO BE USED IN THE WHITEBOX TOOLS FUNCTION
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # SHAPEFILE FILE
    lu_shapefile = gpd.read_file(lu_file)
    # READ SHAPEFILE
    crop_shapefile = gpd.read_file(basin_polygon_sub_path)
    buffered_crop = crop_shapefile.buffer(50)  # CREATES A BUFFER AROUND THE BOUNDARY

    crop_shapefile.crs = pyproj.CRS("EPSG:" + str(dem_espg2int))

    # CREATE AN EMPTY LIST TO HOLD THE CROPPED LAND USE POLYGONS
    cropped_lu_list = []

    # LOOP THROUGH EACH POLYGON IN THE CROP SHAPEFILE
    for poly in buffered_crop.geometry:
        # CROP THE LAND USE POLYGONS USING THE CURRENT CROP POLYGON
        cropped = gpd.overlay(
            lu_shapefile, gpd.GeoDataFrame(geometry=[poly]), how="intersection"
        )

        # APPEND THE CROPPED POLYGONS TO THE LIST
        cropped_lu_list.append(cropped)

    # CONCATENATE THE CROPPED POLYGONS INTO A SINGLE GEODATAFRAME
    cropped_lu = gpd.GeoDataFrame(
        pd.concat(cropped_lu_list, ignore_index=True), crs=lu_shapefile.crs
    )

    # WRITE THE CROPPED LAND USE POLYGONS TO A NEW SHAPEFILE
    lu_clipped = os.path.join(intermediate_storage_files, "LU_Clipped.shp")
    cropped_lu.to_file(lu_clipped)
    lu_clip = gpd.read_file(lu_clipped)
    # PRINT(LU_CLIP.CRS)
    print("LU Clip spatial extents:", lu_clip.total_bounds)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## ASSIGN IMPERVIOUSNESS VALUES - IF THERE ARE NO IMPERVIOUS VALUES IN THE SHAPEFILE
## VALUES IN SHAPEFILE SHOULD BE RATIOS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # TAKE THE CROPPED LU SHAPEFILE TO THE DEM OUTLINE AND
    # ASSIGN IMPERVIOUSNESS TO EACH OF THE CELLS
    # READ THE LAND USE SHAPEFILE
    lu_sp = gpd.read_file(lu_clipped)

    # DEFINE THE COLUMN TO CHECK FOR TYPE OF LAND USE
    column_to_check = "TYPE"  # BASED ON USER INPUT

    # DEFINE A NEW COLUMN
    new_column = "Imperv"

    # MAP VALUES USING THE VALUE_MAPPING DICTIONARY
    lu_sp[new_column] = lu_sp[column_to_check].map(value_mapping).fillna(0)

    # DIVIDE BY 100 TO GET RATIO
    lu_sp[new_column] = lu_sp[new_column] / 100

    # WRITE THE UPDATED GEODATAFRAME TO A NEW SHAPEFILE
    lu_new_field = os.path.join(
        intermediate_storage_files, "Land_Use_Shapefile_Impervious_Values.shp"
    )

    lu_sp.to_file(lu_new_field, driver="ESRI Shapefile")

    print("NEW COLUMN OF RUNOFF COEFFICIENTS ADDED", lu_new_field)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CREATE A NEW FIELD AND CALCULATE POLYGON AREA
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # OPEN THE INPUT SHAPEFILE
    with fiona.open(lu_new_field, "r") as src:
        # GET THE SCHEMA OF THE INPUT SHAPEFILE
        schema = src.schema.copy()

        # ADD THE FIELD TO THE SCHEMA
        schema["properties"]["area"] = "float:19.11"

        # CREATE A NEW SHAPEFILE TO WRITE THE CALCULATED AREA TO
        output_shapefile = os.path.join(
            intermediate_storage_files, "LU_Imp_Polygon_Area.shp"
        )
        with fiona.open(
            output_shapefile, "w", "ESRI Shapefile", schema, src.crs
        ) as dst:
            # LOOP THROUGH THE FEATURES IN THE INPUT SHAPEFILE
            for feature in src:
                # GET THE GEOMETRY OF THE FEATURE
                polygon = shapely.geometry.shape(feature["geometry"])

                # CALCULATE THE AREA OF THE POLYGON
                area = polygon.area

                # UPDATE THE FEATURE TO INCLUDE THE CALCULATED AREA
                feature["properties"]["area"] = area

                # WRITE THE UPDATED FEATURE TO THE OUTPUT SHAPEFILE
                dst.write(feature)
    print("Column Populated with Area")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CREATE A RASTER OF THE LAND USE IMPERVIOUSNESS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # SHAPEFILE THAT IS TO BE WRITTEN TO RASTER
```

```
        # POLYGONSTORASTER
        # READ POLYGON SHAPEFILE
        polygon_toraster = gpd.read_file(output_shapefile)
        # POLYGON_TORASTER = POLYGON_TORASTER.RENAME(COLUMNS={'IMPERVIOUS': 'IMP'})

        # REPROJECTING TO COORDINATE SYSTEM
        polygon_toraster = polygon_toraster.to_crs("EPSG:" + str(dem_espg2int))

        # USING GEOCUBE TO RASTERIZE THE VECTOR

        # LABEL THE NAME OF THE COLUMN FOR THE MEASUREMENTS DEPENDING ON INPUT
        measurement_var = new_column

        polygon_toraster_rasterize = make_geocube(
            vector_data=polygon_toraster,
            measurements=[measurement_var],
            resolution=(-y_res, x_res),
            fill=0,
        )

        # SAVE RASTER CENSUS RASTER
        # RASTER NAME
        lu_raster = os.path.join(intermediate_storage_files, "LU_Raster.tif")
        polygon_toraster_rasterize.rio.to_raster(
            lu_raster
        )  # IF IT IS OPEN SOMEWHERE THE SCRIPT WILL NOT RUN PAST HERE, WILL NEED TO RUN IN A NEW DIRECTORY

        with rasterio.open(lu_raster, "r+") as dst:
            # DEFINE THE CRS USING EPSG CODE
            crs = rasterio.crs.CRS.from_epsg(str(dem_espg2int))

            # WRITE THE CRS TO THE FILE
            dst.crs = crs
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CROP THE LAND USE TIF'S
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # SHAPEFILE FILE
        ras_lu = lu_raster

        # READ SHAPEFILE
        gdf = gpd.read_file(basin_polygon_sub_path)

        gdf["count"] = [i + 1 for i, _ in enumerate(gdf.geometry)]
        outputfile = os.path.join(intermediate_storage_files, "basin_selected_count.shp")
        gdf.to_file(outputfile)

        gdf1 = gpd.read_file(outputfile)

        # GDF.CRS
        names = [x for x in gdf1["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

        # READ SHAPEFILE
        gdf3 = gpd.read_file(outputfile)

        # GDF.CRS
        names = [x for x in gdf3["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

        # READ RASTER DATA
        ras_data = rasterio.open(ras_lu)

        # OUTPUT PATH
        out_raster_lu = os.path.join(
            intermediate_storage_files, "LURasterIntermediateSteps"
        )
        if not os.path.exists(out_raster_lu):
            os.makedirs(out_raster_lu)

        # SAVE OUTPUT FILES AS PER SHAPEFILE FEATURES
        for i in range(len(gdf3)):
            geom = []
            coord = shapely.geometry.mapping(gdf3)["features"][i]["geometry"]
            geom.append(coord)
            with rasterio.open(ras_lu) as src:
                out_image, out_transform = rasterio.mask.mask(src, geom, crop=True)
                out_meta = src.meta
                out_meta.update(
                    {
                        "driver": "GTiff",
                        "height": out_image.shape[1],
                        "width": out_image.shape[2],
                        "transform": out_transform,
                    }
                )
                # FILE_PATH = ''
                with rasterio.open(
                    f"{out_raster_lu}/{names[i]}.tif", "w", **out_meta
                ) as dest:
                    dest.write(out_image)

        luraster_files = glob.glob(out_raster_lu + "/*.tif")
        mergedfilelu = os.path.join(intermediate_storage_files, "Final_LU_Raster.tif")

        # OPEN INPUT RASTERS
        src_files_to_mosaic2 = []
        # PRINT(SRC_FILES_TO_MOSAIC2)

        for file in luraster_files:
            src = rasterio.open(file)
            src_files_to_mosaic2.append(src)

        # MERGE INPUT RASTERS INTO ONE
        mosaic, out_trans = rasterio.merge.merge(src_files_to_mosaic2)

        # CLOSE ALL INPUT RASTER FILES
        for src in src_files_to_mosaic2:
            src.close()

        # UPDATE METADATA FOR OUTPUT RASTER
        out_meta = src.meta.copy()
        out_meta.update(
            {
                "driver": "GTiff",
                "height": mosaic.shape[1],
                "width": mosaic.shape[2],
                "transform": out_trans,
            }
        )

        # WRITE OUTPUT RASTER TO DISK
        with rasterio.open(mergedfilelu, "w", **out_meta) as dest:
            dest.write(mosaic)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## REPROJECT THE FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        reprojectedfiles = os.path.join(intermediate_storage_files, "WhiteBoxToolFiles")
        if not os.path.exists(reprojectedfiles):
            os.makedirs(reprojectedfiles)

        # DEFINE EPSG CODE
        epsg_code = dem_espg2int

        # CREATE SPATIALREFERENCE OBJECT FROM EPSG CODE
        srs = osr.SpatialReference()
        srs.ImportFromEPSG(int(epsg_code))

        # CONVERT SPATIALREFERENCE OBJECT TO WKT STRING
        new_crs_wkt = srs.ExportToWkt()

        # COPY THE DEMS
        # OPEN INPUT RASTER FILE
```

```
input_raster = gdal.Open(dem_clipped_path, gdal.GA_Update)

# GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
existing_geotransform = input_raster.GetGeoTransform()
new_geotransform = (
    existing_geotransform[0],
    existing_geotransform[1],
    0,
    existing_geotransform[3],
    0,
    existing_geotransform[5],
)

# SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
input_raster.SetGeoTransform(new_geotransform)
input_raster.SetProjection(new_crs_wkt)

# CREATE OUTPUT RASTER FILE
driver = gdal.GetDriverByName("GTiff")
outputdemreproj = os.path.join(reprojectedfiles, "Reprojected_DEM.tif")
output_raster = driver.CreateCopy(outputdemreproj, input_raster)

# CLOSE INPUT AND OUTPUT RASTER FILES
input_raster = None
output_raster = None

# COPY THE ABSORPTION
# OPEN INPUT RASTER FILE
input_raster1 = gdal.Open(absorption_file, gdal.GA_Update)

driver = gdal.GetDriverByName("GTiff")
outputabsreproj = os.path.join(reprojectedfiles, "Copy_Absorption.tif")
output_raster1 = driver.CreateCopy(outputabsreproj, input_raster1)

# CLOSE INPUT AND OUTPUT RASTER FILES
input_raster1 = None
output_raster1 = None

# COPY THE EFFICIENCY
# OPEN INPUT RASTER FILE
input_raster2 = gdal.Open(efficiency_file, gdal.GA_Update)

# CREATE OUTPUT RASTER FILE
driver = gdal.GetDriverByName("GTiff")
outputeffreproj = os.path.join(reprojectedfiles, "Copy_Efficiency.tif")
output_raster2 = driver.CreateCopy(outputeffreproj, input_raster2)

# CLOSE INPUT AND OUTPUT RASTER FILES
input_raster2 = None
output_raster2 = None

# COPY THE LAND USE
# OPEN INPUT RASTER FILE
input_raster3 = gdal.Open(mergedfilelu, gdal.GA_Update)

# GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
existing_geotransform3 = input_raster3.GetGeoTransform()
new_geotransform3 = (
    existing_geotransform3[0],
    existing_geotransform3[1],
    0,
    existing_geotransform3[3],
    0,
    existing_geotransform3[5],
)

# SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
input_raster3.SetGeoTransform(new_geotransform3)
input_raster3.SetProjection(new_crs_wkt)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputlureproj = os.path.join(reprojectedfiles, "Reprojected_LandUse.tif")
    output_raster3 = driver.CreateCopy(outputlureproj, input_raster3)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster3 = None
    output_raster3 = None

    # COPY THE FLOWACCUM
    # OPEN INPUT RASTER FILE
    input_raster4 = gdal.Open(flow_accum, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform4 = input_raster4.GetGeoTransform()
    new_geotransform4 = (
        existing_geotransform4[0],
        existing_geotransform4[1],
        0,
        existing_geotransform4[3],
        0,
        existing_geotransform4[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster4.SetGeoTransform(new_geotransform4)
    input_raster4.SetProjection(new_crs_wkt)
    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputflowaccumreproj = os.path.join(reprojectedfiles, "Reprojected_FlowAccum.tif")
    output_raster4 = driver.CreateCopy(outputflowaccumreproj, input_raster4)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster4 = None
    output_raster4 = None
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CHECK SPATIAL EXTENTS OF THE FILES
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # DICTIONARY TO STORE PROCESSED RASTERS
    raster_dict = {}

    # LOOP THROUGH ALL GEOTIFF FILES IN THE FOLDER
    for file_name in os.listdir(reprojectedfiles):
        if file_name.endswith(".tif"):
            file_path = os.path.join(reprojectedfiles, file_name)

            # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
            raster = gdal.Open(file_path, gdal.GA_Update)
            raster_gt = raster.GetGeoTransform()
            raster_proj = raster.GetProjection()
            raster_size = (raster.RasterXSize, raster.RasterYSize)

            # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
            if (
                raster_gt == small_raster.GetGeoTransform()
                and raster_proj == small_raster.GetProjection()
                and raster_size == small_size
            ):
                print(
                    f"{file_name} is already in the correct CRS, resolution, and size."
                )
            else:
                # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
                raster_update_path = os.path.join(
                    reprojectedfiles, f"{file_name.split('.')[0]}_update.tif"
                )
```

```
                new_raster = gdal.Translate(
                    raster_update_path,
                    raster,
                    format="GTiff",
                    width=small_size[0],
                    height=small_size[1],
                )
                new_raster.SetGeoTransform(small_raster.GetGeoTransform())
                new_raster.SetProjection(small_raster.GetProjection())
                print(f"{file_name} has been updated.")

            # CLOSE THE RASTER FILE
            raster = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## LOOP THROUGH THE FOLDER AND CHECK IF THERE ARE UPDATE FILES THEN ASSIGN IT VARIABLES
## IF THERE IS NO CHANGE IN CRS/RES/EXTENTS USE THE OLD TIF
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    wbt_demfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_DEM_update") and filename.endswith(".tif"):
            wbt_demfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_demfile is not None:
        print("Update file found:", wbt_demfile)
    else:
        wbt_demfile = outputdemreproj
        print("Using the old file:", wbt_demfile)

    wbt_absfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Absorption_update") and filename.endswith(
            ".tif"
        ):
            wbt_absfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_absfile is not None:
        print("Update file found:", wbt_absfile)
    else:
        wbt_absfile = outputabsreproj
        print("Using the old file:", wbt_absfile)

    wbt_efffile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Efficiency_update") and filename.endswith(
            ".tif"
        ):
            wbt_efffile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_efffile is not None:
        print("Update file found:", wbt_efffile)
    else:
        wbt_efffile = outputeffreproj
        print("Using the old file:", wbt_efffile)
    wbt_lufile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_LandUse_update") and filename.endswith(
            ".tif"
        ):
            wbt_lufile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_lufile is not None:
        print("Update file found:", wbt_lufile)
    else:
        wbt_lufile = outputlureproj
        print("Using the old file:", wbt_lufile)

    dividing_flowaccum = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_FlowAccum_update") and filename.endswith(
            ".tif"
        ):
            dividing_flowaccum = os.path.join(reprojectedfiles, filename)
            break
    if dividing_flowaccum is not None:
        print("Update file found:", dividing_flowaccum)
    else:
        dividing_flowaccum = outputflowaccumreproj
        print("Using the old file:", dividing_flowaccum)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## USE WBT TO OUTPUT THE D8 MASSFLUX
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    # BRING IN WHITEBOX TOOLS
    wbt = whitebox.WhiteboxTools()

    # DEFINE THE OUTPUT_FILE
    dmassoutput = os.path.join(intermediate_storage_files, "D8MassFluxOutput.tif")

    # USE D8MASSFLUX TOOL AND OUTPUT A FILE
    wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_efffile, wbt_absfile, dmassoutput)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## SAVE IN LOG SCALE
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    with rasterio.open(dmassoutput) as src:
        # READ THE RASTER DATA INTO A NUMPY ARRAY
        data = src.read(1)
        # APPLY LOG TRANSFORMATION
        data_log = np.log(data + 1)

        # CREATE NEW RASTER FILE
        profile = src.profile  # META DATA COPY
        profile.update(nodata=np.nan)

        LogScale_D8 = os.path.join(
            intermediate_storage_files, "LogScale_D8MassFlux.tif"
        )
        with rasterio.open(LogScale_D8, "w", **profile) as dst:
            dst.write(data_log, 1)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## GET THE IMPERVIOUSNESS
## ---------------------------------------------------------------------------------------------------------------------------------------------------
    # TO GET THE IMPERVIOUSNESS = OUTPUTFILE/FLOW ACCUM
    # WE NEED FLOW ACCUM, D8MASSFLUXOUTPUT, AN OUTPUT FOR THE CLIPPED FILE, AN OUTPUT FOR THE
REPROJECTED FILE
    # AND AN OUTPUT FOR THE IMPERVIOUSNESS
    # OUTPUT_FILE = D8MASSFLUX OUTPUT
    output_impervious = os.path.join(intermediate_storage_files, "Final_Imperviousness.tif")

    # DIVIDING!
    # OPEN THE TWO RASTER FILES
    src3 = gdal.Open(dmassoutput)
    src4 = gdal.Open(dividing_flowaccum)

    # GET THE SIZE OF THE RASTERS
    cols = src3.RasterXSize
    rows = src3.RasterYSize

    # CREATE AN OUTPUT RASTER WITH THE SAME SIZE AND PROJECTION AS THE INPUT RASTERS
    driver = gdal.GetDriverByName("GTiff")
    out_raster = driver.Create(output_impervious, cols, rows, 1, gdal.GDT_Float32)
    out_raster.SetProjection(src3.GetProjection())
    out_raster.SetGeoTransform(src3.GetGeoTransform())

    # READ THE DATA FROM BOTH RASTERS INTO NUMPY ARRAYS
    data1 = src3.ReadAsArray()
    data2 = src4.ReadAsArray()

    # PERFORM ELEMENT-WISE DIVISION OF THE TWO ARRAYS
    result = np.divide(data1, data2)

    # SET THE RANGE OF VALUES TO WRITE TO THE OUTPUT RASTER
```

```
data_min = 0
data_max = 1

# CLIP THE RESULT ARRAY TO THE SPECIFIED RANGE
result_clipped = np.clip(result, data_min, data_max)

# WRITE THE RESULT TO THE OUTPUT RASTER
out_raster.GetRasterBand(1).WriteArray(result_clipped)

# CLOSE THE RASTERS
src3 = None
src4 = None
out_raster = None

print("Divided Rasters, Imperviousness Found")

with rasterio.open(output_impervious) as src:
    imp = src.read(1)

# PLOT THE RASTER
plt.figure(figsize=(10, 10))
rasterio.plot.show(imp, transform=src.transform, cmap="terrain")
plt.show()

# OPEN DEM RASTER
small_raster = gdal.Open(outputdemreproj)
small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

# OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
clippedimperviousraster = gdal.Open(output_impervious, gdal.GA_Update)
rasterimp_gt = clippedimperviousraster.GetGeoTransform()
rasterimp_proj = clippedimperviousraster.GetProjection()
rasterimp_size = (
    clippedimperviousraster.RasterXSize,
    clippedimperviousraster.RasterYSize,
)

# CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
if (
    rasterimp_gt == small_raster.GetGeoTransform()
    and rasterimp_proj == small_raster.GetProjection()
    and rasterimp_size == small_size
):
    print("Imperviousness is already in the correct CRS, resolution, and size.")
else:
    # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
    raster_imp_update = os.path.join(
        intermediate_storage_files, "Final_Imperviousness_update.tif"
    )
    new_raster1 = gdal.Translate(
        raster_imp_update,
        clippedimperviousraster,
        format="GTiff",
        width=small_size[0],
        height=small_size[1],
    )
    new_raster1.SetGeoTransform(small_raster.GetGeoTransform())
    new_raster1.SetProjection(small_raster.GetProjection())
    print("Imperviousness has been updated.")

# CLOSE THE RASTER FILES
raster = None

divided_imp = None
for filename in os.listdir(intermediate_storage_files):
    if filename.startswith("Final_Imperviousness_update") and filename.endswith(
        ".tif"
    ):
        divided_imp = os.path.join(intermediate_storage_files, filename)
        break
    if divided_imp is not None:
        print("Update file found:", divided_imp)
    else:
        divided_imp = output_impervious
        print("Using the old file:", divided_imp)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## ATTACH IMPERVIOUSNESS TO THE STREAM SEGMENTS BASED ON THE END XY
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # FIRST EXTRACT THE VALUE OF THE RASTER AT THE STREAM POINT LOCATION
    # STREAM POINT FILE = STREAM_POINT_FIELD_RENAME_PATH
    # RASTER = OUTPUT_IMP

    extract_pts = os.path.join(intermediate_storage_files, "Urban_Stream_Points_Imperviousness.shp")

    # READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
    gdf = gpd.read_file(stream_point_field_rename_path)

    gdf["D8Flux"] = np.nan
    gdf["LUFloAccum"] = np.nan

    # OPEN THE RASTER FILE FOR READING
    with rasterio.open(divided_imp) as src, rasterio.open(
        dmassoutput
    ) as src1, rasterio.open(flow_accum) as src2:
        # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
        for index, row in gdf.iterrows():
            # GET THE X AND Y COORDINATES OF THE POINT
            point = row.geometry
            x, y = point.x, point.y

            # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
            impvalue = list(src.sample([(x, y)]))
            massfluxvalue = list(src1.sample([(x, y)]))
            flowaccumvalue = list(src2.sample([(x, y)]))

            # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
            gdf.loc[index, "Imp"] = impvalue[0][0]
            gdf.loc[index, "D8Flux"] = massfluxvalue[0][0]
            gdf.loc[index, "LUFloAccum"] = flowaccumvalue[0][0]

    # WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
    gdf.to_file(extract_pts)

    print("Impervious Values Extracted to Match Stream Points")

    # NOW MERGE THE VALUES BASED ON THE COLUMN
    points_gdf = gpd.read_file(extract_pts)
    lines_gdf = gpd.read_file(copy_strm_net)

    intstep = os.path.join(intermediate_storage_files, "Copy_Rural_Stream_Power.shp")
    urban_segments = os.path.join(
        intermediate_storage_files, "Urban_Imperviousness_Attachment.shp"
    )

    lines_gdf.to_file(intstep)

    lines_gdf2 = gpd.read_file(intstep)

    # POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
    # THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
    common_column = "pXY"

    # MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
    merged_gdf = lines_gdf2.set_index(common_column).join(
        points_gdf.set_index(common_column)[["Imp", "D8Flux", "LUFloAccum"]]
    )
    print(merged_gdf.columns)

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf.to_file(urban_segments)
```

```
        gdf = gpd.read_file(urban_segments)
        print(
            "Impervious, D8 Mass Flux, Flow Accumulation Values Appended to Stream Segments"
        )

    elif lurasteranalysis == "Y":
        print("Raster of LU Imperviousness is provided")
        # ENSURE INTERMEDIATE STORAGE FOLDER EXISTS
        if not os.path.exists(intermediate_storage_files):
            os.makedirs(intermediate_storage_files)

        # DEFINE THE NEW LOCATION TO SAVE THE RASTER
        impraster = os.path.join(intermediate_storage_files, "LU_Imperviousness_raster.tif")

        # OPEN THE ORIGINAL RASTER FILE
        with rasterio.open(lurasteranalysispath) as src:
            raster_data = src.read(1)  # READ THE FIRST BAND OF THE RASTER
            profile = src.profile  # GET THE METADATA FOR THE RASTER
            # NORMALIZE IMPERVIOUSNESS VALUES (IF VALUES ARE IN PERCENTAGE FORM)
            raster_data = raster_data / 100.0
            raster_data = np.clip(raster_data, 0, 1)  # CLAMP VALUES BETWEEN 0 AND 1

        # SAVE THE RASTER TO THE OUTPUT PATH WITHOUT MODIFICATION
        with rasterio.open(impraster, 'w', **profile) as dst:
            dst.write(raster_data, 1)  # WRITE THE ORIGINAL RASTER DATA TO THE FIRST BAND

        print(f"Raster saved successfully to {impraster}")

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## PREPROCESS THE INPUT RASTERS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # RESOLUTION EXTRACTED AT TOP
        with rasterio.open(dem_clipped_path) as src:
            # GET THE AFFINE TRANSFORM FROM THE METADATA
            transform = src.transform

            # THE RESOLUTION IS STORED IN THE CELLSIZE OF THE AFFINE TRANSFORM
            x_resolution = transform.a
            y_resolution = transform.e

            # THE CELLSIZE IS THE ABSOLUTE VALUE OF THE RESOLUTION, AS IT MAY BE NEGATIVE IF THE RASTER IS
ROTATED
            x_cellsize = abs(x_resolution)
            y_cellsize = abs(y_resolution)

        crs = rasterio.crs.CRS.from_epsg(int(dem_espg2int))

        # ABSORPTION RASTER
        # PREPROCESS THE ABSORPTION RASTER, WE WANT TO HAVE ZERO ABSORPTION
        # OPEN THE RASTER FILE
        with rasterio.open(dem_clipped_path) as src:
            # READ THE DATA INTO A NUMPY ARRAY
            data = src.read(1)

            # SET ALL CELLS TO THE DESIRED VALUE
            data[:] = 0

            # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
            meta = src.meta.copy()
            meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

            absorption_file = os.path.join(intermediate_storage_files, "Absorption.tif")
            # WRITE THE UPDATED ARRAY TO A NEW FILE
            with rasterio.open(absorption_file, "w", **meta) as dst:
                dst.write(data, 1)

        # EFFICIENCY RASTER
        # PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY
        with rasterio.open(dem_clipped_path) as src:
            # READ THE DATA INTO A NUMPY ARRAY
            data = src.read(1)

            # SET ALL CELLS TO THE DESIRED VALUE
            data[:] = 1

            # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
            meta = src.meta.copy()
            meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

            efficiency_file = os.path.join(intermediate_storage_files, "Efficiency.tif")
            # WRITE THE UPDATED ARRAY TO A NEW FILE
            with rasterio.open(efficiency_file, "w", **meta) as dst:
                dst.write(data, 1)

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CROP THE LAND USE TIF'S
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # SHAPEFILE FILE
        ras_lu = impraster

        # READ SHAPEFILE
        gdf = gpd.read_file(basin_polygon_sub_path)

        gdf["count"] = [i + 1 for i, _ in enumerate(gdf.geometry)]
        outputfile = os.path.join(intermediate_storage_files, "basin_selected_count.shp")
        gdf.to_file(outputfile)

        gdf1 = gpd.read_file(outputfile)

        # GDF.CRS
        names = [x for x in gdf1["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

        # READ SHAPEFILE
        gdf3 = gpd.read_file(outputfile)

        # GDF.CRS
        names = [x for x in gdf3["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

        # READ RASTER DATA
        ras_data = rasterio.open(ras_lu)

        # OUTPUT PATH
        out_raster_lu = os.path.join(
            intermediate_storage_files, "LURasterIntermediateSteps"
        )
        if not os.path.exists(out_raster_lu):
            os.makedirs(out_raster_lu)

        # SAVE OUTPUT FILES AS PER SHAPEFILE FEATURES
        for i in range(len(gdf3)):
            geom = []
            coord = shapely.geometry.mapping(gdf3)["features"][i]["geometry"]
            geom.append(coord)
            with rasterio.open(ras_lu) as src:
                out_image, out_transform = rasterio.mask.mask(src, geom, crop=True)
                out_meta = src.meta
                out_meta.update(
                    {
                        "driver": "GTiff",
                        "height": out_image.shape[1],
                        "width": out_image.shape[2],
                        "transform": out_transform,
                    }
                )
                # FILE_PATH = ''
                with rasterio.open(
                    f"{out_raster_lu}/{names[i]}.tif", "w", **out_meta
                ) as dest:
                    dest.write(out_image)
```

```
    luraster_files = glob.glob(out_raster_lu + "/*.tif")
    mergedfilelu = os.path.join(intermediate_storage_files, "Final_LU_Raster.tif")

    # OPEN INPUT RASTERS
    src_files_to_mosaic2 = []
    # PRINT(SRC_FILES_TO_MOSAIC2)

    for file in luraster_files:
        src = rasterio.open(file)
        src_files_to_mosaic2.append(src)

    # MERGE INPUT RASTERS INTO ONE
    mosaic, out_trans = rasterio.merge.merge(src_files_to_mosaic2)

    # CLOSE ALL INPUT RASTER FILES
    for src in src_files_to_mosaic2:
        src.close()

    # UPDATE METADATA FOR OUTPUT RASTER
    out_meta = src.meta.copy()
    out_meta.update(
        {
            "driver": "GTiff",
            "height": mosaic.shape[1],
            "width": mosaic.shape[2],
            "transform": out_trans,
        }
    )

    # WRITE OUTPUT RASTER TO DISK
    with rasterio.open(mergedfilelu, "w", **out_meta) as dest:
        dest.write(mosaic)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## REPROJECT THE FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    reprojectedfiles = os.path.join(intermediate_storage_files, "WhiteBoxToolFiles")
    if not os.path.exists(reprojectedfiles):
        os.makedirs(reprojectedfiles)

    # DEFINE EPSG CODE
    epsg_code = dem_espg2int

    # CREATE SPATIALREFERENCE OBJECT FROM EPSG CODE
    srs = osr.SpatialReference()
    srs.ImportFromEPSG(int(epsg_code))

    # CONVERT SPATIALREFERENCE OBJECT TO WKT STRING
    new_crs_wkt = srs.ExportToWkt()

    # COPY THE DEMS
    # OPEN INPUT RASTER FILE
    input_raster = gdal.Open(dem_clipped_path, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform = input_raster.GetGeoTransform()
    new_geotransform = (
        existing_geotransform[0],
        existing_geotransform[1],
        0,
        existing_geotransform[3],
        0,
        existing_geotransform[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster.SetGeoTransform(new_geotransform)
    input_raster.SetProjection(new_crs_wkt)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputdemreproj = os.path.join(reprojectedfiles, "Reprojected_DEM.tif")
    output_raster = driver.CreateCopy(outputdemreproj, input_raster)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster = None
    output_raster = None

    # COPY THE ABSORPTION
    # OPEN INPUT RASTER FILE
    input_raster1 = gdal.Open(absorption_file, gdal.GA_Update)

    driver = gdal.GetDriverByName("GTiff")
    outputabsreproj = os.path.join(reprojectedfiles, "Copy_Absorption.tif")
    output_raster1 = driver.CreateCopy(outputabsreproj, input_raster1)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster1 = None
    output_raster1 = None

    # COPY THE EFFICIENCY
    # OPEN INPUT RASTER FILE
    input_raster2 = gdal.Open(efficiency_file, gdal.GA_Update)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputeffreproj = os.path.join(reprojectedfiles, "Copy_Efficiency.tif")
    output_raster2 = driver.CreateCopy(outputeffreproj, input_raster2)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster2 = None
    output_raster2 = None

    # COPY THE LAND USE
    # OPEN INPUT RASTER FILE
    input_raster3 = gdal.Open(mergedfilelu, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform3 = input_raster3.GetGeoTransform()
    new_geotransform3 = (
        existing_geotransform3[0],
        existing_geotransform3[1],
        0,
        existing_geotransform3[3],
        0,
        existing_geotransform3[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster3.SetGeoTransform(new_geotransform3)
    input_raster3.SetProjection(new_crs_wkt)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputlureproj = os.path.join(reprojectedfiles, "Reprojected_LandUse.tif")
    output_raster3 = driver.CreateCopy(outputlureproj, input_raster3)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster3 = None
    output_raster3 = None

    # COPY THE FLOWACCUM
    # OPEN INPUT RASTER FILE
    input_raster4 = gdal.Open(flow_accum, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL SIZE
    existing_geotransform4 = input_raster4.GetGeoTransform()
    new_geotransform4 = (
        existing_geotransform4[0],
        existing_geotransform4[1],
```

```
        0,
        existing_geotransform4[3],
        0,
        existing_geotransform4[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster4.SetGeoTransform(new_geotransform4)
    input_raster4.SetProjection(new_crs_wkt)
    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputflowaccumreproj = os.path.join(reprojectedfiles, "Reprojected_FlowAccum.tif")
    output_raster4 = driver.CreateCopy(outputflowaccumreproj, input_raster4)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster4 = None
    output_raster4 = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CHECK SPATIAL EXTENTS OF THE FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # DICTIONARY TO STORE PROCESSED RASTERS
    raster_dict = {}

    # LOOP THROUGH ALL GEOTIFF FILES IN THE FOLDER
    for file_name in os.listdir(reprojectedfiles):
        if file_name.endswith(".tif"):
            file_path = os.path.join(reprojectedfiles, file_name)

            # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
            raster = gdal.Open(file_path, gdal.GA_Update)
            raster_gt = raster.GetGeoTransform()
            raster_proj = raster.GetProjection()
            raster_size = (raster.RasterXSize, raster.RasterYSize)

            # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
            if (
                raster_gt == small_raster.GetGeoTransform()
                and raster_proj == small_raster.GetProjection()
                and raster_size == small_size
            ):
                print(
                    f"{file_name} is already in the correct CRS, resolution, and size."
                )
            else:
                # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
                raster_update_path = os.path.join(
                    reprojectedfiles, f"{file_name.split('.')[0]}_update.tif"
                )
                new_raster = gdal.Translate(
                    raster_update_path,
                    raster,
                    format="GTiff",
                    width=small_size[0],
                    height=small_size[1],
                )
                new_raster.SetGeoTransform(small_raster.GetGeoTransform())
                new_raster.SetProjection(small_raster.GetProjection())
                print(f"{file_name} has been updated.")

            # CLOSE THE RASTER FILE
            raster = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## LOOP THROUGH THE FOLDER AND CHECK IF THERE ARE UPDATE FILES THEN ASSIGN IT VARIABLES
## IF THERE IS NO CHANGE IN CRS/RES/EXTENTS USE THE OLD TIF
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    wbt_demfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_DEM_update") and filename.endswith(".tif"):
            wbt_demfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_demfile is not None:
        print("Update file found:", wbt_demfile)
    else:
        wbt_demfile = outputdemreproj
        print("Using the old file:", wbt_demfile)

    wbt_absfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Absorption_update") and filename.endswith(
            ".tif"
        ):
            wbt_absfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_absfile is not None:
        print("Update file found:", wbt_absfile)
    else:
        wbt_absfile = outputabsreproj
        print("Using the old file:", wbt_absfile)

    wbt_efffile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Efficiency_update") and filename.endswith(
            ".tif"
        ):
            wbt_efffile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_efffile is not None:
        print("Update file found:", wbt_efffile)
    else:
        wbt_efffile = outputeffreproj
        print("Using the old file:", wbt_efffile)
    wbt_lufile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_LandUse_update") and filename.endswith(
            ".tif"
        ):
            wbt_lufile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_lufile is not None:
        print("Update file found:", wbt_lufile)
    else:
        wbt_lufile = outputlureproj
        print("Using the old file:", wbt_lufile)

    dividing_flowaccum = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_FlowAccum_update") and filename.endswith(
            ".tif"
        ):
            dividing_flowaccum = os.path.join(reprojectedfiles, filename)
            break
    if dividing_flowaccum is not None:
        print("Update file found:", dividing_flowaccum)
    else:
        dividing_flowaccum = outputflowaccumreproj
        print("Using the old file:", dividing_flowaccum)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## USE WBT TO OUTPUT THE D8 MASSFLUX
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # BRING IN WHITEBOX TOOLS
    wbt = whitebox.WhiteboxTools()

    # DEFINE THE OUTPUT_FILE
    dmassoutput = os.path.join(intermediate_storage_files, "D8MassFluxOutput.tif")

    # USE D8MASSFLUX TOOL AND OUTPUT A FILE
```

```
    wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_efffile, wbt_absfile, dmassoutput)
    #WBT.D8_MASS_FLUX(DEM_CLIPPED_PATH, MERGEDFILELU, EFFICIENCY_FILE, ABSORPTION_FILE, DMASSOUTPUT)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## SAVE IN LOG SCALE
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    with rasterio.open(dmassoutput) as src:
        # READ THE RASTER DATA INTO A NUMPY ARRAY
        data = src.read(1)
        # APPLY LOG TRANSFORMATION
        data_log = np.log(data + 1)

        # CREATE NEW RASTER FILE
        profile = src.profile  # META DATA COPY
        profile.update(nodata=np.nan)

        LogScale_D8 = os.path.join(
            intermediate_storage_files, "LogScale_D8MassFlux.tif"
        )
        with rasterio.open(LogScale_D8, "w", **profile) as dst:
            dst.write(data_log, 1)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## GET THE IMPERVIOUSNESS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # TO GET THE IMPERVIOUSNESS = OUTPUTFILE/FLOW ACCUM
    # WE NEED FLOW ACCUM, D8MASSFLUXOUTPUT, AN OUTPUT FOR THE CLIPPED FILE, AN OUTPUT FOR THE
REPROJECTED FILE
    # AND AN OUTPUT FOR THE IMPERVIOUSNESS
    # OUTPUT_FILE = D8MASSFLUX OUTPUT
    output_impervious = os.path.join(intermediate_storage_files, "Final_Imperviousness.tif")

    # DIVIDING!
    # OPEN THE TWO RASTER FILES
    src3 = gdal.Open(dmassoutput)
    src4 = gdal.Open(dividing_flowaccum)

    # GET THE SIZE OF THE RASTERS
    cols = src3.RasterXSize
    rows = src3.RasterYSize

    # CREATE AN OUTPUT RASTER WITH THE SAME SIZE AND PROJECTION AS THE INPUT RASTERS
    driver = gdal.GetDriverByName("GTiff")
    out_raster = driver.Create(output_impervious, cols, rows, 1, gdal.GDT_Float32)
    out_raster.SetProjection(src3.GetProjection())
    out_raster.SetGeoTransform(src3.GetGeoTransform())

    # READ THE DATA FROM BOTH RASTERS INTO NUMPY ARRAYS
    data1 = src3.ReadAsArray()
    data2 = src4.ReadAsArray()

    # PERFORM ELEMENT-WISE DIVISION OF THE TWO ARRAYS
    result = np.divide(data1, data2)

    # SET THE RANGE OF VALUES TO WRITE TO THE OUTPUT RASTER
    data_min = 0
    data_max = 1

    # CLIP THE RESULT ARRAY TO THE SPECIFIED RANGE
    result_clipped = np.clip(result, data_min, data_max)

    # WRITE THE RESULT TO THE OUTPUT RASTER
    out_raster.GetRasterBand(1).WriteArray(result_clipped)

    # CLOSE THE RASTERS
    src3 = None
    src4 = None
    out_raster = None

    print("Divided Rasters, Imperviousness Found")

    with rasterio.open(output_impervious) as src:
        imp = src.read(1)

    # PLOT THE RASTER
    plt.figure(figsize=(10, 10))
    rasterio.plot.show(imp, transform=src.transform, cmap="terrain")
    plt.show()

    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
    clippedimperviousraster = gdal.Open(output_impervious, gdal.GA_Update)
    rasterimp_gt = clippedimperviousraster.GetGeoTransform()
    rasterimp_proj = clippedimperviousraster.GetProjection()
    rasterimp_size = (
        clippedimperviousraster.RasterXSize,
        clippedimperviousraster.RasterYSize,
    )

    # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
    if (
        rasterimp_gt == small_raster.GetGeoTransform()
        and rasterimp_proj == small_raster.GetProjection()
        and rasterimp_size == small_size
    ):
        print("Imperviousness is already in the correct CRS, resolution, and size.")
    else:
        # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
        raster_imp_update = os.path.join(
            intermediate_storage_files, "Final_Imperviousness_update.tif"
        )
        new_raster1 = gdal.Translate(
            raster_imp_update,
            clippedimperviousraster,
            format="GTiff",
            width=small_size[0],
            height=small_size[1],
        )
        new_raster1.SetGeoTransform(small_raster.GetGeoTransform())
        new_raster1.SetProjection(small_raster.GetProjection())
        print("Imperviousness has been updated.")

    # CLOSE THE RASTER FILES
    raster = None

    divided_imp = None
    for filename in os.listdir(intermediate_storage_files):
        if filename.startswith("Final_Imperviousness_update") and filename.endswith(
            ".tif"
        ):
            divided_imp = os.path.join(intermediate_storage_files, filename)
            break
    if divided_imp is not None:
        print("Update file found:", divided_imp)
    else:
        divided_imp = output_impervious
        print("Using the old file:", divided_imp)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## ATTACH IMPERVIOUSNESS TO THE STREAM SEGMENTS BASED ON THE END XY
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # FIRST EXTRACT THE VALUE OF THE RASTER AT THE STREAM POINT LOCATION
    # STREAM POINT FILE = STREAM_POINT_FIELD_RENAME_PATH
    # RASTER = OUTPUT_IMP

    extract_pts = os.path.join(intermediate_storage_files, "Urban_Stream_Points_Imperviousness.shp")

    # READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
    gdf = gpd.read_file(stream_point_field_rename_path)
```

```
    gdf["D8Flux"] = np.nan
    gdf["LUFloAccum"] = np.nan

    # OPEN THE RASTER FILE FOR READING
    with rasterio.open(divided_imp) as src, rasterio.open(
        dmassoutput
    ) as src1, rasterio.open(flow_accum) as src2:
        # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
        for index, row in gdf.iterrows():
            # GET THE X AND Y COORDINATES OF THE POINT
            point = row.geometry
            x, y = point.x, point.y

            # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
            impvalue = list(src.sample([(x, y)]))
            massfluxvalue = list(src1.sample([(x, y)]))
            flowaccumvalue = list(src2.sample([(x, y)]))

            # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
            gdf.loc[index, "Imp"] = impvalue[0][0]
            gdf.loc[index, "D8Flux"] = massfluxvalue[0][0]
            gdf.loc[index, "LUFloAccum"] = flowaccumvalue[0][0]

    # WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
    gdf.to_file(extract_pts)

    print("Impervious Values Extracted to Match Stream Points")

    # NOW MERGE THE VALUES BASED ON THE COLUMN
    points_gdf = gpd.read_file(extract_pts)
    lines_gdf = gpd.read_file(copy_strm_net)

    intstep = os.path.join(intermediate_storage_files, "Copy_Rural_Stream_Power.shp")
    urban_segments = os.path.join(
        intermediate_storage_files, "Urban_Imperviousness_Attachment.shp"
    )

    lines_gdf.to_file(intstep)

    lines_gdf2 = gpd.read_file(intstep)

    # POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
    # THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
    common_column = "pXY"

    # MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
    merged_gdf = lines_gdf2.set_index(common_column).join(
        points_gdf.set_index(common_column)[["Imp", "D8Flux", "LUFloAccum"]]
    )
    print(merged_gdf.columns)

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf.to_file(urban_segments)

    gdf = gpd.read_file(urban_segments)
    print(
        "Impervious, D8 Mass Flux, Flow Accumulation Values Appended to Stream Segments"
    )
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## ATTACH IMPERVIOUSNESS TO THE SEGMENTS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # FIRST EXTRACT THE VALUE OF THE RASTER AT THE STREAM POINT LOCATION
    # STREAM POINT FILE = STREAM_POINT_FIELD_RENAME_PATH
    # RASTER = OUTPUT_IMP

    extract_pts = os.path.join(
        intermediate_storage_files, "Urban_Stream_Points_Imperviousness_raster.shp"
    )

    # READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
    gdf = gpd.read_file(stream_point_field_rename_path)

    gdf["D8Flux"] = np.nan
    gdf["LUFloAccum"] = np.nan

    # OPEN THE RASTER FILE FOR READING
    with rasterio.open(divided_imp) as src, rasterio.open(
        dmassoutput
    ) as src1, rasterio.open(flow_accum) as src2:
        # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
        for index, row in gdf.iterrows():
            # GET THE X AND Y COORDINATES OF THE POINT
            point = row.geometry
            x, y = point.x, point.y

            # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
            impvalue = list(src.sample([(x, y)]))
            massfluxvalue = list(src1.sample([(x, y)]))
            flowaccumvalue = list(src2.sample([(x, y)]))

            # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
            gdf.loc[index, "Imp"] = impvalue[0][0]
            gdf.loc[index, "D8Flux"] = massfluxvalue[0][0]
            gdf.loc[index, "LUFloAccum"] = flowaccumvalue[0][0]

    # WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
    gdf.to_file(extract_pts)

    print("Impervious Values Extracted to Match Stream Points")

    # NOW MERGE THE VALUES BASED ON THE COLUMN
    points_gdf = gpd.read_file(extract_pts)
    lines_gdf = gpd.read_file(copy_strm_net)

    intstep = os.path.join(intermediate_storage_files, "Copy_Rural_Stream_Power.shp")
    urban_segments = os.path.join(
        intermediate_storage_files, "Urban_Imperviousness_Attachment_raster.shp"
    )

    lines_gdf.to_file(intstep)

    lines_gdf2 = gpd.read_file(intstep)

    # POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
    # THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
    common_column = "pXY"

    # MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
    merged_gdf = lines_gdf2.set_index(common_column).join(
        points_gdf.set_index(common_column)[["Imp"]]
    )
    print(merged_gdf.columns)

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf.to_file(urban_segments)

    gdf = gpd.read_file(urban_segments)
    print(
        "Impervious Values Appended to Stream Segments"
    )
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE THE DISCHARGE 2 YR
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    calculation = gpd.read_file(urban_segments)

    # CREATES NEW COLUMN AND SETS DEFAULT VALUE TO 0
    calculation["OGLUQ2m3_s"] = 0
    calculation["OGLUSP2"] = 0
    calculation["Wg_Nperm3"] = 9810
    calculation["OGLUSSP2"] = 0
```

```
    calculation["OGLUQ2m3_s"] = pd.to_numeric(calculation["OGLUQ2m3_s"], errors="coerce")
    calculation["SlopeAVGLU"] = pd.to_numeric(calculation["SlopeAVGLU"], errors="coerce")
    calculation["Wg_Nperm3"] = pd.to_numeric(calculation["Wg_Nperm3"], errors="coerce")
    calculation["RurWidth"] = pd.to_numeric(calculation["RurWidth"], errors="coerce")
    calculation["RuralSSP"] = pd.to_numeric(calculation["RuralSSP"], errors="coerce")

    # ORIGINAL DISCHARGE EQUATION
    # FORMULA FOR DISCHARGE IS Q=CA^X(IA^B) WHERE C, X AND B ARE COEFFICCIENTS, A IS THE DRAINAGE AREA AND
IA IS TOTAL IMPERVIOUSNESS PERCENTAGE
    # A IS IN THE COLUMN TITLED DA_KM2
    # IA IS IN COLUMN TITLED IMP

    c = rQ_a
    x = rQ_b
    b = 0.30

    calculation["OGLUQ2m3_s"] = calculation.apply(
        lambda row: c * (row["DA_km2"] ** x) * ((row["Imp"] * 100) ** b), axis=1
    )

    ## CALCULATE TOTAL POWER
    calculation["OGLUSP2"] = calculation.apply(
        lambda row: (row["OGLUQ2m3_s"]) * (row["SlopeAVGLU"]) * (row["Wg_Nperm3"]), axis=1
    )

    # ORIGINAL METHOD
    ## CALCULATE SPECIFIC STREAM POWER
    calculation["OGLUSSP2"] = calculation.apply(
        lambda row: row["OGLUSP2"] / row["RurWidth"], axis=1 #USING PHILLIPS-DESLOGES WIDTH
    )

    # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON)
    pos_val = calculation["RuralSSP"] > 0
    if pos_val.any():
        calculation["OGLUSSPR2"] = calculation.apply(
            lambda row: row["OGLUSSP2"] / row["RuralSSP"] if row["RuralSSP"] != 0 else 0,
            axis=1,
        )
    else:
        calculation["OGLUSSPR2"] = 0
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    ## REVISED NOSWM DISCHARGE EQUATION
    a = rQ_a
    b = rQ_b
    c = uQ_c
    d = uQ_d

    calculation["E5LUQ2m3_s"] = 0
    calculation["E5LUSP2"] = 0

    calculation["E5LUQ2m3_s"] = pd.to_numeric(calculation["E5LUQ2m3_s"], errors="coerce")
    calculation["E5LUSP2"] = pd.to_numeric(calculation["E5LUSP2"], errors="coerce")

    calculation["E5LUQ2m3_s"] = calculation.apply(
        lambda row: a * (((row["DA_km2"])*(1-row["Imp"])) ** b) +  c * (((row["DA_km2"])*(row["Imp"])) ** d), axis=1
    )

    # TOTAL POWER
    calculation["E5LUSP2"] = calculation.apply(
        lambda row: row["E5LUQ2m3_s"] * row["SlopeAVGLU"] * row["Wg_Nperm3"], axis=1
    )

    # SPECIFIC STREAM POWER
    ### NEWWIDTHURBAN-CALIBRATION
    # ENSURE THAT THE DIFFERENCE IS NON-NEGATIVE
    positive_diff = np.maximum(0, (calculation["E5LUQ2m3_s"] - calculation["Q_m3pers"])/calculation["Q_m3pers"])

    # CALCULATE E5URBWIDTH USING THE NON-NEGATIVE DIFFERENCE
    calculation["E5UrbWidth"] = calculation["RurWidth"]*(1 + urb_width*(positive_diff**0.50))

    # CONVERT COLUMNS TO NUMERIC TYPE
    calculation["E5UrbWidth"] = pd.to_numeric(calculation["E5UrbWidth"], errors="coerce")

    # DISPLAY THE UPDATED DATAFRAME
    print(calculation)

    pos_val4 = calculation["E5UrbWidth"] > 0

    if pos_val4.any():
        calculation["E5ULUSSP2"] = calculation.apply(
            lambda row: row["E5LUSP2"] / row["E5UrbWidth"] if row["E5UrbWidth"] != 0 else 0,
            axis=1,
        )
    else:
        calculation["E5ULUSSP2"] = 0

    # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON)
    pos_val5 = calculation["RuralSSP"] > 0
    if pos_val5.any():
        calculation["E5ULUSSPR2"] = calculation.apply(
            lambda row: row["E5ULUSSP2"] / row["RuralSSP"]
            if row["RuralSSP"] != 0
            else 0,
            axis=1,
        )
    else:
        calculation["E5ULUSSPR2"] = 0

    output_file = os.path.join(
        watershed_output, "Final_Land_Use_Stream_Power_Output.shp"
    )

    calculation.to_file(output_file, driver="ESRI Shapefile")
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## ENGINEERED SWM OPTION
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    if engswmanalysis == "Y":
        print("Running Analysis for Including Engineered SWM")
        engineeredswm_folder = os.path.join(intermediate_storage_files,"Engineered_SWM")
        # ENSURE THE OUTPUT FOLDER EXISTS, OR CREATE IT IF IT DOESN'T
        if not os.path.exists(engineeredswm_folder):
            os.makedirs(engineeredswm_folder)

        # PATH FOR THE COPY OF DEM
        swm_dem_path = os.path.join(engineeredswm_folder, "DEM_Copy_Lakes.tif")
        wbt.flatten_lakes(dem_clipped_path,engineeredsswm,swm_dem_path)
        print("DEM adjusted for ponds successfully.")

        # GET REVISED FLOW DIRECTION (D8 METHOD)
        swm_flow_dir = os.path.join(engineeredswm_folder, "flow_dir_swm.tif")
        wbt.d8_pointer(swm_dem_path, swm_flow_dir)
        print("Flow direction adjustment completed.")

        # GET FLOW ACCUMULATION
        swm_flow_accum = os.path.join(engineeredswm_folder, "flow_accum_swm.tif")
        wbt.d8_flow_accumulation(swm_dem_path, swm_flow_accum, out_type="cells")
        print("Flow accumulation calculation completed.")
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## EFFICIENCY RASTER - ENGINEERED SWM PONDS OPTION 1
## PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY WHERE THERE ARE NO PONDS, AND 0 FOR
WHERE THERE ARE PONDS
## THE USER NEEDS TO PUT A VALUE IN FOR IMPERVIOUSNESS THAT THEY WANT
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # GETTING THE OUTLET FOR EACH OF THE PONDS AND WRITING AN EFFICIENCY RASTER
        # OPEN THE DEM RASTER TO GET ITS DIMENSIONS AND TRANSFORM
        with rasterio.open(swm_dem_path) as src_dem:
            height = src_dem.height
            width = src_dem.width
            transform = src_dem.transform
```

```
# CREATE A NEW EFFICIENCY RASTER FILE
engefficiency_raster_file = os.path.join(engineeredswm_folder, "EngineeringEfficiencyRaster.tif")

# OPEN THE FLOW ACCUMULATION RASTER TO GET ITS DIMENSIONS AND TRANSFORMATION
with rasterio.open(flow_accum) as src:
    width = src.width
    height = src.height
    transform = src.transform
    crs = src.crs

# OPEN THE FLOW DIRECTION RASTER TO GET ITS DIMENSIONS AND TRANSFORMATION
with rasterio.open(swm_flow_dir) as src:  # ASSUMING `FLOW_DIR` IS THE PATH TO THE FLOW DIRECTION RASTER FILE
    flow_dir_data = src.read(1)  # READ THE RASTER DATA
    transform = src.transform
    crs = src.crs
# CREATE AN EMPTY ARRAY TO HOLD THE RASTERIZED POLYGONS
efficiency_data = np.ones((height, width), dtype=np.float32)
# CREATE AN EMPTY LIST TO STORE THE MAXIMUM FLOW ACCUMULATION WITHIN EACH POLYGON
max_flow_accumulations = []

# OPEN THE SHAPEFILE TO GET ITS CRS
with fiona.open(engineeredsswm, "r") as shapefile:
    crs = shapefile.crs

    for feature in shapefile:
        geometry = feature["geometry"]
        rasterized_polygon = rasterize(
            [geometry],
            out_shape=(height, width),
            transform=transform,
            fill=0,  # SET TO 0 TO INDICATE THAT THE PIXEL FALLS WITHIN A POLYGON (POND AREA)
            dtype=np.uint8
        )
        efficiency_data[rasterized_polygon == 0] = 0  # SET THE CORRESPONDING PIXELS TO 0 (POND AREA)

# SET ALL REMAINING PIXELS (NON-POND AREAS) TO 1
efficiency_data[efficiency_data != 0] = 1
# SAVE THE EFFICIENCY RASTER TO A GEOTIFF FILE
with rasterio.open(
    engefficiency_raster_file,
    "w",
    driver="GTiff",
    width=width,
    height=height,
    count=1,
    dtype=np.float32,
    crs=crs,
    transform=transform,
) as dst:
    dst.write(efficiency_data, 1)

print(f"Engineering efficiency raster saved to {engefficiency_raster_file}")

# DEFINE THE FUNCTION TO IDENTIFY CONTRIBUTING CELLS
def identify_contributing_cells(flow_accum_data, outlet_cell):
    contributing_cells = set()
    visited_cells = set()
    queue = [outlet_cell]
    drainage_area = 0  # INITIALIZE DRAINAGE AREA

    while queue:
        x, y = queue.pop(0)
        if (x, y) not in visited_cells:
            visited_cells.add((x, y))
            contributing_cells.add((x, y))

            # INCREMENT DRAINAGE AREA BY 1 FOR EACH CONTRIBUTING CELL
            drainage_area += 1

            # GET THE FLOW ACCUMULATION VALUE OF THE CURRENT CELL
            x, y = int(x), int(y)  # ENSURE X AND Y ARE INTEGERS
            flow_accumulation = flow_accum_data[y, x]

            # FIND NEIGHBORING CELLS WITH HIGHER FLOW ACCUMULATION
            neighbors = get_higher_accumulation_neighbors(x, y, flow_accum_data, visited_cells, flow_accumulation)

            # ADD NEIGHBORING CELLS TO THE QUEUE
            queue.extend(neighbors)
    print(f"Outlet Cell: {outlet_cell}, Drainage Area: {drainage_area}")  # ADD THIS LINE TO PRINT THE OUTLET CELL AND DRAINAGE AREA

    return contributing_cells, drainage_area  # RETURN BOTH CONTRIBUTING CELLS AND DRAINAGE AREA

# DEFINE THE FUNCTION TO GET NEIGHBORING CELLS WITH HIGHER FLOW ACCUMULATION
def get_higher_accumulation_neighbors(x, y, flow_accum_data, visited_cells, current_accumulation):
    neighbors = []

    # DEFINE THE NEIGHBORHOOD WINDOW (E.G., 3X3)
    window_size = 3
    half_window = window_size // 2

    # ITERATE OVER THE NEIGHBORHOOD
    for j in range(y - half_window, y + half_window + 1):
        for i in range(x - half_window, x + half_window + 1):
            # CHECK IF THE CELL IS WITHIN THE RASTER BOUNDARIES
            if 0 <= i < flow_accum_data.shape[1] and 0 <= j < flow_accum_data.shape[0]:
                # CHECK IF THE CELL IS NOT IN VISITED_CELLS
                if (i, j) not in visited_cells:
                    # CHECK IF THE CELL HAS HIGHER FLOW ACCUMULATION
                    if flow_accum_data[j, i] > current_accumulation:
                        neighbors.append((i, j))

    return neighbors

# DEFINE THE FUNCTION TO CALCULATE DRAINAGE AREA
def calculate_drainage_area(contributing_cells, cell_size):
    # ASSUMING EACH CELL CONTRIBUTES A UNIT AREA
    return len(contributing_cells) * cell_size**2  # ASSUMING CELL_SIZE IS IN METERS

### POND OUTLETS
# OPEN THE FLOW ACCUMULATION RASTER TO READ ITS DATA AND TRANSFORM
with rasterio.open(flow_accum) as flow_src:
    flow_data = flow_src.read(1)
    flow_transform = flow_src.transform

# OPEN THE OUTPUT RASTER CONTAINING POND BOUNDARIES TO READ ITS DATA AND TRANSFORM
with rasterio.open(engefficiency_raster_file) as output_src:
    output_data = output_src.read(1)
    output_transform = output_src.transform

# OPEN THE SHAPEFILE TO GET ITS CRS
with fiona.open(engineeredsswm, "r") as shapefile:
    crs = shapefile.crs

# ITERATE OVER EACH POLYGON IN THE SHAPEFILE
with fiona.open(engineeredsswm, "r") as shapefile:
    for feature in shapefile:
        # EXTRACT THE GEOMETRY OF THE CURRENT POLYGON
        geometry = feature["geometry"]

        # MASK THE FLOW ACCUMULATION RASTER WITH THE POLYGON BOUNDARY
        mask = geometry_mask([geometry], flow_src.shape, flow_transform, invert=True)
        masked_flow = np.where(mask, flow_data, np.nan)

        # FIND THE MAXIMUM FLOW ACCUMULATION AND ITS LOCATION WITHIN THE POLYGON BOUNDARY
        max_index = np.nanargmax(masked_flow)
        max_flow = np.nanmax(masked_flow)
        max_coordinates = np.unravel_index(max_index, masked_flow.shape)
```

```
            # APPEND THE MAXIMUM FLOW ACCUMULATION AND ITS LOCATION TO THE LIST
            max_flow_accumulations.append({
                'max_flow': max_flow,
                'max_coordinates': max_coordinates
            })

        # PRINT THE MAXIMUM FLOW ACCUMULATION AND ITS LOCATION WITHIN EACH POLYGON BOUNDARY THIS IS THE
INFLOW LOCATION
        for i, max_flow_info in enumerate(max_flow_accumulations):
            print(f"Polygon {i+1}: Maximum flow accumulation = {max_flow_info['max_flow']} at location {max_flow_info['max_coordinates']}")

        # CREATE A NEW EFFICIENCY RASTER WITH CELLS CONTAINING THE MAXIMUM FLOW ACCUMULATIONS SET TO 0
AND ALL OTHER CELLS SET TO 1
        efficiency_data = np.ones_like(output_data, dtype=np.uint8)
        for max_flow_info in max_flow_accumulations:
            max_coordinates = max_flow_info['max_coordinates']
            efficiency_data[max_coordinates] = 0

        # WRITE THE EFFICIENCY RASTER TO A NEW GEOTIFF FILE
        maxcellefficiency_raster_file = os.path.join(engineeredswm_folder, "MaxCellEngineeringEfficiencyRaster.tif")
        with rasterio.open(
            maxcellefficiency_raster_file,
            "w",
            driver="GTiff",
            width=output_src.width,
            height=output_src.height,
            count=1,
            dtype=np.uint8,
            crs=output_src.crs,
            transform=output_src.transform,
        ) as dst:
            dst.write(efficiency_data, 1)

        print(f"New efficiency raster saved to {maxcellefficiency_raster_file}")

        # OPEN THE EFFICIENCY RASTER
        with rasterio.open(maxcellefficiency_raster_file) as src:
            # READ THE RASTER DATA
            efficiency_data = src.read(1)

        # GET THE AFFINE TRANSFORMATION MATRIX
        transform = src.transform

        # GET THE CELL SIZE
        cell_size = transform.a

        # FIND THE COORDINATES OF CELLS WITH A VALUE OF 0
        zero_indices = np.where(efficiency_data == 0)
        zero_coords = [src.xy(row, col) for row, col in zip(zero_indices[0], zero_indices[1])]

        # CONVERT COORDINATES TO POINT GEOMETRIES
        points = [Point(x, y) for x, y in zero_coords]

        # CREATE A GEODATAFRAME WITH THE POINT GEOMETRIES
        pour_points_gdf1 = gpd.GeoDataFrame(geometry=points, crs=src.crs)

        # SAVE THE POUR POINTS AS A SHAPEFILE
        pour_points_file1 = os.path.join(engineeredswm_folder,"PondOutlets.shp")
        pour_points_gdf1.to_file(pour_points_file1)

        print(f"Pour points saved to {pour_points_file1}")

        ## CHANGE THE FLOW DIRECTION RASTER
        # READ THE SHAPEFILE
        gdf = gpd.read_file(engineeredsswm)

        # EXTRACT POINTS FROM EACH POLYGON
        extracted_points = []
        for idx, row in gdf.iterrows():
            polygon = row['geometry']
            boundary = polygon.boundary
            line = LineString(boundary)
            points = [(coord[0], coord[1]) for coord in line.coords]
            extracted_points.extend(points)

        # CONVERT EXTRACTED POINTS TO A GEODATAFRAME
        geometry = [Point(x, y) for x, y in extracted_points]
        crs = gdf.crs  # GET THE COORDINATE REFERENCE SYSTEM FROM THE ORIGINAL GEODATAFRAME
        points_gdf = gpd.GeoDataFrame(geometry=geometry, crs=crs)

        # SAVE THE GEODATAFRAME TO A NEW SHAPEFILE
        pondpoints = os.path.join(engineeredswm_folder, "pondboundarypoints.shp")
        points_gdf.to_file(pondpoints)
        print("Pond points extracted")

        da_output_folder = os.path.join(engineeredswm_folder,"Pond_Drainage_Areas")
        # ENSURE THE OUTPUT FOLDER EXISTS, OR CREATE IT IF IT DOESN'T
        if not os.path.exists(da_output_folder):
            os.makedirs(da_output_folder)

        print("Drainage area raster extraction beginning.")

        output_raster = os.path.join(da_output_folder, 'outletdrainagearea.tif')
        wbt.watershed(flow_dir, pour_points_file1, output_raster)
        output_shapefile = os.path.join(da_output_folder, 'outletdrainagearea.shp')
        wbt.raster_to_vector_polygons(output_raster, output_shapefile)

        # OPEN THE INPUT SHAPEFILE
        driver = ogr.GetDriverByName('ESRI Shapefile')
        dataset = driver.Open(output_shapefile, 1)  # OPEN IN EDIT MODE
        layer = dataset.GetLayer()

        target_srs = osr.SpatialReference()
        target_srs.SetFromUserInput(str(dem_espg2int))

        # SET THE CRS FOR THE DRAINAGE AREA SHAPEFILE
        # ADD A NEW FIELD FOR DRAINAGE AREA
        field_defn = ogr.FieldDefn('SWMDA', ogr.OFTReal)
        layer.CreateField(field_defn)

        # GET THE RASTER GEOTRANSFORM
        raster_dataset = gdal.Open(output_raster)
        gt = raster_dataset.GetGeoTransform()

        # LOOP THROUGH EACH FEATURE IN THE SHAPEFILE
        for feature in layer:
            # GET THE GEOMETRY OF THE FEATURE
            geometry = feature.GetGeometryRef()

            # CALCULATE THE AREA OF THE FEATURE'S GEOMETRY
            area = geometry.GetArea()

            # CONVERT THE DRAINAGE AREA FROM SQUARE METERS TO SQUARE KILOMETERS
            drainage_area_sqkm = area / 1e6  # 1 SQUARE KILOMETER = 1,000,000 SQUARE METERS

            # SET THE DRAINAGE AREA ATTRIBUTE FOR THE FEATURE
            feature.SetField('SWMDA', drainage_area_sqkm)

            # SAVE THE CHANGES
            layer.SetFeature(feature)

        # CLOSE THE DATASET
        dataset = None

        print("Drainage area extraction and recording completed.")

        # OPEN THE POUR POINTS SHAPEFILE
        pour_points_dataset = driver.Open(pour_points_file1, 1)  # OPEN IN EDIT MODE
        pour_points_layer = pour_points_dataset.GetLayer()
```

```
# OPEN THE DRAINAGE AREA SHAPEFILE
drainage_area_dataset = driver.Open(output_shapefile, 0)  # OPEN IN READ-ONLY MODE
drainage_area_layer = drainage_area_dataset.GetLayer()

# GET THE FIELD INDEX FOR SWMDA IN THE DRAINAGE AREA LAYER
swmda_field_index = drainage_area_layer.GetLayerDefn().GetFieldIndex('SWMDA')

# ADD A NEW FIELD FOR DRAINAGE AREA IN THE POUR POINTS SHAPEFILE
pour_points_field_defn = ogr.FieldDefn('SWMDA', ogr.OFTReal)
pour_points_layer.CreateField(pour_points_field_defn)

# LOOP THROUGH EACH POUR POINT FEATURE
for pour_point_feature in pour_points_layer:
    # GET THE GEOMETRY OF THE POUR POINT
    pour_point_geometry = pour_point_feature.GetGeometryRef()

    # RESET SWMDA VALUE TO 0 BY DEFAULT
    pour_point_feature.SetField('SWMDA', 0)

    # LOOP THROUGH EACH DRAINAGE AREA FEATURE
    drainage_area_layer.ResetReading()
    for drainage_area_feature in drainage_area_layer:
        # GET THE GEOMETRY OF THE DRAINAGE AREA FEATURE
        drainage_area_geometry = drainage_area_feature.GetGeometryRef()

        # CHECK IF THE POUR POINT INTERSECTS WITH THE DRAINAGE AREA FEATURE
        if pour_point_geometry.Intersects(drainage_area_geometry):
            # IF THERE IS AN INTERSECTION, ASSIGN THE DRAINAGE AREA VALUE TO THE POUR POINT
            drainage_area_value = drainage_area_feature.GetField(swmda_field_index)
            pour_point_feature.SetField('SWMDA', drainage_area_value)
            break  # EXIT THE LOOP ONCE THE DRAINAGE AREA VALUE IS ASSIGNED

    # SAVE THE CHANGES TO THE POUR POINT FEATURE
    pour_points_layer.SetFeature(pour_point_feature)

# CLOSE THE DATASETS
pour_points_dataset = None
drainage_area_dataset = None

print("Drainage area extraction to pour points shapefile completed.")
engineereddmassoutput = os.path.join(engineeredswm_folder,"EngSWMD8MassFluxOutput.tif")

# # COPY THE ENG SWM EFFICIENCY
# OPEN INPUT RASTER FILE
input_raster4 = gdal.Open(maxcellefficiency_raster_file, gdal.GA_Update)

reprojectedfiles = os.path.join(intermediate_storage_files, "WhiteBoxToolFiles")
if not os.path.exists(reprojectedfiles):
    os.makedirs(reprojectedfiles)

# CREATE OUTPUT RASTER FILE
driver = gdal.GetDriverByName("GTiff")
outputswmeffreproj = os.path.join(reprojectedfiles, "Copy_EngSWMEfficiency.tif")
output_raster4 = driver.CreateCopy(outputswmeffreproj, input_raster4)

# CLOSE INPUT AND OUTPUT RASTER FILES
input_raster4 = None
output_raster4 = None

wbt_engswmfile = None
for filename in os.listdir(reprojectedfiles):
    if filename.startswith("Copy_EngSWMEfficiency") and filename.endswith(
        ".tif"
    ):
        wbt_engswmfile = os.path.join(reprojectedfiles, filename)
        break
if wbt_engswmfile is not None:
    print("Update file found:", wbt_engswmfile)
else:
    wbt_engswmfile = outputswmeffreproj
        print("Using the old file:", wbt_engswmfile)

    output_impervious1 = os.path.join(engineeredswm_folder, "Final_Eng_Imperviousness.tif")

    # IF A IMPERVIOUS RASTER IS PROVIDED, THESE STEPS WILL BE EXECUTED
    if lurasteranalysis == "Y":
        print("Raster of LU Imperviousness is provided")
            # ENSURE INTERMEDIATE STORAGE FOLDER EXISTS
        if not os.path.exists(engineeredswm_folder):
            os.makedirs(engineeredswm_folder)

        # DEFINE THE NEW LOCATION TO SAVE THE RASTER
        impraster = os.path.join(engineeredswm_folder, "LU_Imperviousness_raster.tif")

        # OPEN THE ORIGINAL RASTER FILE
        with rasterio.open(lurasteranalysispath) as src:
            raster_data = src.read(1)  # READ THE FIRST BAND OF THE RASTER
            profile = src.profile  # GET THE METADATA FOR THE RASTER
            # NORMALIZE IMPERVIOUSNESS VALUES (IF VALUES ARE IN PERCENTAGE FORM)
            raster_data = raster_data / 100.0
            raster_data = np.clip(raster_data, 0, 1)  # CLAMP VALUES BETWEEN 0 AND 1

        # SAVE THE RASTER TO THE OUTPUT PATH WITHOUT MODIFICATION
        with rasterio.open(impraster, 'w', **profile) as dst:
            dst.write(raster_data, 1)  # WRITE THE ORIGINAL RASTER DATA TO THE FIRST BAND

        print(f"Raster saved successfully to {impraster}")

## ------------------------------------------------------------------------------------------------------------------------------------------------------
## PREPROCESS THE INPUT RASTERS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # RESOLUTION EXTRACTED AT TOP
        with rasterio.open(dem_clipped_path) as src:
            # GET THE AFFINE TRANSFORM FROM THE METADATA
            transform = src.transform

            # THE RESOLUTION IS STORED IN THE CELLSIZE OF THE AFFINE TRANSFORM
            x_resolution = transform.a
            y_resolution = transform.e

            # THE CELLSIZE IS THE ABSOLUTE VALUE OF THE RESOLUTION, AS IT MAY BE NEGATIVE IF THE RASTER IS
ROTATED
            x_cellsize = abs(x_resolution)
            y_cellsize = abs(y_resolution)

        crs = rasterio.crs.CRS.from_epsg(int(dem_espg2int))

        # ABSORPTION RASTER
        # PREPROCESS THE ABSORPTION RASTER, WE WANT TO HAVE ZERO ABSORPTION
        # OPEN THE RASTER FILE
        with rasterio.open(dem_clipped_path) as src:
            # READ THE DATA INTO A NUMPY ARRAY
            data = src.read(1)

            # SET ALL CELLS TO THE DESIRED VALUE
            data[:] = 0

            # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
            meta = src.meta.copy()
            meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

            absorption_file = os.path.join(engineeredswm_folder, "Absorption.tif")
            # WRITE THE UPDATED ARRAY TO A NEW FILE
            with rasterio.open(absorption_file, "w", **meta) as dst:
                dst.write(data, 1)

        # EFFICIENCY RASTER
        # PREPROCESS THE EFFICIENCY RASTER, WE WANT TO HAVE 1 EFFICIENCY
        with rasterio.open(dem_clipped_path) as src:
            # READ THE DATA INTO A NUMPY ARRAY
```

```
        data = src.read(1)

        # SET ALL CELLS TO THE DESIRED VALUE
        data[:] = 1

        # UPDATE THE METADATA WITH THE NEW SHAPE OF THE ARRAY
        meta = src.meta.copy()
        meta.update({"height": data.shape[0], "width": data.shape[1], "crs": crs})

        efficiency_file = os.path.join(engineeredswm_folder, "Efficiency.tif")
        # WRITE THE UPDATED ARRAY TO A NEW FILE
        with rasterio.open(efficiency_file, "w", **meta) as dst:
            dst.write(data, 1)

## ---------------------------------------------------------------------------------------------------------------------------------------------------
## CROP THE LAND USE TIF'S
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # SHAPEFILE FILE
        ras_lu = impraster

        # READ SHAPEFILE
        gdf = gpd.read_file(basin_polygon_sub_path)

        gdf["count"] = [i + 1 for i, _ in enumerate(gdf.geometry)]
        outputfile = os.path.join(engineeredswm_folder, "basin_selected_count.shp")
        gdf.to_file(outputfile)

        gdf1 = gpd.read_file(outputfile)

        # GDF.CRS
        names = [x for x in gdf1["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

        # READ SHAPEFILE
        gdf3 = gpd.read_file(outputfile)

        # GDF.CRS
        names = [x for x in gdf3["count"]]  # GET COLUMN VALUE FROM SHAPEFILE

        # READ RASTER DATA
        ras_data = rasterio.open(ras_lu)

        # OUTPUT PATH
        out_raster_lu = os.path.join(
            engineeredswm_folder, "LURasterIntermediateSteps"
        )
        if not os.path.exists(out_raster_lu):
            os.makedirs(out_raster_lu)

        # SAVE OUTPUT FILES AS PER SHAPEFILE FEATURES
        for i in range(len(gdf3)):
            geom = []
            coord = shapely.geometry.mapping(gdf3)["features"][i]["geometry"]
            geom.append(coord)
            with rasterio.open(ras_lu) as src:
                out_image, out_transform = rasterio.mask.mask(src, geom, crop=True)
                out_meta = src.meta
                out_meta.update(
                    {
                        "driver": "GTiff",
                        "height": out_image.shape[1],
                        "width": out_image.shape[2],
                        "transform": out_transform,
                    }
                )
                # FILE_PATH = ''
                with rasterio.open(
                    f"{out_raster_lu}/{names[i]}.tif", "w", **out_meta
                ) as dest:
                    dest.write(out_image)

        luraster_files = glob.glob(out_raster_lu + "/*.tif")
        mergedfilelu = os.path.join(engineeredswm_folder, "Final_LU_Raster.tif")

        # OPEN INPUT RASTERS
        src_files_to_mosaic2 = []
        # PRINT(SRC_FILES_TO_MOSAIC2)

        for file in luraster_files:
            src = rasterio.open(file)
            src_files_to_mosaic2.append(src)

        # MERGE INPUT RASTERS INTO ONE
        mosaic, out_trans = rasterio.merge.merge(src_files_to_mosaic2)

        # CLOSE ALL INPUT RASTER FILES
        for src in src_files_to_mosaic2:
            src.close()

        # UPDATE METADATA FOR OUTPUT RASTER
        out_meta = src.meta.copy()
        out_meta.update(
            {
                "driver": "GTiff",
                "height": mosaic.shape[1],
                "width": mosaic.shape[2],
                "transform": out_trans,
            }
        )

        # WRITE OUTPUT RASTER TO DISK
        with rasterio.open(mergedfilelu, "w", **out_meta) as dest:
            dest.write(mosaic)
    ## ---------------------------------------------------------------------------------------------------------------------------------------------------
    ## REPROJECT THE FILES
    ## ---------------------------------------------------------------------------------------------------------------------------------------------------
        reprojectedfiles = os.path.join(engineeredswm_folder, "WhiteBoxToolFiles")
        if not os.path.exists(reprojectedfiles):
            os.makedirs(reprojectedfiles)

        # DEFINE EPSG CODE
        epsg_code = dem_espg2int

        # CREATE SPATIALREFERENCE OBJECT FROM EPSG CODE
        srs = osr.SpatialReference()
        srs.ImportFromEPSG(int(epsg_code))

        # CONVERT SPATIALREFERENCE OBJECT TO WKT STRING
        new_crs_wkt = srs.ExportToWkt()

        # COPY THE DEMS
        # OPEN INPUT RASTER FILE
        input_raster = gdal.Open(dem_clipped_path, gdal.GA_Update)

        # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL
SIZE
        existing_geotransform = input_raster.GetGeoTransform()
        new_geotransform = (
            existing_geotransform[0],
            existing_geotransform[1],
            0,
            existing_geotransform[3],
            0,
            existing_geotransform[5],
        )

        # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
        input_raster.SetGeoTransform(new_geotransform)
        input_raster.SetProjection(new_crs_wkt)

        # CREATE OUTPUT RASTER FILE
```

```
    driver = gdal.GetDriverByName("GTiff")
    outputdemreproj = os.path.join(reprojectedfiles, "Reprojected_DEM.tif")
    output_raster = driver.CreateCopy(outputdemreproj, input_raster)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster = None
    output_raster = None

    # COPY THE ABSORPTION
    # OPEN INPUT RASTER FILE
    input_raster1 = gdal.Open(absorption_file, gdal.GA_Update)

    driver = gdal.GetDriverByName("GTiff")
    outputabsreproj = os.path.join(reprojectedfiles, "Copy_Absorption.tif")
    output_raster1 = driver.CreateCopy(outputabsreproj, input_raster1)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster1 = None
    output_raster1 = None

    # COPY THE EFFICIENCY
    # OPEN INPUT RASTER FILE
    input_raster2 = gdal.Open(efficiency_file, gdal.GA_Update)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputeffreproj = os.path.join(reprojectedfiles, "Copy_Efficiency.tif")
    output_raster2 = driver.CreateCopy(outputeffreproj, input_raster2)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster2 = None
    output_raster2 = None

    # COPY THE LAND USE
    # OPEN INPUT RASTER FILE
    input_raster3 = gdal.Open(mergedfilelu, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL
SIZE
    existing_geotransform3 = input_raster3.GetGeoTransform()
    new_geotransform3 = (
        existing_geotransform3[0],
        existing_geotransform3[1],
        0,
        existing_geotransform3[3],
        0,
        existing_geotransform3[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster3.SetGeoTransform(new_geotransform3)
    input_raster3.SetProjection(new_crs_wkt)

    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputlureproj = os.path.join(reprojectedfiles, "Reprojected_LandUse.tif")
    output_raster3 = driver.CreateCopy(outputlureproj, input_raster3)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster3 = None
    output_raster3 = None

    # COPY THE FLOWACCUM
    # OPEN INPUT RASTER FILE
    input_raster4 = gdal.Open(flow_accum, gdal.GA_Update)

    # GET EXISTING GEOTRANSFORM AND CREATE NEW GEOTRANSFORM WITH SAME ORIGIN BUT DIFFERENT PIXEL
SIZE
    existing_geotransform4 = input_raster4.GetGeoTransform()
    new_geotransform4 = (
        existing_geotransform4[0],
        existing_geotransform4[1],
        0,
        existing_geotransform4[3],
        0,
        existing_geotransform4[5],
    )

    # SET NEW GEOTRANSFORM AND COORDINATE SYSTEM
    input_raster4.SetGeoTransform(new_geotransform4)
    input_raster4.SetProjection(new_crs_wkt)
    # CREATE OUTPUT RASTER FILE
    driver = gdal.GetDriverByName("GTiff")
    outputflowaccumreproj = os.path.join(reprojectedfiles, "Reprojected_FlowAccum.tif")
    output_raster4 = driver.CreateCopy(outputflowaccumreproj, input_raster4)

    # CLOSE INPUT AND OUTPUT RASTER FILES
    input_raster4 = None
    output_raster4 = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CHECK SPATIAL EXTENTS OF THE FILES
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
    # OPEN DEM RASTER
    small_raster = gdal.Open(outputdemreproj)
    small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

    # DICTIONARY TO STORE PROCESSED RASTERS
    raster_dict = {}

    # LOOP THROUGH ALL GEOTIFF FILES IN THE FOLDER
    for file_name in os.listdir(reprojectedfiles):
        if file_name.endswith(".tif"):
            file_path = os.path.join(reprojectedfiles, file_name)

            # OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
            raster = gdal.Open(file_path, gdal.GA_Update)
            raster_gt = raster.GetGeoTransform()
            raster_proj = raster.GetProjection()
            raster_size = (raster.RasterXSize, raster.RasterYSize)

            # CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
            if (
                raster_gt == small_raster.GetGeoTransform()
                and raster_proj == small_raster.GetProjection()
                and raster_size == small_size
            ):
                print(
                    f"{file_name} is already in the correct CRS, resolution, and size."
                )
            else:
                # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
                raster_update_path = os.path.join(
                    reprojectedfiles, f"{file_name.split('.')[0]}_update.tif"
                )
                new_raster = gdal.Translate(
                    raster_update_path,
                    raster,
                    format="GTiff",
                    width=small_size[0],
                    height=small_size[1],
                )
                new_raster.SetGeoTransform(small_raster.GetGeoTransform())
                new_raster.SetProjection(small_raster.GetProjection())
                print(f"{file_name} has been updated.")

            # CLOSE THE RASTER FILE
            raster = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## LOOP THROUGH THE FOLDER AND CHECK IF THERE ARE UPDATE FILES THEN ASSIGN IT VARIABLES
## IF THERE IS NO CHANGE IN CRS/RES/EXTENTS USE THE OLD TIF
```

```
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
    wbt_demfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_DEM_update") and filename.endswith(".tif"):
            wbt_demfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_demfile is not None:
        print("Update file found:", wbt_demfile)
    else:
        wbt_demfile = outputdemreproj
        print("Using the old file:", wbt_demfile)

    wbt_absfile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Absorption_update") and filename.endswith(
            ".tif"
        ):
            wbt_absfile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_absfile is not None:
        print("Update file found:", wbt_absfile)
    else:
        wbt_absfile = outputabsreproj
        print("Using the old file:", wbt_absfile)

    wbt_efffile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_Efficiency_update") and filename.endswith(
            ".tif"
        ):
            wbt_efffile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_efffile is not None:
        print("Update file found:", wbt_efffile)
    else:
        wbt_efffile = outputeffreproj
        print("Using the old file:", wbt_efffile)
    wbt_lufile = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_LandUse_update") and filename.endswith(
            ".tif"
        ):
            wbt_lufile = os.path.join(reprojectedfiles, filename)
            break
    if wbt_lufile is not None:
        print("Update file found:", wbt_lufile)
    else:
        wbt_lufile = outputlureproj
        print("Using the old file:", wbt_lufile)

    dividing_flowaccum = None
    for filename in os.listdir(reprojectedfiles):
        if filename.startswith("Reprojected_FlowAccum_update") and filename.endswith(
            ".tif"
        ):
            dividing_flowaccum = os.path.join(reprojectedfiles, filename)
            break
    if dividing_flowaccum is not None:
        print("Update file found:", dividing_flowaccum)
    else:
        dividing_flowaccum = outputflowaccumreproj
        print("Using the old file:", dividing_flowaccum)
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## USE WBT TO OUTPUT THE D8 MASSFLUX
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
    # BRING IN WHITEBOX TOOLS
    wbt = whitebox.WhiteboxTools()

    # DEFINE THE OUTPUT_FILE
    dmassoutput = os.path.join(engineeredswm_folder, "D8MassFluxOutput.tif")

    # USE D8MASSFLUX TOOL AND OUTPUT A FILE
    wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_efffile, wbt_absfile, dmassoutput)
    #WBT.D8_MASS_FLUX(DEM_CLIPPED_PATH, MERGEDFILELU, EFFICIENCY_FILE, ABSORPTION_FILE, DMASSOUTPUT)
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## SAVE IN LOG SCALE
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
    with rasterio.open(dmassoutput) as src:
        # READ THE RASTER DATA INTO A NUMPY ARRAY
        data = src.read(1)
        # APPLY LOG TRANSFORMATION
        data_log = np.log(data + 1)

        # CREATE NEW RASTER FILE
        profile = src.profile  # META DATA COPY
        profile.update(nodata=np.nan)

        LogScale_D8 = os.path.join(
            engineeredswm_folder, "LogScale_D8MassFlux.tif"
        )
        with rasterio.open(LogScale_D8, "w", **profile) as dst:
            dst.write(data_log, 1)


    # OPTION 1 D8 MASS FLUX FOR THE PONDS WITH EFFICIENCY OF ZERO
    wbt.d8_mass_flux(wbt_demfile, wbt_lufile, wbt_engswmfile, wbt_absfile, engineereddmassoutput) # THIS ONE SETS THE OUTLET TO
ZERO

    #WBT.D8_MASS_FLUX(WBT_DEMFILE, WBT_LUFILE, WBT_EFFFILE, WBT_ABSFILE, ENGINEEREDDMASSOUTPUT) #
THIS ONE DOES NOT CHANGE THE EFFICIENCY

    # DIVIDING!
    # OPEN THE TWO RASTER FILES
    src5 = gdal.Open(engineereddmassoutput)
    src4 = gdal.Open(flow_accum) # DIVIDING BY THE NEW FLOW ACCUMULATION RASTER GENERATED THAT TAKES INTO
ACCOUNT THE PONDS

    # THE FLOW ACCUMUALTION RASTER TAKES INTO ACCOUNT THE PONTS, SO IT IS NOT NECESSARY TO DEFINE THE
RELATIONSHIP BETWEEN
    # THE CONTRIBUTING DRAINAGE AREA AND THE TOTAL DRAINAGE AREA.
    # GET THE SIZE OF THE RASTERS
    cols = src5.RasterXSize
    rows = src5.RasterYSize

    # CREATE AN OUTPUT RASTER WITH THE SAME SIZE AND PROJECTION AS THE INPUT RASTERS
    driver = gdal.GetDriverByName("GTiff")
    out_raster1 = driver.Create(output_impervious1, cols, rows, 1, gdal.GDT_Float32)
    out_raster1.SetProjection(src5.GetProjection())
    out_raster1.SetGeoTransform(src5.GetGeoTransform())

    # READ THE DATA FROM BOTH RASTERS INTO NUMPY ARRAYS
    data3 = src5.ReadAsArray()
    data2 = src4.ReadAsArray()

    # PERFORM ELEMENT-WISE DIVISION OF THE TWO ARRAYS
    result1 = np.divide(data3, data2)

    # SET THE RANGE OF VALUES TO WRITE TO THE OUTPUT RASTER
    data_min = 0
    data_max = 1

    # CLIP THE RESULT ARRAY TO THE SPECIFIED RANGE
    result_clipped1 = np.clip(result1, data_min, data_max)

    # WRITE THE RESULT TO THE OUTPUT RASTER
    out_raster1.GetRasterBand(1).WriteArray(result_clipped1)

    # CLOSE THE RASTERS
    src5 = None
    src4 = None
```

```
out_raster1 = None

print("Divided Rasters Engineered SWM, Imperviousness Found")

with rasterio.open(output_impervious1) as src:
    imp = src.read(1)

# PLOT THE RASTER
plt.figure(figsize=(10, 10))
rasterio.plot.show(imp, transform=src.transform, cmap="terrain")
plt.show()

# OPEN DEM RASTER
small_raster = gdal.Open(dem_clipped_path)
small_size = (small_raster.RasterXSize, small_raster.RasterYSize)

# OPEN THE FILE AND GET ITS GEOTRANSFORM AND PROJECTION
clippedengimperviousraster = gdal.Open(output_impervious1, gdal.GA_Update)
rasterimp_gt1 = clippedengimperviousraster.GetGeoTransform()
rasterimp_proj1 = clippedengimperviousraster.GetProjection()
rasterimp_size1 = (
    clippedengimperviousraster.RasterXSize,
    clippedengimperviousraster.RasterYSize,
)

# CHECK IF THE FILE'S CRS, RESOLUTION, AND SIZE MATCH THOSE OF THE SMALLER RASTER
if (
    rasterimp_gt1 == small_raster.GetGeoTransform()
    and rasterimp_proj1 == small_raster.GetProjection()
    and rasterimp_size1 == small_size
):
    print("Imperviousness is already in the correct CRS, resolution, and size.")
else:
    # CREATE A NEW RASTER WITH THE SAME CRS, RESOLUTION, AND SIZE AS THE SMALLER RASTER
    raster_imp_update1 = os.path.join(
        engineeredswm_folder, "Final_Eng_Imperviousness_update.tif"
    )
    new_raster2 = gdal.Translate(
        raster_imp_update1,
        clippedengimperviousraster,
        format="GTiff",
        width=small_size[0],
        height=small_size[1],
    )
    new_raster2.SetGeoTransform(small_raster.GetGeoTransform())
    new_raster2.SetProjection(small_raster.GetProjection())
    print("Imperviousness has been updated.")

# CLOSE THE RASTER FILES
raster = None

divided_imp1 = None
for filename in os.listdir(engineeredswm_folder):
    if filename.startswith("Final_Eng_Imperviousness_update") and filename.endswith(
        ".tif"
    ):
        divided_imp1 = os.path.join(engineeredswm_folder, filename)
        break
if divided_imp1 is not None:
    print("Update file found:", divided_imp1)
else:
    divided_imp1 = output_impervious1
    print("Using the old file:", divided_imp1)

extract_pts1 = os.path.join(
    engineeredswm_folder, "Urban_Stream_Points_EngSWM_Imperviousness.shp"
)

# READ THE ORIGINAL SHAPEFILE INTO A GEOPANDAS DATAFRAME
gdf = gpd.read_file(extract_pts)

gdf["SWMImp"] = np.nan
gdf["SWMD8Flux"] = np.nan
gdf["SWMLUFloAc"] = np.nan

# OPEN THE RASTER FILE FOR READING
with rasterio.open(divided_imp1) as src4, rasterio.open(
    engineereddmassoutput
) as src5, rasterio.open(flow_accum) as src6:
    # LOOP THROUGH THE POINTS IN THE GEOPANDAS DATAFRAME
    for index, row in gdf.iterrows():
        # GET THE X AND Y COORDINATES OF THE POINT
        point1 = row.geometry
        x, y = point1.x, point1.y

        # GET THE VALUE OF THE RASTER AT THE POINT LOCATION
        engimpvalue = list(src4.sample([(x, y)]))
        endmassfluxvalue = list(src5.sample([(x, y)]))
        flowaccumvalue = list(src6.sample([(x, y)]))

        # UPDATE THE IMPERVIOUSNESS COLUMN WITH THE RASTER VALUE
        gdf.loc[index, "SWMImp"] = engimpvalue[0][0]
        gdf.loc[index, "SWMD8Flux"] = endmassfluxvalue[0][0]
        gdf.loc[index, "SWMLUFloAc"] = flowaccumvalue[0][0]

# WRITE THE UPDATED GEOPANDAS DATAFRAME TO A NEW SHAPEFILE
gdf.to_file(extract_pts1)

print("Impervious Values for Eng SWM Scenario Extracted to Match Stream Points")

# NOW MERGE THE VALUES BASED ON THE COLUMN
points_gdf1 = gpd.read_file(extract_pts1)
lines_gdf3 = gpd.read_file(copy_strm_net)

engurban_segments = os.path.join(
    engineeredswm_folder, "Urban_Eng_SWM_Imperviousness_Attachment.shp"
)

lines_gdf3.to_file(engurban_segments)

lines_gdf4 = gpd.read_file(engurban_segments)

# POINTS_GDF = POINTS_GDF.RENAME(COLUMNS={"PXY": "ENDXY"})
# THE SEGMENTS FILE WE ARE USING ALREADY HAS A PXY COLUMN
common_column = "pXY"

# MERGE THE TWO GEODATAFRAMES BASED ON THE COMMON COLUMN
merged_gdf1 = lines_gdf2.set_index(common_column).join(
    points_gdf1.set_index(common_column)[["SWMImp", "SWMD8Flux", "SWMLUFloAc"]]
)
print(merged_gdf1.columns)

# ATTACH THE DRAINAGE AREAS FROM THE OUTLETS TO THE STREAM SEGMENTS
# DEFINE THE PROXIMITY THRESHOLD
proximity_threshold = 10  # THIS IS THE THRESHOLD TO SEARCH FOR WHERE THE POUR POINTS ARE

# ADD A NEW FIELD FOR DRAINAGE AREA IN SQUARE KILOMETERS
merged_gdf1['SWMDA'] = None

snappedpourpoints = os.path.join(engineeredswm_folder,"SnappedPourPoints.shp")
# STREAMS = OS.PATH.JOIN(WATERSHED_OUTPUT,"EXTRACT_STREAM.TIF")
# WBT.JENSON_SNAP_POUR_POINTS(POUR_POINTS_FILE1, STREAMS, SNAPPEDPOURPOINTS, PROXIMITY_THRESHOLD)

# READ THE POUR POINTS SHAPEFILE
pour_points = gpd.read_file(pour_points_file1)

# INITIALIZE THE CUMULATIVE DRAINAGE AREA COLUMN IF IT DOES NOT EXIST
if 'SWMDA' not in merged_gdf1.columns:
    merged_gdf1['SWMDA'] = 0
```

```
    # INITIALIZE CUMULATIVE DRAINAGE AREA VALUE
    cumulative_swmda = 0

    # INITIALIZE A DICTIONARY TO TRACK SEGMENTS WITH POUR POINTS
    pour_point_segments = {}

    # CREATE A LIST TO STORE THE SNAPPED POUR POINT LOCATIONS
    snapped_locations = []

    # SNAP POUR POINTS TO THE NEAREST STREAM SEGMENT AND UPDATE POUR_POINT_SEGMENTS DICTIONARY
    for idx, pour_point in pour_points.iterrows():
        pour_point_geometry = pour_point.geometry
        distances = merged_gdf1.geometry.apply(lambda line: line.distance(pour_point_geometry))
        nearest_stream_idx = distances.idxmin()

        # CALCULATE THE SNAPPED POINT LOCATION ON THE NEAREST STREAM SEGMENT
        nearest_stream_geom = merged_gdf1.loc[nearest_stream_idx].geometry
        snapped_location = nearest_stream_geom.interpolate(nearest_stream_geom.project(pour_point_geometry))
        snapped_locations.append({'geometry': snapped_location, 'original_index': idx, 'nearest_stream_index': nearest_stream_idx})

        if nearest_stream_idx in pour_point_segments:
            pour_point_segments[nearest_stream_idx] += pour_point['SWMDA']
        else:
            pour_point_segments[nearest_stream_idx] = pour_point['SWMDA']

    # ITERATE OVER EACH SEGMENT IN THE STREAM NETWORK
    for index, segment_row in merged_gdf1.iterrows():
        # CHECK IF THE CURRENT SEGMENT HAS A POUR POINT
        if index in pour_point_segments:
            cumulative_swmda += pour_point_segments[index]
            #print(f"Segment {index}: Cumulative SWMDA updated to {cumulative_swmda}")
        else:
            print(f"Segment {index}: Cumulative SWMDA carried forward as {cumulative_swmda}")

        # ASSIGN THE CUMULATIVE DRAINAGE AREA VALUE TO THE SEGMENT
        merged_gdf1.at[index, 'SWMDA'] = cumulative_swmda

    # CREATE A GEODATAFRAME FOR THE SNAPPED POUR POINT LOCATIONS
    snapped_locations_gdf = gpd.GeoDataFrame(snapped_locations, crs=merged_gdf1.crs)

    # SAVE THE SNAPPED POUR POINT LOCATIONS TO A NEW SHAPEFILE
    snapped_locations_gdf.to_file(snappedpourpoints)

    # PRINT A MESSAGE INDICATING COMPLETION
    print("Drainage Area Values Extracted and Assigned to engurban_segments")
    print("Snapped pour points saved")

    # WRITE THE UPDATED LINES DATAFRAME TO A SHAPEFILE
    merged_gdf1.to_file(engurban_segments)

    print("Impervious, D8 Mass Flux, Flow Accumulation Values Appended to Stream Segments")
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE THE DISCHARGE 2 YR
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    calculation1 = gpd.read_file(engurban_segments)

    # CREATES NEW COLUMN AND SETS DEFAULT VALUE TO 0
    calculation1["SWMLUQ2m3_s"] = 0
    calculation1["SWMLUSP2"] = 0
    calculation1["Wg_Nperm3"] = 9810
    calculation1["SWMLUSSP2"] = 0

    calculation1["SWMLUQ2m3_s"] = pd.to_numeric(calculation1["SWMLUQ2m3_s"], errors="coerce")
    calculation1["SlopeAVGLU"] = pd.to_numeric(calculation1["SlopeAVGLU"], errors="coerce")
    calculation1["Wg_Nperm3"] = pd.to_numeric(calculation1["Wg_Nperm3"], errors="coerce")
    calculation1["RurWidth"] = pd.to_numeric(calculation1["RurWidth"], errors="coerce")
    calculation1["RuralSSP"] = pd.to_numeric(calculation1["RuralSSP"], errors="coerce")

    # ENGINEERED SWM DISCHARGE EQUATION
    # FORMULA FOR DISCHARGE IS Q=CA^X(IA^B) WHERE C, X AND B ARE COEFFICCIENTS, A IS THE DRAINAGE AREA
AND IA IS TOTAL IMPERVIOUSNESS PERCENTAGE
    # A IS IN THE COLUMN TITLED DA_KM2
    # IA IS IN COLUMN TITLED IMP

    a = rQ_a
    b = rQ_b

    c = u_Qs_c
    f = u_Qs_d

    # FUNCTION TO APPLY THE CALCULATION WITH DEBUGGING
    def calculate_flow(row):
        # EXTRACT VALUES
        DA_km2 = row["DA_km2"]
        SWMImp = row["SWMImp"]

        # CHECK FOR MISSING VALUES
        if pd.isna(DA_km2) or pd.isna(SWMImp):
            print("Missing values detected, skipping row.")
            return None  # OR SOME DEFAULT VALUE

        # PRINT ROW VALUES
        #print(f"Processing row: DA_km2={DA_km2}, SWMImp={SWMImp}")

        # CALCULATE INTERMEDIATE RESULTS
        pervious_contribution = a * (DA_km2 * (1 - SWMImp)) ** b
        impervious_contribution = c * (DA_km2 * SWMImp) ** f

        # PRINT INTERMEDIATE RESULTS
        #print(f"Pervious contribution: {pervious_contribution}")
        #print(f"Impervious contribution: {impervious_contribution}")

        # FINAL CALCULATION
        result = pervious_contribution + impervious_contribution
        #print(f"Result: {result}")

        return result

    # APPLY THE CALCULATION
    calculation1["SWMLUQ2m3_s"] = calculation1.apply(calculate_flow, axis=1)


    ## CALCULATE TOTAL POWER
    calculation1["SWMLUSP2"] = calculation1.apply(
        lambda row: (row["SWMLUQ2m3_s"]) * (row["SlopeAVGLU"]) * (row["Wg_Nperm3"]), axis=1
    )

    # ORIGINAL METHOD
    ## CALCULATE SPECIFIC STREAM POWER
    calculation1["SWMLUSSP2"] = calculation1.apply(
        lambda row: row["SWMLUSP2"] / row["RurWidth"], axis=1 #USING PHILLIPS-DESLOGES WIDTH
    )

    # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON)
    pos_val = calculation1["RuralSSP"] > 0
    if pos_val.any():
        calculation1["SWMLUSSPR2"] = calculation1.apply(
            lambda row: row["SWMLUSSP2"] / row["RuralSSP"] if row["RuralSSP"] != 0 else 0,
            axis=1,
        )
    else:
        calculation1["SWMLUSSPR2"] = 0

    # SPECIFIC STREAM POWER
    ### NEWWIDTHURBAN-CALIBRATION
    # ENSURE THAT THE DIFFERENCE IS NON-NEGATIVE
    positive_diff1 = np.maximum(0, (calculation1["SWMLUQ2m3_s"] - calculation1["Q_m3pers"])/calculation1["Q_m3pers"])

    # CALCULATE E5URBWIDTH USING THE NON-NEGATIVE DIFFERENCE
```

```
        calculation1["SWMUrbWidth"] = calculation1["RurWidth"]*(1 + urb_width*(positive_diff1**0.50))

        # CONVERT COLUMNS TO NUMERIC TYPE
        calculation1["SWMUrbWidth"] = pd.to_numeric(calculation1["SWMUrbWidth"], errors="coerce")

        # DISPLAY THE UPDATED DATAFRAME
        print(calculation1)

        pos_val6 = calculation1["SWMUrbWidth"] > 0

        if pos_val6.any():
            calculation1["SWMULSSP2"] = calculation1.apply(
                lambda row: row["SWMLUSP2"] / row["SWMUrbWidth"] if row["SWMUrbWidth"] != 0 else 0,
                axis=1,
            )
        else:
            calculation1["SWMULSSP2"] = 0

        # SSPR CALCULATION (2YR LU URBAN - RURAL COMPARISON) ASHLUSSP2/RURSSPASH
        pos_val7 = calculation1["RuralSSP"] > 0
        if pos_val7.any():
            calculation1["SWMULSPR2"] = calculation1.apply(
                lambda row: row["SWMULSSP2"] / row["RuralSSP"]
                if row["RuralSSP"] != 0
                else 0,
                axis=1,
            )
        else:
            calculation1["SWMULSPR2"] = 0

        output_file1 = os.path.join(
            watershed_output, "Final_Land_Use_EngSWM_Stream_Power_Output.shp"
        )
        calculation1.to_file(output_file1, driver="ESRI Shapefile")

    print(counter)

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_output, 'stream_points_fields_renamed.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)

    lu_analysis_file = None

## END OF MULTIPLE LU ANALYSIS SCRIPT
end3 = time.time()
print("The time of execution of LU Multiple is :", (end3 - start3), "seconds")
gc.collect()
##########################################################################################################################
##########################################################
##########################################################################################################################
##########################################################
## MULTIPLE WATERSHED AND HECRAS SELECTED
start4 = time.time()

if check_boundary > 0 and hec_analysis == "Y":
    counter == 2

if (counter == 2 and hec_analysis == "Y"
):  # THIS IS THE INDICATOR THAT MULTIPLE BOUNDARIES IS RUNNING
    print("HECRAS STARTING FOR MULTIPLE WATERSHED")

    # COUNT NUMBER OF WATERSHEDS TO RUN THE ANALYSIS FOR
    folder_count = 0

    # ITERATE OVER THE ITEMS IN THE DIRECTORY
    for watershed in os.scandir(hec_watershed):
        if watershed.is_dir():
            folder_count += 1

    print("Number of folders:", folder_count)

    # NUMBER OF TIMES THE PROGRAM HAS RUN
    num = 1

    # LOOPING THROUGH THE HEC-RAS FOLDERS THAT ARE PRESENT
    if num <= folder_count:
        print("MULTIPLE WATERSHED, THIS WILL LOOP MULTIPLE TIMES")
        print("Number of times program ran:", num)
        print("Number of folders:", folder_count)

        if num == folder_count:
            counter = 1
            print(
                "Number of times the loop has run is greater than the folder count so the loop will stop"
            )
        else:
            print("Program ran less times than the number of folders")

        num += 1

        for watershed in os.listdir(hec_watershed):
            watershed_path = os.path.join(hec_watershed, watershed)
            watershed_output = os.path.join(data_dir, "HECRAS Analysis")
            if os.path.exists(watershed_output):
                print("FOLDER CREATED")
            else:
                intermediate_dir = os.makedirs(watershed_output)

            watershed_outputs_id = os.path.join(watershed_output, watershed)
            if os.path.exists(watershed_outputs_id):
                print("FOLDER CREATED")
            else:
                intermediate_dir = os.makedirs(watershed_outputs_id)

            wbt.set_working_dir(watershed_outputs_id)

            if os.path.isdir(
                watershed_path
            ):  # IF THE SUBFOLDER PATH EXISTS THEN PERFORM THE SCRIPT
## ---------------------------------------------------------------------------------------------------------------------------
## REDEFINE THE VARIABLES
## ---------------------------------------------------------------------------------------------------------------------------
                hec_csv = os.path.join(
                    watershed_path, "hec_csv.csv"
                )  # HECRAS STATIONS DATA
                hec_point_raw = os.path.join(
                    watershed_path, "hec_shp.shp"
                )  # HECRAS AS POINTS SHAPEFILE

                bridge_hec_csv = os.path.join(
                    watershed_path, "bridge_hec_csv.csv"
                )  # HECRAS BRIDGE DATA
                bridge_point_raw = os.path.join(
                    watershed_path, "bridge_point_raw.shp"
                )  # HECRAS BRIDGE DATA AS SHAPEFILES

                culv_hec_csv = os.path.join(
                    watershed_path, "culv_hec_csv.csv"
                )  # CULVERT DATA
                culv_point_raw = os.path.join(
                    watershed_path, "culv_point_raw.shp"
                )  # CULVERT DATA AS A SHAPEFILE

                # OPEN THE DEM RASTER FILE
                with rasterio.open(dem) as src:
                    # GET THE CRS OF THE RASTER FILE
                    crs = src.crs
```

```
    # USE PYPROJ TO DETECT THE EPSG CODE OF THE CRS
    dem_espg2int = pyproj.CRS.from_string(crs.to_string()).to_epsg()

    print(f"EPSG code: {dem_espg2int}")

# READ THE HEC-RAS POINTS
hecpoint_src = gpd.read_file(hec_point_raw)

# GET THE CRS OF THE HEC-RAS POINTS
hecpoint_src_prj = hecpoint_src.crs

# GET THE EPSG CODE OF THE CRS
hecpoint_src_prj_epsg_int = hecpoint_src_prj.to_epsg()
print(f"EPSG code: {hecpoint_src_prj_epsg_int}")

if dem_espg2int != hecpoint_src_prj_epsg_int:
    print("Reprojecting river stations")
    hecpoint_src_copy = hecpoint_src.to_crs(epsg=dem_espg2int)

    hecpoint_prj_path = str(data_dir) + str("/hecpoint_prj.shp")
    hecpoint_src_copy.to_file(
        hecpoint_prj_path, driver="ESRI Shapefile"
    )
    hec_point = hecpoint_prj_path

    # SET THE OUTPUT CRS FOR THE FINAL RESULTS
    output_crs = dem_espg2int
else:
    print("Will not reproject river stations")
    hec_point = hec_point_raw

    # SET THE OUTPUT CRS FOR THE FINAL RESULTS
    output_crs = dem_espg2int

# BRIDGE ANALYSIS - REPROJECT POINT FEATURES
try:
    if bridge_analysis == "Y":
        bridgept_src = geopandas.read_file(bridge_point_raw)
        bridgept_src_prj = bridgept_src.crs

        # ELSE:
        print("Will not reproject bridge stations")
        bridge_point = bridge_point_raw
except:
    print("Bridge analysis fails", file=sys.stderr)

# CULVERT ANALYSIS
try:
    if culvert_analysis == "Y":
        culvpt_src = geopandas.read_file(culv_point_raw)
        culvpt_src_prj = culvpt_src.crs

        # ELSE:
        print("Will not reproject culvert stations")
        culv_point = culv_point_raw
except:
    print("Culvert analysis fails", file=sys.stderr)

# FOR SUBWATERSHED ANALYSIS ONLY
if subwatershed_analysis == "Y":
    subwshed_src = geopandas.read_file(subwshed_raw)
    subwshed_src_prj = subwshed_src.crs
    subwshed_src_prj_epsg_str = subwshed_src_prj["init"].split(":")[1]
    subwshed_src_prj_epsg_int = int(subwshed_src_prj_epsg_str)

    if dem_obj_prj_espg_str != subwshed_src_prj_epsg_str:
        print("Reprojecting subwatershed")
        # PYEPSG.GET(DEM_OBJ_PRJ_ESPG_STR)
        subwshed_src_copy = subwshed_src.to_crs(epsg=dem_espg2int)
        subwshed_prj_path = str(data_dir) + str("/subwatershed_prj.shp")
        subwshed_src_copy.to_file(
            subwshed_prj_path, driver="ESRI Shapefile"
        )

        subwshed_poly = subwshed_prj_path

    else:
        print("Will not reproject subwatershed")
        subwshed_poly = subwshed_raw

print("HEC-RAS STARTING FOR MULTIPLE WATERSHEDS")
if hec_analysis == "Y" or bridge_analysis == "Y" or culvert_analysis == "Y":
## ------------------------------------------------------------------------------------------------------------------------------------------
## CLIP DEM TO BASIN
## ------------------------------------------------------------------------------------------------------------------------------------------
    # MAKE COPY OF HEC POINTS
    wbt.set_working_dir(watershed_outputs_id)
    wbt.verbose = False

    if not os.path.exists(watershed_outputs_id):
        os.makedirs(watershed_outputs_id)

    hec_analysis_file = os.path.join(watershed_outputs_id, "HECRAS_Analysis.gpkg")

    hec_point_forbasin = geopandas.read_file(hec_point)
    basin_hecpts_path = os.path.join(watershed_outputs_id, "basin_hecpts.shp")
    hec_point_forbasin = hec_point_forbasin.loc[
        :, hec_point_forbasin.columns.isin(["REACH","RIVER_STA", "geometry"])
    ]
    hec_point_forbasin.to_file(basin_hecpts_path)
    layer_name = "basin_hecpts"
    hec_point_forbasin.to_file(hec_analysis_file, layer=layer_name, driver="GPKG")

    basin_raster_path = os.path.join(data_dir, "basins.tif")
    # EXTRACT ALL RASTER VALUES TO STREAM POINTS
    print("Extracting raster values")
    wbt.verbose = True
    wbt.extract_raster_values_at_points(basin_raster_path, basin_hecpts_path)
    wbt.verbose = False

    # OPEN HECRAS POINTS FILE
    basin_hecpts_df = geopandas.read_file(basin_hecpts_path)
    before_last = len(basin_hecpts_df.columns) - 2
    name_map = {basin_hecpts_df.columns[before_last]: "BasinValue"}
    basin_hecpts_df.rename(columns=name_map, inplace=True)

    # GET MOST FREQUEST BASIN VALUE
    basin_valuemode = basin_hecpts_df["BasinValue"].mode()
    basin_valuemode_int = basin_valuemode[0]
    print("basin_value_mod_int: ", basin_valuemode_int)

    # CONVERT BASIN TO POLYGON
    basin_polygon_path = os.path.join(
        watershed_outputs_id, "basins_polygon.shp"
    )

    mask = None
    with rasterio.Env():  # UPDATED 17/6/2019 CHANGED RASTERIO.DRIVERS() TO RASTERIO.ENV()
        with rasterio.open(basin_raster_path) as src:
            image = src.read(1)  # FIRST BAND
            results = (
                {"properties": {"Value": v}, "geometry": s}
                for i, (s, v) in enumerate(
                    shapes(image, mask=mask, transform=src.transform)
                )
            )  # UPDATED 17/6/2019 CHANGED SRC.AFFINE TO SRC.TRANSFORM
    geoms = list(results)
    polygonized_basin_df = geopandas.GeoDataFrame.from_features(geoms)
    polygonized_basin_df.to_file(basin_polygon_path, driver="ESRI Shapefile")
```

```
        layer_name = "basins_polygon"
        polygonized_basin_df.to_file(hec_analysis_file, layer=layer_name, driver="GPKG")

        # SELECT BASIN
        basin_polygon_sub_path = os.path.join(
            watershed_outputs_id, "basin_selected.shp"
        )
        polygonized_basin_sub_df = polygonized_basin_df.loc[
            polygonized_basin_df["Value"] == float(basin_valuemode_int)
        ]
        polygonized_basin_sub_df.crs = output_crs
        polygonized_basin_sub_df.to_file(
            basin_polygon_sub_path, driver="ESRI Shapefile"
        )

        layer_name = "basin_selected"
        polygonized_basin_sub_df.to_file(hec_analysis_file, layer=layer_name, driver="GPKG")

        # CLIP DEM
        dem_clipped_path = os.path.join(
            watershed_outputs_id, "breached_dem_clipped.tif"
        )
        overallbreached_dem = os.path.join(data_dir, "breached_dem.tif")

        with rasterio.open(overallbreached_dem) as src:
            out_image, out_transform = rasterio.mask.mask(src, polygonized_basin_sub_df.geometry.values, crop=True)
            out_meta = src.meta.copy()
            out_meta.update({
                "height": out_image.shape[1],
                "width": out_image.shape[2],
                "transform": out_transform
            })
            with rasterio.open(dem_clipped_path, "w", **out_meta) as dst:
                dst.write(out_image)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'basins_polygon.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## GENERAL WORKFLOW GET FLOW DIRECTION (D8 METHOD)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        gen_flow_dir = os.path.join(watershed_outputs_id, "flow_dir.tif")
        wbt.d8_pointer(dem_clipped_path, gen_flow_dir)

        # GET FLOW ACCUMULATION
        gen_flow_accum = os.path.join(watershed_outputs_id, "flow_accum.tif")
        wbt.d8_flow_accumulation(dem_clipped_path, gen_flow_accum, out_type="cells")

        # EXTRACT STREAMS BASED ON DRAINAGE THRESHOLD
        if pruningda == None:
            gen_extr_strm = os.path.join(watershed_outputs_id, "extract_stream_initial.tif")
            wbt.extract_streams(gen_flow_accum, gen_extr_strm, initial_dathres)
        else:
            gen_extr_strm = os.path.join(watershed_outputs_id, "extract_stream_usprune.tif")
            wbt.extract_streams(gen_flow_accum, gen_extr_strm, (pruningda/cell_size))

        # IF THE USER DOESNT GIVE A DRAINAGE AREA VALUE

        # GET STREAM VECTOR FROM RASTER STREAM
        gen_strm_vec = os.path.join(watershed_outputs_id, "stream_vector_initial.shp")
        wbt.raster_streams_to_vector(gen_extr_strm, gen_flow_dir, gen_strm_vec)

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_vector_initial'
        gdf = gpd.read_file(gen_strm_vec)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        initial_dem_points_path = os.path.join(watershed_outputs_id, "dem_points_initial.shp")
        wbt.raster_to_vector_points(gen_extr_strm, initial_dem_points_path)

        # SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
        initial_dem_points = geopandas.read_file(initial_dem_points_path)

        # WRITE TO A GEOPACKAGE
        print("Writing DEM Points Initial Output")
        layer_name = 'dem_points_initial'
        initial_dem_points.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        initial_stream_point_path = os.path.join(
            watershed_outputs_id, "stream_points_initial.shp"
        )
        initial_dem_points[initial_dem_points["VALUE"] == 1].to_file(
            initial_stream_point_path, driver="ESRI Shapefile"
        )
        # WRITE TO A GEOPACKAGE
        print("Writing Stream Points Initial Output")
        layer_name = 'stream_points_initial'
        gdf = gpd.read_file(initial_stream_point_path)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## STREAM PRUNING: REDEFINE STREAM THRES BASED ON MIN FLOW ACCUM OF HEC POINTS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        #IF INITIAL_DATHRES_PARAM == 1000:
        # EXTRACT FLOW ACCUMULATION TO STREAM POINTS
        wbt.extract_raster_values_at_points("flow_accum.tif", initial_stream_point_path)

        strm_point_faccum_gdf = geopandas.read_file(initial_stream_point_path)
        before_last = len(strm_point_faccum_gdf.columns) - 2
        name_map = {strm_point_faccum_gdf.columns[before_last]: "Faccum"}
        strm_point_faccum_gdf.rename(columns=name_map, inplace=True)

        # OPEN HEC RAS POINTS TO CREATE BUFFER AROUND THEM
        hec_pts_buffer_gdf = geopandas.read_file(basin_hecpts_path)
        hec_pts_buffer_gdf["geometry"] = hec_pts_buffer_gdf.geometry.buffer(50)

        hec_pts_buffer_gdf = hec_pts_buffer_gdf.rename(columns={"FID": "RIVER_STA"})

        buffers_hec_path = os.path.join(watershed_outputs_id, "hecpt_buffers50m.shp")
        hec_pts_buffer_gdf.to_file(buffers_hec_path, driver="ESRI Shapefile")
        print("Writing HECPT Buffers Output")

        # WRITE TO A GEOPACKAGE
        layer_name = 'hecpt_buffers50m'
        gdf = gpd.read_file(buffers_hec_path)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # JOIN POINTS TO BUFFER
        point_in_buffer = geopandas.sjoin(hec_pts_buffer_gdf,strm_point_faccum_gdf, how='left')

        # DISPLAY THE COLUMN NAMES
        column_names = point_in_buffer.columns.tolist()
        print("Column names:", column_names)

        # IF USER GIVES AN ACTUAL THRESHOLD THEY WANT, SKIP BRANCHIGNG AND REPLACE
FACCUM_THRES_25PCNT WITH THEIR VALUE
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## BRANCHING
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # GROUP BY "RIVER_STA" AND CALCULATE THE MAXIMUM "FACCUM" AT EACH RIVER STATION
        point_in_buffer_max = point_in_buffer.groupby("RIVER_STA")["Faccum"].apply(lambda x: x.max())

        if pruningda == None:
            #PLOT FOR ALL OF THE POINTS NOT SEPERATED BY REACH ID
            point_in_buffer = point_in_buffer.dropna()
            point_in_buffer = point_in_buffer[point_in_buffer['Faccum'] != 0]

            # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA
            x_coordinates = point_in_buffer['RIVER_STA']
            flow_accumulation = point_in_buffer['Faccum']
```

```
# PLOT THE X-COORDINATE VS. FLOW ACCUMULATION
plt.scatter(x_coordinates, flow_accumulation)

# PERFORM LINEAR REGRESSION
slope, intercept, r_value, p_value, std_err = linregress(x_coordinates, flow_accumulation)
# ADD A REGRESSION LINE TO THE PLOT
plt.plot(x_coordinates, intercept + slope * x_coordinates, color='red', label='Regression Line')

print(intercept + slope * x_coordinates)
print("Intercept:", intercept)
print(x_coordinates, flow_accumulation)

# CUSTOMIZE THE PLOT
plt.xlabel('River Station')
plt.ylabel('Flow Accumulation')
#PLT.TEXT(0.1, 0.9, F'INTERCEPT: {INTERCEPT}', TRANSFORM=PLT.GCA().TRANSAXES, FONTSIZE=7)
plt.title(f'X-coordinate vs. Flow Accumulation for all Points')
plt.legend()

# SAVE THE PLOT AS AN lMAGE FILE
plot_filename = os.path.join(watershed_output,"HECAll_Reaches.tif")
plt.savefig(plot_filename, format="tif")

# SHOW OR SAVE THE PLOT
plt.show()

# # CREATE A LIST TO STORE DATA FOR LATER DATAFRAME CREATION
# RESULT_DATA = []

# CREATE AN EMPTY DATAFRAME TO STORE THE MAXIMUM FLOW ACCUMULATIONS
min_flow_table = pd.DataFrame(columns=['REACH', 'Min_Flow_Accumulation'])

# PLOTS FOR EACH INDIVIDUAL REACH ID
# EXTRACT UNIQUE REACH NUMBERS FROM YOUR DATA
unique_reach_numbers = point_in_buffer['REACH'].unique()

# PRINT THE UNIQUE REACH NUMBERS FOR REFERENCE
print("Unique REACH Numbers:", unique_reach_numbers)

# LOOP THROUGH EACH UNIQUE REACH NUMBER
for reach_number in unique_reach_numbers:
    # FILTER DATA BASED ON THE CURRENT REACH NUMBER
    filtered_data = point_in_buffer[point_in_buffer['REACH'] == reach_number]

    # FILTER OUT DATA POINTS WITH ZERO FLOW ACCUMULATION
    filtered_data = filtered_data[filtered_data['Faccum'] != 0]

    # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING
    if not filtered_data.empty:

        # CALCULATE THE MAXIMUM FLOW ACCUMULATION WITHIN THE FILTERED DATA FOR EACH RIVER_STA
        max_flow_per_station = filtered_data.groupby("RIVER_STA")["Faccum"].max().reset_index()

    # CHECK IF THERE ARE ANY DATA POINTS LEFT AFTER FILTERING WITHIN EACH RIVER_STA GROUP
        if not max_flow_per_station.empty:
            # EXTRACT X-COORDINATE AND FLOW ACCUMULATION DATA FROM MAX_FLOW_PER_STATION
            x_coordinates = max_flow_per_station['RIVER_STA'].values.reshape(-1, 1)
            flow_accumulation = max_flow_per_station['Faccum']

            # CREATE A NEW PLOT FOR EACH REACH NUMBER
            plt.figure()
            plt.scatter(x_coordinates, flow_accumulation)

            # CREATE A LINEAR REGRESSION MODEL
            regr = linear_model.LinearRegression()
            regr.fit(x_coordinates, flow_accumulation)

            # MAKE PREDICTIONS USING THE X FOR LINEAR LINE
            Y_pred = regr.predict(x_coordinates)

            # ASSIGN THE Y_PRED VALUES TO A NEW COLUMN 'Y_PRED' IN MAX_FLOW_PER_STATION
            max_flow_per_station['Y_pred'] = Y_pred

            # REPLACE NEGATIVE VALUES IN COLUMN 'Y_PRED' WITH VALUES FROM COLUMN 'FACCUM'
            mask = max_flow_per_station['Y_pred'] < 0
            max_flow_per_station.loc[mask, 'Y_pred'] = max_flow_per_station.loc[mask, 'Faccum']

            # PLOT FLOW ACCUMULATION AGAINST RIVER STATION
            plt.scatter(x_coordinates, flow_accumulation, color="black")
            # ADD A REGRESSION LINE TO THE PLOT
            plt.plot(x_coordinates, Y_pred, color='red', label='Regression Line')

            # CUSTOMIZE THE PLOT
            plt.xlabel('River Station')
            plt.ylabel('Flow Accumulation')
            plt.title(f'X-coordinate vs. Flow Accumulation for REACH {reach_number}')
            plt.legend()

            # SAVE THE PLOT AS AN IMAGE FILE
            plot_filename = os.path.join(watershed_output, f"hecreach_{reach_number}_plot.tif")
            plt.savefig(plot_filename, format="tif")

            # SHOW THE CURRENT PLOT
            plt.show()

        #MINIMUM FLOW ACCUMULATION OF THE REACH
            print(f'{reach_number}',max_flow_per_station)
            min_flow_accumulation = max_flow_per_station['Y_pred'].min()
            print(f'{reach_number}', min_flow_accumulation)

            # APPEND THE RESULT TO THE MAX_FLOW_TABLE DATAFRAME USING CONCAT
            min_flow_table = pd.concat([min_flow_table, pd.DataFrame({'REACH': [reach_number],'Min_Flow_Accumulation':
[min_flow_accumulation]})], ignore_index=True)

    # FIND THE MINIMUM FACCUM_MIN_YPRED
    min_faccum_min_ypred = min_flow_table["Min_Flow_Accumulation"].min()

    # CALCULATE FACCUM_THRES_25PCNT
    if min_faccum_min_ypred < 0:
        faccum_thres_25pcnt = initial_dathres_param
    else:
        faccum_thres_25pcnt = int(min_faccum_min_ypred)

    # PRINT THE RESULTS
    print("Result Table:")
    print(min_flow_table)
    # ASSUMING MIN_FLOW_TABLE IS A PANDAS DATAFRAME
    path = os.path.join(watershed_outputs_id,"min_flow_table.csv")
    min_flow_table.to_csv(path, index=False)
    print("Minimum faccum_min_ypred:", min_faccum_min_ypred)
    print("faccum_thres_25pcnt:", faccum_thres_25pcnt)

else:
    faccum_thres_25pcnt = round(pruningda/cell_size)
    print(faccum_thres_25pcnt)

# EXTRACT STREAM
wbt.extract_streams("flow_accum.tif", "extract_stream.tif", faccum_thres_25pcnt)
print("Streams have been extracted with a threshold applied")

# GET POINTS FROM RASTER STREAM BASED ON VALUE = 1
wbt.raster_to_vector_points("extract_stream.tif", "dem_points.shp")

# SELECT ONLY POINTS WHICH FALL ON STREAM USING GEOPANDAS
dem_points_path = os.path.join(watershed_outputs_id,"dem_points.shp")
dem_points = geopandas.read_file(dem_points_path)

# WRITE TO A GEOPACKAGE
layer_name = 'dem_points'
gdf = gpd.read_file(gen_strm_vec)
```

```
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        new_stream_point_path = os.path.join(watershed_outputs_id,"stream_points.shp")
        dem_points[dem_points["VALUE"] == 1].to_file(
            new_stream_point_path, driver="ESRI Shapefile"
        )
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_points'
        gdf = gpd.read_file(new_stream_point_path)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # GET STREAM VECTOR FROM RASTER STREAM
        wbt.raster_streams_to_vector(
            "extract_stream.tif", "flow_dir.tif", "stream_vector_pruned.shp"
        )

        # CLIP STREAM VECTOR TO SUBWATERSHED ANALYSIS
        if subwatershed_analysis == "Y":
            subwshed_df = geopandas.read_file(subwshed_poly)

            stream_vector_pruned_path = os.path.join(watershed_outputs_id,"stream_vector_pruned.shp")
            stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
            subwshed_stream_df = geopandas.sjoin(
                stream_vector_pruned_df, subwshed_df, op="intersects"
            )
            subwshed_stream_list = subwshed_stream_df["FID"].tolist()
            # SUBSET BY LIST OF LINES ONLY WITHIN SUBWATERSHED
            subwshed_stream_selected_df = stream_vector_pruned_df[
                stream_vector_pruned_df["FID"].isin(subwshed_stream_list)
            ]
            stream_vector_path = os.path.join(watershed_outputs_id,"stream_vector.shp")
            subwshed_stream_selected_df.to_file(stream_vector_path, driver="ESRI Shapefile")

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_vector'
            gdf = gpd.read_file(gen_strm_vec)
            gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')


        else:
            stream_vector_pruned_path = os.path.join(watershed_outputs_id,"stream_vector_pruned.shp")
            stream_vector_pruned_df = geopandas.read_file(stream_vector_pruned_path)
            stream_vector_path = os.path.join(watershed_outputs_id,"stream_vector.shp")
            stream_vector_pruned_df.to_file(stream_vector_path, driver="ESRI Shapefile")

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_vector_pruned'
            gdf = gpd.read_file(stream_vector_pruned_path)
            gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'stream_vector_pruned.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
                os.remove(file_path)

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_vector'
            gdf = gpd.read_file(stream_vector_path)
            gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## EXTRACT ALL RASTER VALUES TO STREAM POINTS
## GET STREAM LINK CLASS
## CLASSIFICATION KEY
## VALUE  CLASS
## 1      EXTERIOR LINK
## 2      INTERIOR LINK
## 3      SOURCE NODE (HEADWATER)
## 4      LINK NODE (CONFLUENCE)
## 5      SINK NODE (OUTLET)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
            wbt.stream_link_class(
                "flow_dir.tif", "extract_stream.tif", "stream_link_class.tif"
            )

            # GET UNIQUE ID FOR EACH STREAM LINK CLASS
            wbt.stream_link_identifier(
                "flow_dir.tif", "extract_stream.tif", "stream_link_id.tif"
            )

            # GET LENGTH FOR EACH STREAM LINK CLASS
            wbt.stream_link_length(
                "flow_dir.tif", "stream_link_id.tif", "stream_link_length.tif"
            )

            # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
            wbt.tributary_identifier(
                "flow_dir.tif", "extract_stream.tif", "tributary_id.tif"
            )

            # GET UNIQUE ID FOR EACH TRIBUTARY CLASS
            wbt.strahler_stream_order(
                "flow_dir.tif", "extract_stream.tif", "strahler_order.tif"
            )

            # GET DISTANCE TO OUTLET
            wbt.distance_to_outlet(
                "flow_dir.tif", "extract_stream.tif", "distance_to_outlet.tif"
            )

            # GET THE DISTANCE TO THE FURTHEST UPSTREAM CHANNEL HEAD FOR EACH STREAM CELL.
            wbt.farthest_channel_head(
                "flow_dir.tif", "extract_stream.tif", "distance_to_headwater.tif"
            )

            # GET STREAM SLOPE ALONG RASTER STREAM
            # WBT.STREAM_SLOPE_CONTINUOUS("FLOW_DIR.TIF", "STREAM_LINK_ID.TIF", DEM , "STREAM_SLOPE.TIF")

            # EXTRACT ALL RASTER VALUES TO SELECTED STREAM POINTS
            wbt.extract_raster_values_at_points(

"distance_to_headwater.tif;distance_to_outlet.tif;stream_link_class.tif;stream_link_id.tif;stream_link_length.tif;tributary_id.tif;strahler_order.tif;f
low_accum.tif",
                new_stream_point_path,
            )

            # EXTRACT RASTER DEM VALUES TO STREAM POINTS
            src_filename = dem
            shp_filename = new_stream_point_path

            src_ds = gdal.Open(src_filename)
            gt = src_ds.GetGeoTransform()
            rb = src_ds.GetRasterBand(1)

            ds = ogr.Open(shp_filename, 1)
            lyr = ds.GetLayer()
            lyr.CreateField(ogr.FieldDefn("Elev", ogr.OFTReal))
            for feat in lyr:
                geom = feat.GetGeometryRef()
                mx, my = geom.GetX(), geom.GetY()  # COORD IN MAP UNITS

                # CONVERT FROM MAP TO PIXEL COORDINATES.
                # ONLY WORKS FOR GEOTRANSFORMS WITH NO ROTATION.
                px = int((mx - gt[0]) / gt[1])  # X PIXEL
                py = int((my - gt[3]) / gt[5])  # Y PIXEL

                intval = rb.ReadAsArray(px, py, 1, 1)
                elev = intval[0]
                for value in elev:
                    v = float(value)
```

```
            # PRINT(V)

        # UPDATE NEW FIELD
        lyr.SetFeature(feat)
        feat.SetField("Elev", v)
        lyr.SetFeature(feat)

    # RENAME FIELDS
    stream_point = geopandas.read_file(new_stream_point_path)

    name_map = {
        stream_point.columns[2]: "DistUp",
        stream_point.columns[3]: "DistDown",
        stream_point.columns[4]: "LinkClass",
        stream_point.columns[5]: "LinkID",
        stream_point.columns[6]: "LinkLength",
        stream_point.columns[7]: "TribID",
        stream_point.columns[8]: "Strahler",
        stream_point.columns[9]: "FlowAccum",
        stream_point.columns[10]: "Elev",
    }
    stream_point.rename(columns=name_map, inplace=True)

    # GET XY COORDINATES
    stream_point["X"] = stream_point.geometry.x
    stream_point["Y"] = stream_point.geometry.y
    stream_point["pXY"] = (
        stream_point["X"].map(str) + "," + stream_point["Y"].map(str)
    )

    # SAVE AS SHAPEFILE
    stream_point_field_rename_path = os.path.join(watershed_outputs_id,"stream_points_fields_renamed.shp")
    stream_point.to_file(stream_point_field_rename_path, driver="ESRI Shapefile")

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_points_fields_renamed'
    gdf = gpd.read_file(stream_point_field_rename_path)
    stream_point.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## 2) STREAM SEGMENTATION
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    print("Stream segmentation beginning")

    stream_point = os.path.join(watershed_outputs_id,"stream_points.shp")
    stream_vector = os.path.join(watershed_outputs_id,"stream_vector.shp")

    # CREATE FUNCTION TO SPLIT LINE AT POINTS
    def split_line_at_points(line, point):
        gdf_line = line
        gdf_points = point

        # JOIN ALL LINE AND POINTS
        line = gdf_line.geometry.unary_union
        coords = gdf_points.geometry.unary_union

        # SNAP AND SPLIT COORDS ON LINE
        # RETURNS GEOMETRYCOLLECTION
        split_line = shapely.ops.split(line, coords)

        # CHECK IF SPLIT_LINE IS A GEOMETRYCOLLECTION
        if isinstance(split_line, shapely.geometry.collection.GeometryCollection):
            # ITERATE OVER INDIVIDUAL GEOMETRIES IN THE COLLECTION
            segments = [feature for feature in split_line.geoms]
        else:
            segments = [split_line]

        # TRANSFORM GEOMETRY COLLECTION TO GEODATAFRAME
        gdf_segments = geopandas.GeoDataFrame(
            list(range(len(segments))), geometry=segments
        )
        gdf_segments.columns = ["index", "geometry"]

        # SAVE SEGMENTS
        stream_segments = os.path.join(watershed_outputs_id,"stream_segments.shp")
        gdf_segments.to_file(stream_segments, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments'
        gdf = gpd.read_file(stream_segments)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # RETURN GDF_SEGMENTS

    split_line_at_points(gpd.read_file(stream_vector), gpd.read_file(stream_point))
    # 3 MINS PROCESSING TIME
    print("segmentation finished")
    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_vector'
    gdf = gpd.read_file(stream_vector)
    gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

    # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
    file_pattern = os.path.join(watershed_outputs_id, 'stream_vector.*')
    file_list = glob.glob(file_pattern)
    for file_path in file_list:
        os.remove(file_path)

    # WRITE TO A GEOPACKAGE
    layer_name = 'stream_points'
    gdf = gpd.read_file(stream_point)
    gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## GET NODES OF SEGMENTS AND JOIN POINTS' ATTRIBUTES
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # GET LINE NODES (START AND END) USING WHITEBOX TO MATCH POINT NODES
    stream_segments = os.path.join(watershed_outputs_id,"stream_segments.shp")
    stream_segment_nodes = os.path.join(watershed_outputs_id,"stream_segment_nodes.shp")
    wbt.extract_nodes(stream_segments, stream_segment_nodes)

    # CREATE JOIN IDENTIFIER - FIELD "INDEX" (PLUS 1) OF STREAM SEGMENT IS FIELD "PARENTID" OF STREAM
SEGMENT NODE.
    segment_attribute = ogr.Open(stream_segments, 1)
    segment_lyr = segment_attribute.GetLayer()
    segment_lyr.CreateField(ogr.FieldDefn("JoinID", ogr.OFTInteger))

    for feat in segment_lyr:
        # GET INDEX VALUE
        feat_v = feat.GetField("index")
        new_v = feat_v + 1
        # UPDATE NEW FIELD
        segment_lyr.SetFeature(feat)
        feat.SetField("JoinID", new_v)
        segment_lyr.SetFeature(feat)

    # JOIN STREAM POINTS DATA TO SEGMENTS' NODES
    stream_point_field_rename_df = geopandas.read_file(stream_point_field_rename_path)
    stream_segment_node_df = geopandas.read_file(stream_segment_nodes)

    stream_point_join2_segment_node = geopandas.sjoin(
        stream_segment_node_df,
        stream_point_field_rename_df,
        how="inner",
        op="intersects",
    )
    stream_segment_join2_point_path = os.path.join(watershed_outputs_id,"stream_point_join2_segment_node.shp"
    )
    stream_point_join2_segment_node.to_file(
        stream_segment_join2_point_path, driver="ESRI Shapefile"
    )
```

```
        # USE DISTANCE FROM OUTLET TO FIND THE COORDINATES OF START NODES AND END NODES OF SEGMENTS.
SAVE THE END COORDINATES IN THE SAME ROW AS THE START COORDINATES.
        stream_segment_join2_point_df = geopandas.read_file(stream_segment_join2_point_path)

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_point_join2_segment_node'
        stream_segment_join2_point_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_point_join2_segment_node.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segment_nodes'
        stream_segment_node_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_segment_nodes.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)

        stream_segment_join2_point_df["X"] = stream_segment_join2_point_df.geometry.x
        stream_segment_join2_point_df["Y"] = stream_segment_join2_point_df.geometry.y
        stream_segment_join2_point_df["StartXY"] = (
            stream_segment_join2_point_df["X"].map(str)
            + ","
            + stream_segment_join2_point_df["Y"].map(str)
        )

        # FIND MIN DISTANCE FROM OUTLET (I.E. LOWER NODE) AND UPDATE NEW COLUMN WITH ENDXY
COORDINATES.
        stream_segment_join2_point_df[
            "DistDown_min"
        ] = stream_segment_join2_point_df.groupby(["PARENT_ID"])["DistDown"].transform(min)

        # SUBSET DATA BY EXTRACTING START NODES
        stream_segment_join2_point_df_startnodes = stream_segment_join2_point_df[
            stream_segment_join2_point_df["DistDown_min"]
            != stream_segment_join2_point_df["DistDown"]
        ]
        stream_segment_join2_point_df_endnodes = stream_segment_join2_point_df[
            stream_segment_join2_point_df["DistDown_min"]
            == stream_segment_join2_point_df["DistDown"]
        ]
        stream_segment_join2_point_df_endnodes.loc[
            :, "EndXY"
        ] = stream_segment_join2_point_df_endnodes["StartXY"]

        stream_segment_join2_point_df_subset = pd.merge(
            stream_segment_join2_point_df_startnodes,
            stream_segment_join2_point_df_endnodes[["PARENT_ID", "EndXY"]],
            how="left",
            on="PARENT_ID",
        )

        # MATCH THE ELEV VALUE FROM THE STREAM POINTS FOR BOTH NODES OF STREAM SEGMENTS
        stream_segment_join2_point_df_subset[
            "EndElev"
        ] = stream_segment_join2_point_df_subset["EndXY"].map(
            stream_point_field_rename_df.set_index("pXY")["Elev"]
        )
        stream_segment_join2_point_df_subset[
            "StartElev"
        ] = stream_segment_join2_point_df_subset["StartXY"].map(
            stream_point_field_rename_df.set_index("pXY")["Elev"]
        )
        stream_segment_join2_point_df_subset_path = os.path.join(watershed_outputs_id,"stream_point_join2_segment_node_subset.shp")
        stream_segment_join2_point_df_subset.to_file(
            stream_segment_join2_point_df_subset_path, driver="ESRI Shapefile"
        )
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## JOIN STREAM SEGMENT NODES TO STREAM SEGMENT BASED ON PARENT ID
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        stream_segments_df = geopandas.read_file(stream_segments)
        stream_segment_join2_point_df_subset_df = geopandas.read_file(
            stream_segment_join2_point_df_subset_path
        )

        # MUST DROP GEOMETRY OF POINT TO ALLOW FILE TO BE SAVED IN THE FUTURE
        stream_segment_join2_point_df_subset_df = (
            stream_segment_join2_point_df_subset_df.drop("geometry", axis=1)
        )

        # STREAM_SEGMENTS_ATTRIBUTE_DF =
STREAM_SEGMENTS_DF.JOIN(STREAM_SEGMENT_JOIN2_POINT_DF_SUBSET.SET_INDEX('PARENT_ID'), ON='JOINID',
RSUFFIX = "_P")
        stream_segments_attribute_df = stream_segments_df.merge(
            stream_segment_join2_point_df_subset_df,
            left_on="JoinID",
            right_on="PARENT_ID",
            how="left",
        )

        # 1 MIN
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_point_join2_segment_node_subset'
        stream_segment_join2_point_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_point_join2_segment_node_subset.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments'
        gdf = gpd.read_file(stream_point)
        gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE DEM SLOPE, DRAINAGE AREA, DISCHARGE AND WIDTH (RURAL SCENARIO)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # CALCULATE LENGTH OF SEGMENTS
        stream_segments_attribute_df["Length"] = stream_segments_attribute_df.length

        # CALCULATE SLOPE USING DEM ELEVATION
        stream_segments_attribute_df["Slope"] = (
            (
                stream_segments_attribute_df["EndElev"]
                - stream_segments_attribute_df["StartElev"]
            )
            / (stream_segments_attribute_df["Length"])
        ) * -1

        # CALCULATE DRAINAGE AREA (DA = (FLOW ACCUMULATION/1000000)*100
        stream_segments_attribute_df["DA_km2"] = (
            stream_segments_attribute_df["FlowAccum"] / 1000000
        ) * cell_size

        # CALCULATE RURAL DISCHARGE USING Q=CA^X WHERE C AND X ARE COEFFICCIENTS AND A IS THE DRAINAGE
AREA
        stream_segments_attribute_df["Q_m3pers"] = rQ_a * (
            stream_segments_attribute_df["DA_km2"] ** rQ_b
        )

        # CALCULATE WIDTH (EMPIRICAL - DA WIDTH)
        stream_segments_attribute_df["RurWidth"] = rW_a * (
            stream_segments_attribute_df["DA_km2"] ** rW_b
```

```
        )
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## SMOOTH DEM SLOPE
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # SET INDEX FROM HIGH ELEVATION TO LOW, I.E. BY DECREASING DISTANCE DOWNSTREAM
        stream_segments_attribute_df = stream_segments_attribute_df.sort_values(
            by=["DistDown"], ascending=False
        )
        stream_segments_attribute_df["SID"] = np.arange(
            len(stream_segments_attribute_df)
        )
        stream_segments_attribute_df.set_index("SID", inplace=True)
        stream_segments_attribute_df["index"] = np.arange(
            len(stream_segments_attribute_df)
        )

        # ADD NEW FIELDS TO STORE RESULTS
        stream_segments_attribute_df["deltaDA"] = None
        stream_segments_attribute_df["SavgID"] = None
        stream_segments_attribute_df["SavgRID"] = None
        stream_segments_attribute_df["SlopeAVG"] = None
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CREATE SEGMENTS BLOCKS OVER WHICH TO SMOOTH SLOPE
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # CREATE LIST WITH ALL ELEVATION VALUES AND SORT IN DESCENDING ORDER. REMOVE ROWS WITHOUT
ATTRIBUTES
        stream_segments_attribute_df = stream_segments_attribute_df.dropna(
            axis=0, subset=["PARENT_ID"]
        )

        elev_set = set(stream_segments_attribute_df.apply(lambda row: (row["index"], row["EndElev"]), axis=1))

        elev_list_u = list(elev_set)
        elev_list = sorted(elev_list_u, key=lambda x: x[0])  # ACTIVATED

        # START FROM THE SEGMENT WITH HIGHEST ELEVATION, ASSIGN THE SAME SEGMENT ID WHILE TRACING
DOWNSTREAM OF THE REACH UNTIL THE DIFFERENCE RATIO IN DRAINAGE AREA IS >=0.1.
        seg_id = 0
        elev_list_count = elev_list

        while len(elev_list_count) != 0:
            elev_pair = elev_list_count[0]
            max_elev = elev_pair[1]
            max_id = elev_pair[0]
            seg_id = seg_id + 1
            del elev_list_count[0]

            # MAX_ELEV AND MAX_ID ONLY HAS ONE POSSIBLE MATCH, SKIP THE LOOP AND USE EXISTING INDEXES
            seg_r_id = 0
            start_endxy = stream_segments_attribute_df.loc[max_id, "EndXY"]
            start_da = stream_segments_attribute_df.loc[max_id, "DA_km2"]
            stream_segments_attribute_df.loc[max_id, "SavgID"] = seg_id
            stream_segments_attribute_df.loc[max_id, "SavgRID"] = seg_r_id

            if start_da is None or start_da == 0:
                start_da = 0.0

            # RATHER THAN LOOPING THROUGH MULTIPLE TIMES, QUERY THE DATAFRAME FOR THE MAIN CONDITION
            ssaIdxs = stream_segments_attribute_df.query(
                "@start_endxy == StartXY"
            ).index.tolist()
            # NOW LOOP THROUGH THE INITIAL MATCHES AND DO MATH
            while len(ssaIdxs) != 0:
                index = ssaIdxs[0]
                del ssaIdxs[0]
                r_da = stream_segments_attribute_df.loc[index, "DA_km2"]
                r_seg_id = stream_segments_attribute_df.loc[index, "SavgID"]
                if r_da == 0 or r_da == None:
                    delta_da = 0.0
                else:
                    diff_da = r_da - start_da
                    delta_da = (diff_da) / r_da
                if delta_da < thres_da and r_seg_id == None:
                    seg_r_id = seg_r_id + 1
                    stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
                    stream_segments_attribute_df.loc[index, "SavgRID"] = seg_r_id
                    stream_segments_attribute_df.loc[index, "SavgID"] = seg_id
                    stream_segments_attribute_df.loc[index, "deltaDA"] = delta_da
                    start_endxy = stream_segments_attribute_df.loc[index, "EndXY"]
                    # FIND MATCHES FOR THE NEW START_ENDXY
                    ssaIdxs = stream_segments_attribute_df.query(
                        "@start_endxy == StartXY"
                    ).index.tolist()
                    # DELETE PAIR IN LIST IF EXIST
                    r_elev = stream_segments_attribute_df.loc[index, "EndElev"]
                    r_id = index
                    pair = (r_id, r_elev)
                    # PRINT PAIR
                    if pair in elev_list_count:
                        elev_list_count.remove(pair)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE ROLLING MEAN OF SEGMENT USING 25 REACHES UPSTREAM AND 25 REACHES DOWNSTREAM IN A
SEGMENT
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        seg_id_l_unique = stream_segments_attribute_df["SavgID"].unique().tolist()

        # LOOP THROUGH TABLE AND COLLECT REACHIDS
        avg_window = {}

        for seg_id in seg_id_l_unique:
            # DONT LOOP AND CHECK CONDITIONS, GET INDEXES THAT MATCH CONDITION INSTEAD THEN LOOP
            ssaIdxs = stream_segments_attribute_df.query(
                "@seg_id == SavgID"
            ).index.tolist()
            for index in ssaIdxs:
                seg_r_id = stream_segments_attribute_df.loc[index, "SavgRID"]
                slope_raw = stream_segments_attribute_df.loc[index, "Slope"]
                avg_window[seg_r_id] = slope_raw

            avg_window_df = pd.Series(avg_window)
            roll_avg_df = avg_window_df.rolling(
                window=Savg_cell_window, min_periods=1, center=True
            ).mean()

            # LOOP THROUGH SET AND CALCULATE ROLLING MEAN
            for i, j in roll_avg_df.items():
                seg_r_id_avg = i
                avg_value = j
                ssaIdxs = stream_segments_attribute_df.query(
                    "(@seg_id == SavgID) & (@seg_r_id_avg == SavgRID)"
                ).index.tolist()
                for index in ssaIdxs:
                    stream_segments_attribute_df.loc[index, "SlopeAVG"] = abs(avg_value)

        # MUST FILL NULL SLOPE WITH 0
        stream_segments_attribute_df.fillna(value=0, inplace=True)

        stream_segments_smoothedslope_df_path = os.path.join(watershed_outputs_id,"stream_segments_SAVGslope.shp")
        stream_segments_attribute_df.to_file(
            stream_segments_smoothedslope_df_path, driver="ESRI Shapefile"
        )

        # WRITE TO A GEOPACKAGE
        layer_name = 'stream_segments_SAVGslope'
        stream_segments_attribute_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_segments_SAVGslope.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
```

```
            os.remove(file_path)
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE RURAL STREAM POWER
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
            # CALCULATE TOTAL STREAM POWER (SP) AND SPECIFIC STREAM POWER (SSP)
            stream_segments_attribute_df["RuralSP"] = pd.to_numeric(
               9810
               * stream_segments_attribute_df["SlopeAVG"]
               * stream_segments_attribute_df["Q_m3pers"],
               errors='coerce'
            )

            # RURAL SSP (EMPIRICAL FORMUL - DA SSP)
            stream_segments_attribute_df["RuralSSP"] = pd.to_numeric(
               stream_segments_attribute_df["RuralSP"] / stream_segments_attribute_df["RurWidth"],
               errors='coerce'
            )

            # FIND THE MAXIMUM RURAL WIDTH OLD METHOD
            max_val_pdmethod = stream_segments_attribute_df["RurWidth"].max()

            # # CALCULATE D84
            # # EQUATION 16 FROM FERGUSON (2005) ASSUMPTION: DI = DB
            # # CONSTANTS FOR D84 PREDICTION
            gamma = 9790 # N/M3
            kappa = 0.41 # CONSTANT FROM FERGUSON 2005 - WORKS FOR D IN MM, POWER IN W/M2
            R = 1.65 # SUBMERGED SPECIFIC GRAVITY
            rho = 1000 # DENSITY OF WATER
            theta_cb = 0.045 # SHEAR STRESS THRESHOLD
            m =2.80 # ROUGHNESS MULTIPLIER FOR D84 FROM LOPEZ AND BARRANGAN E.G. HEY 1979
            gee = 9.81 #ACCELERATION BY GRAVITY

            # C = LOG10(30*THETA_CB*R/(E*M*!S_MPERM!))"
            # EXPRESSIONDB =
"((KAPPA*!SPOWER_WPERM!/(2.30*RHO*LOG10(30*THETA_CB*R/(EXP*M*!S_MPERM!))))**(2/3))/(THETA_CB*R*GEE)"
            stream_segments_attribute_df["SlopeAVG"] = pd.to_numeric(stream_segments_attribute_df["SlopeAVG"], errors='coerce')
            stream_segments_attribute_df["eqlogC"] = np.log10(30 * theta_cb * R / (np.exp(1) * m *
np.abs(stream_segments_attribute_df["SlopeAVG"])))
            stream_segments_attribute_df["D84_mm"] = ((((kappa * stream_segments_attribute_df["RuralSSP"]) / (2.30 * rho *
stream_segments_attribute_df["eqlogC"])) ** (2.0 / 3)) / (theta_cb * R * gee)) * rho

            # SAVE FILE, MUST FILL NULL WITH 0 OR THE COLUMN WILL BE IMPORTED AS A STRING
            stream_segments_attribute_df.fillna(value=0, inplace=True)

            stream_segments_attribute_df_path = os.path.join(watershed_outputs_id,"stream_segments_attribute.shp")
            stream_segments_attribute_df.to_file(
               stream_segments_attribute_df_path, driver="ESRI Shapefile"
            )

            # SAVE A COPY OF SEGMENTS FOR MANIPULATION
            stream_segments_attribute_df_path_c = os.path.join(watershed_outputs_id,"stream_segments_attribute_edited.shp")
            stream_segments_attribute_df.to_file(
               stream_segments_attribute_df_path_c, driver="ESRI Shapefile"
            )

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_segments_attribute'
            stream_segments_attribute_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

            # WRITE TO A GEOPACKAGE
            layer_name = 'stream_segments_attribute_edited'
            stream_segments_attribute_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## END OF RURAL STREAM POWER CALCULATION
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
            gdf_strm = gpd.read_file(
               stream_segments_attribute_df_path_c
            )  # RENAME TO RURAL STREAM POWER FINAL
            copy_strm_net = os.path.join(watershed_outputs_id,"Final_Rural_Stream_Power_Output.shp")
            # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
            gdf_strm.to_file(copy_strm_net, driver="ESRI Shapefile")
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## WRITE SHAPEFILES TO A GEOPACKAGE TO SAVE SPACE
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'basin_hecpts.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'basin_selected.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'stream_points_initial.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'stream_vector_initial.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'dem_points.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'stream_vector.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'hecpt_buffers50m_join.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'dem_points_initial.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
            # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
            file_pattern = os.path.join(watershed_outputs_id, 'hecpt_buffers50m.*')
            file_list = glob.glob(file_pattern)
            for file_path in file_list:
               os.remove(file_path)
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT HECRAS DATA TO SEGMENTS
## ------------------------------------------------------------------------------------------------------------------------------------------------------------
            # TRANSFORM HECRASFILE FOR FLOOD EVENTS (.CSV) INTO COMPATIBLE FORMAT
            # READ FILE AS DATAFRAME (DF) BY SKIPPING 9 ROWS AND SPECIFY COLUMN HEADERS
            # IF CLIPPING POLITICAL BOUNDARY IS YES, AND HEC ANALYSIS IS YES, THEN LOOP THROUGH THE MULTIPLE
HECRAS DATA SHAPEFILES FILES FOR MULTIPLE WATERSHEDS
            # AND THE WATERSHEDS SAVED IN THE FOLDERS
            if hec_analysis == "Y":
               print("Reading csv\n")
               hec_df = pd.read_csv(hec_csv, header=[0, 1])

               print("Reached csv read point\n")
               # REMOVE UNNAMED CELLS IN COLUMN NAMES
               # FOR NEW VERSION OF PANDAS 0.21 AND OLD
               for i, columns in enumerate(hec_df.columns.levels):
                  columns_new = columns.tolist()
                  for j, row in enumerate(columns_new):
                     if "Unnamed: " in row:
                        columns_new[j] = ""
```

```
        if (
            pd.__version__ < "0.21.0"
        ):  # HTTPS://STACKOVERFLOW.COM/A/48186976/716469
            hec_df.columns.set_levels(columns_new, level=i, inplace=True)
        else:
            hec_df = hec_df.rename(
                columns=dict(zip(columns.tolist(), columns_new)), level=i
            )

    # COMBINE BOTH ROWS AS HEADERS
    col_raw = hec_df.columns
    col_raw.tolist()
    # NEW_COL = PD.INDEX([E[0] + E[1] FOR E IN COL_RAW.TOLIST()]) FOR GDB CLASS, LONG NAMES ARE ALLOWED
(I.E. "QCHANNEL_M3PERS")
    new_col = pd.Index(
        [e[0] for e in col_raw.tolist()]
    )  # ONLY NAME, NOT METRIC FOR SHAPEFILE
    hec_df.columns = new_col

    # TRANSFORM FIELD NAME BY REPLACE SPACE AND SYMBOLS WITH UNDERSCORE
    hec_df.columns = hec_df.columns.str.replace(" ", "")
    hec_df.columns = hec_df.columns.str.replace("(", "")
    hec_df.columns = hec_df.columns.str.replace(")", "")
    hec_df.columns = hec_df.columns.str.replace("/", "per")
    hec_df.columns = hec_df.columns.str.replace(".", "")
    hec_df.columns = hec_df.columns.str.replace("_", "")
    hec_df.columns = hec_df.columns.str.lower()

    # SELECT ONLY NEEDED COLUMNS
    hec_df = hec_df[
        [
            "riversta",
            "profile",
            "qtotal",
            "topwidth",
            "qchannel",
            "topwchnl",
            "minchel",
        ]
    ]

    # SHORTEN COLUMN NAME : ONLY FOR SHAPEFILE
    hec_df.rename(
        columns={
            "riversta": "RSta",
            "profile": "Profile",
            "qtotal": "QTotal",
            "topwidth": "TopW",
            "qchannel": "QCh",
            "topwchnl": "TopWCh",
            "minchel": "ChEl",
        },
        inplace=True,
    )

    # TRANSFORM COLUMNS WITH PROFILE - KEEP ONLY NUMBER
    # HEC_DF["PROFILE"] = HEC_DF["PROFILE"].STR.REPLACE(R'\D', R'') # NOT IN USE
    hec_df["Profile"].replace(
        regex=True, inplace=True, to_replace=r"\D", value=r""
    )

    # REPLACE ALL SPACE IN HEC_DF
    hec_df = hec_df.replace(" ", "", regex=True)

    # DROP ROWS WITH NONE
    hec_df = hec_df.dropna(axis=0, how="all")

    # DROP ROWS WITH NONE UNDER PROFILE
    hec_df = hec_df.dropna(subset=["Profile"], axis=0, how="all")

    # REPLACE ALL SPACE IN HEC_DF
    hec_df = hec_df.replace("_", "", regex=True)

    # SAVE THE LIST OF CULVERT IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
    hec_profile_list = list(hec_df["Profile"])
    hec_profile_list_drop = list(dict.fromkeys(hec_profile_list))

    # PIVOT TABLE BY MAKING THE PROFILE (I.E. STORM EVENT) DATA INTO COLUMNS
    # HEC_DF_PIV = HEC_DF.PIVOT(INDEX='RSTA', COLUMNS = 'PROFILE')
    hec_df_piv = pd.pivot_table(
        hec_df, index="RSta", columns="Profile"
    )  # OLD CODE

    # GROUP STACKED COLUMNS
    hec_df_piv.columns = hec_df_piv.columns.map("".join)

    # SAVE TRANSFORMED HEC RAS FILE
    hec_csv_f = str(watershed_outputs_id) + "/hecras_csv_formatted.csv"
    hec_df_piv.to_csv(hec_csv_f)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT RIVER STATION POINTS TO TRANSFORMED HECRAS DATA
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # READ HEC POINTS SHAPEFILE
    hec_point_df = geopandas.read_file(hec_point)

    # SELECT ONLY NEEDED ROWS
    hec_point_df = hec_point_df[["RIVER_STA", "geometry"]]

    # READ HEC FORMATTED DATA
    hec_csv_df = pd.read_csv(hec_csv_f)

    # JOIN TRANSFORMED HECRAS DATA TO HEC POINTS SHAPEFILE
    hec_pts_joined_data = pd.merge(
        hec_point_df,
        hec_csv_df,
        how="inner",
        left_on=["RIVER_STA"],
        right_on=["RSta"],
    )

    # SAVING AS SHAPEFILE
    hec_pts_joined_data_path = os.path.join(watershed_outputs_id,"hec_points_joined2data.shp")
    hec_pts_joined_data.to_file(
        hec_pts_joined_data_path, driver="ESRI Shapefile"
    )

    # WRITE TO A GEOPACKAGE
    layer_name = 'hec_points_joined2data'
    hec_pts_joined_data.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT RIVER STATION POINTS WITH HECRAS DATA TO STREAM REACH
## ------------------------------------------------------------------------------------------------------------------------------------------------------
    # OPEN LINE AND POINTS DATA
    gdf_segments = geopandas.read_file(stream_segments_attribute_df_path)
    gdf_hecpoint = geopandas.read_file(hec_pts_joined_data_path)

    # SAVE A COPY OF SEGMENTS FOR MANIPULATION
    gdf_segments_c = geopandas.read_file(
        stream_segments_attribute_df_path_c
    )

    # UNION ALL GEOMETRIES
    shply_line = gdf_segments.geometry.unary_union
    shply_point = gdf_hecpoint.geometry.unary_union

    # SELECT POINTS WITHIN 45 M OF RIVER BY CREATING POLYGON
    buffer_45 = shply_line.buffer(45)
    buff_df = geopandas.GeoDataFrame(geometry=[buffer_45])
```

```
# JOIN POINTS TO BUFFER
point_in_buffer = geopandas.sjoin(gdf_hecpoint, buff_df, how="left")

# DROP POINTS OUTSIDE OF BUFFER
point_45 = point_in_buffer.dropna(axis=0, subset=["index_right"])

# SAVE POINTS
point_45_path = os.path.join(watershed_outputs_id,"hec_points_within45m.shp")
point_45.to_file(point_45_path, driver="ESRI Shapefile")

gdf_hecpoint_45m = geopandas.read_file(point_45_path)

# WRITE TO A GEOPACKAGE
layer_name = 'hec_points_within45m'
gdf_hecpoint_45m.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')


# GET CENTROID OF SEGMENTS AND SAVE AS NEW FILE
gdf_segments_c["geometry"] = gdf_segments_c["geometry"].centroid
seg_centroid_path = os.path.join(watershed_outputs_id,"segments_centroid.shp")
gdf_segments_c.to_file(seg_centroid_path, driver="ESRI Shapefile")

# OPEN CENTROID FILE
gdf_centroid = geopandas.read_file(seg_centroid_path)

# WRITE TO A GEOPACKAGE
layer_name = 'segments_centroid'
gdf_centroid.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

# FUNCTION TO FIND NEAREST SEGMENT'S CENTROID TO HECDATA POINT
def nearest(
    row,
    geom_union,
    df1,
    df2,
    geom1_col="geometry",
    geom2_col="geometry",
    src_column=None,
):
    """Find the nearest point and return the corresponding
    value from specified column."""
    # FIND THE GEOMETRY THAT IS CLOSEST
    nearest = (
        df2[geom2_col] == nearest_points(row[geom1_col], geom_union)[1]
    )
    # GET THE CORRESPONDING VALUE FROM DF2 (MATCHING IS BASED ON THE GEOMETRY)
    value = df2[nearest][src_column].values[
        0
    ]  # GET VALUES IS DEPREACIATED THE NEW ONE IS CALLED VALUES
    return value

# CREATE MULTIPOINT: CENTROID OF SEGMENTS
unary_union = gdf_centroid.unary_union

# FIND ID OF LINE CLOSEST TO HEC POINT
gdf_hecpoint_45m["NearSegID"] = gdf_hecpoint_45m.apply(
    nearest,
    geom_union=unary_union,
    df1=gdf_hecpoint_45m,
    df2=gdf_centroid,
    src_column="JoinID",
    axis=1,
)

# ONLY 1 HEC POINT CAN BE ATTACHED TO A SEGMENT. CHOOSE ONLY 1 POINT PER SEGMENT.
gdf_hecpoint_45m_sub = gdf_hecpoint_45m.drop_duplicates(
    subset="NearSegID", keep="first", inplace=False
)

# JOIN HEC POINTS TO STREAM SEGMENTS, MUST DROP FIRST GEOMETRY COLUMN
        gdf_hecpoint_45m_sub = gdf_hecpoint_45m_sub.drop(columns="geometry")
        hec45m_2_seg = pd.merge(
            gdf_segments,
            gdf_hecpoint_45m_sub,
            how="left",
            left_on=["JoinID"],
            right_on=["NearSegID"],
        )

        hec45m_2_seg_path = os.path.join(watershed_outputs_id,"hec45m_join2_segments.shp")
        hec45m_2_seg.to_file(hec45m_2_seg_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'segments_centroid'
        gdf_centroid.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        print("Completed saving", hec45m_2_seg_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------------
## INTERPOLATE HECRAS DATA FOR SEGMENTS WITHOUT DATA
## ---------------------------------------------------------------------------------------------------------------------------------------------------
        # IMPORT SHAPEFILE CONTAINING STORM EVENT DATA AS DATAFRAME
        strm_hec_path = os.path.join(watershed_outputs_id,"hec45m_join2_segments.shp")
        strm_hec_df = geopandas.read_file(strm_hec_path)

        # WRITE TO A GEOPACKAGE
        layer_name = 'hec45m_join2_segments'
        strm_hec_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

        print("Read the file:", strm_hec_df)

        # RESET UNIQUE INDEX OF DATAFRAME USING DISTANCE DOWNSTREAM IN DESCENDING ORDER SO THAT THE
REACHES ARE ORDERED FROM THEIR UPSTREAM LOCATION TO DOWNSTREAM LOCATION
        strm_hec_df["iRID"] = np.arange(len(strm_hec_df))
        strm_hec_df = strm_hec_df.sort_values(by=["DistDown"], ascending=False)
        strm_hec_df.set_index("iRID", inplace=True)

        # REPLACE NAN WITH NONE
        strm_hec_df = strm_hec_df.replace({np.nan: None})

        #STRM_HEC_DF = STRM_HEC_DF.WHERE((PD.NOTNULL(STRM_HEC_DF)), NONE) THIS IS NOT WORKING IN NEW
PYTHON VERSOIN

        # FIELDS
        river_sta_id = "RSta" # RIVER_STA
        unique_id = "JoinID" # REACH ID
        dist_value = "Length" # STREAM LENGTH

        #ONLY REQUIRED FIELDS FOR STREAM POWER WILL BE INTERPOLATED
        req_fields = ['QCh','TopWCh', 'ChEl']

        raw_value_split = []

        for fld in req_fields:
            for p in hec_profile_list_drop:
                val = str(fld) + str(p)
                raw_value_split.append(val)

        # LOOP THROUGH EACH FIELD FOR INTERPOLATION AND CREATE NEW FIELD NAMES
        calc_name_list = []
        for r in raw_value_split:
            raw_value = str(r)
            calc_name = str("i") + str(r)
            calc_name_list.append(calc_name)
            # ADD FIELD TO IDENTIFY REACHES THAT WILL BE INTERPOLATED BETWEEN TWO SELECTED RIVER
STATIONS
            strm_hec_df["iID"] = None
            strm_hec_df["iLENGTH"] = None
            # NEW FIELD FOR CALCULATED VALUE
            strm_hec_df[calc_name] = None
            # CREATE LIST WITH ALL RIVER STATION IDS WHERE DATA EXISTS.
```

```
            sta_list = []
            sta_list_c = []
            for index, row in strm_hec_df.iterrows():
                r_sta = row[river_sta_id]
                r_id = row[unique_id]
                raw_v = row[raw_value]
                if raw_v is not None:
                    pair = (r_id, r_sta)
                    sta_list.append(pair)
                    sta_list_c.append(pair)
            #PRINT(STA_LIST)
            # LOOP THROUGH TABLE, GET FIRST STATION IN LIST OF STATION WITH EXISTING DATA, ASSIGN A
CUMULATIVE LENGTH (INTP_LENGTH) OF 0 TO THE FIRST STATION, SAVE THE VALUE FOR THE FIELD
            # TO BE INTERPOLATED IN Y_LIST AND SAVE THE CUMULATIVE DISTANCE IN THE X_LIST.
            y_list = []
            x_list = []
            x_intp_list = []

            intp_id = 0

            while len(sta_list_c) != 0:
                x_list=[]
                y_list=[]
                x_intp_list = []
                # BEGIN LOOP TO GET FIRST STATION
                first_sta = sta_list_c[0][1]
                first_rid = sta_list_c[0][0]
                #PRINT(FIRST_STA, FIRST_RID)
                intp_id = intp_id + 1
                del sta_list_c[0]
                #PRINT("PROCESSING STATION " + STR(FIRST_STA))
                for index, row in strm_hec_df.iterrows():
                    r_sta = row[river_sta_id]
                    r_id = row[unique_id]
                    if first_sta == r_sta and first_rid == r_id:
                        #PRINT("R_STA " + STR(R_STA))
                        #PRINT("R_ID " + STR(R_ID))
                        dist_sum = 0
                        #print(dist_sum, "In if loop")
                        y_start = 0
                        start_endxy = row["EndXY"] # GET END OF REACH
                        #PRINT("STARTSTA: " + STR())
                        first_rawv = row[raw_value] # GET THE VALUE TO AVERAGE
                        x_list.append(y_start)    # ADD FIRST X=0 TO X LIST
                        y_list.append(first_rawv) # ADD FIRST ELEV TO Y LIST
                        strm_hec_df.loc[index,calc_name] = first_rawv
                        strm_hec_df.loc[index,"iID"] = intp_id
                        strm_hec_df.loc[index,"iLENGTH"] = dist_sum
                        break

                dist_sum
                #print(dist_sum)
                # TRACE DOWNSTREAM USING END XY COORDINATES OF FIRST REACH TO FIND START XY COORDINATES
OF SUBSEQUENT REACH
                # IF BOTH RAW VALUE FIELD (FIELD TO BE INTERPOLATED) AND INTERPOLATED FIELD (CALC_NAME)ARE
NONE, THEN UPDATE
                # THE REACH'S CUMULENGTH LENGTH (INTP_LENGTH), IDENTIFIER (INTP_ID) AND SAVE THE REACH ID. IF
RAW VALUE EXISTS, BREAK LOOP.
                # E.G. LET'S INTERPOLATE ELEVATION VALUES
                # REACH ID = [1,2,3,4]  STATIONS 2 AND 3 ARE LOCATED BETWEEN 1 AND 4. 1 AND 4 HAVE HECRAS DATA.
                # INTP_LENGTH = [0,10,20,30]
                # INTP_ID = [1,1,1,1]
                # RAW VALUE = [120, NA, NA, 30]
                # X_LIST = [0,10,20,30] # SAME AS CUMULATIVE LENGTH
                # Y_LIST = [120, NA, NA, 30]
                # INTERPOLATED RAW VALUE = [120, 90, 60,30]
                for index, row in strm_hec_df.iterrows():
                    r_startxy = row["StartXY"]
                    if r_startxy == start_endxy:
                        distv = row[dist_value] # GET THE DISTANCE
                        rawv = row[raw_value] # GET THE ELEV
                        intp_v = row[calc_name]
                        #PRINT DISTV
                        #PRINT RAWV
                        #PRINT INTP_V
                        if rawv == None or np.isnan(rawv) and intp_v == None:
                            dist_sum = dist_sum + distv
                            #PRINT("SUM DISTANCE " + STR(DIST_SUM))
                            x_intp_list.append(dist_sum)
                            strm_hec_df.loc[index,"iLENGTH"] = dist_sum
                            strm_hec_df.loc[index,"iID"] = intp_id
                            start_endxy = row["EndXY"]

                            #DELETE PAIR IN LIST IF EXIST
                            r_sta = row[river_sta_id]
                            r_id = row[unique_id]
                            pair = (r_id,r_sta)
                            #PRINT(PAIR)
                            if pair in sta_list_c:
                                sta_list_c.remove(pair)
                        elif rawv is not None:
                            dist_sum = dist_sum + distv
                            y_list.append(rawv)
                            x_list.append(dist_sum)
                            break
                # USE X_LIST (CUMULATIVE LENGTH) AND Y_LIST (RAW VALUES)
                for index, row in strm_hec_df.iterrows():
                    intp_id_tbl = row["iID"]
                    y_sum_tbl = row["iLENGTH"]
                    rawv_tbl = row[calc_name]
                    if intp_id == intp_id_tbl and rawv_tbl == None:
                        intp_rawv = np.interp(y_sum_tbl, x_list, y_list)
                        if len(x_list) == 1:
                            strm_hec_df.loc[index,calc_name] = None   #### MUST CHECK THIS. IF DIFFERENCE IS TOO SMALL, MIN
ELEV MAY STAY THE SAME. (SOLVED)
                        else:
                            strm_hec_df.loc[index,calc_name] = intp_rawv

            # SAVE FILE, REPLACE NONE WITH 0
            strm_hec_df.fillna(value=np.nan, inplace=True)
            #STRM_HEC_DF.FILLNA(0, INPLACE=TRUE)
            strm_hec_df.replace(to_replace= np.nan, value=0, inplace=True)


            strm_hec_gdf_i_path = os.path.join(watershed_outputs_id, "segment_dem_hec_connected_intp.shp")
            strm_hec_df.to_file(strm_hec_gdf_i_path, driver="ESRI Shapefile")

            # WRITE TO A GEOPACKAGE
            layer_name = 'segment_dem_hec_connected_intp'
            strm_hec_df.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')

            print("Saved file", strm_hec_gdf_i_path)
## ---------------------------------------------------------------------------------------------------------------------------------------
## CALCULATE STREAM POWER (URBAN SCENARIO)
## ---------------------------------------------------------------------------------------------------------------------------------------
            # CREATE NEW FIELDS TO STORE TOTAL AND SPECIFIC STREAM POWER
            strm_hec_intp_gdf = geopandas.read_file(strm_hec_gdf_i_path)
            # CHECK IF ITOPWCH2 (2 YR RETURN PERIOD) IS LESS THAN RURAL WIDTH, IF LESS THAN EMPIRICAL USE THE
RURAL WIDTH AT GIVEN POINT
            # LOOP THROUGH THE ROWS AND UPDATE 'MODW' BASED ON THE CURRENT WIDTH CALC
            # DO WIDTHS FOR EVERY STORM

            for p in hec_profile_list_drop:
                check_field = "iTopWCh" + str(p)
                new_emp_width = "ModW" + str(p)
                #NEW_ASH_WIDTH = "MODWASH" + STR(P)

                strm_hec_intp_gdf[new_emp_width] = None

                # UPDATE NEW_EMP_WIDTH COLUMN BASED ON CONDITIONS
```

```
    strm_hec_intp_gdf[check_field] = pd.to_numeric(strm_hec_intp_gdf[check_field], errors='coerce').astype(float)
    mask = (strm_hec_intp_gdf[check_field] >= 0) & (strm_hec_intp_gdf[check_field] >= strm_hec_intp_gdf["RurWidth"])
    strm_hec_intp_gdf.loc[mask, new_emp_width] = strm_hec_intp_gdf.loc[mask, check_field]
    strm_hec_intp_gdf.loc[~mask, new_emp_width] = strm_hec_intp_gdf.loc[~mask, "RurWidth"]

    # CONVERT NEW_EMP_WIDTH COLUMN TO FLOAT
    strm_hec_intp_gdf[new_emp_width] = pd.to_numeric(strm_hec_intp_gdf[new_emp_width], errors="coerce").astype(float)

    # CONVERT COLUMNS TO NUMERIC DATA TYPES
    strm_hec_intp_gdf[check_field] = pd.to_numeric(strm_hec_intp_gdf[check_field], errors="coerce")

# SAVE THE UPDATED SHAPEFILE
strm_hec_intp_gdf.to_file(strm_hec_gdf_i_path)

for p in hec_profile_list_drop:

    sp_field = "SP" + str(p)
    disch_field = "iQCh" + str(p)
    strm_hec_intp_gdf["SlopeAVG"] = pd.to_numeric(strm_hec_intp_gdf["SlopeAVG"], errors="coerce")
    strm_hec_intp_gdf[disch_field] = pd.to_numeric(strm_hec_intp_gdf[disch_field], errors="coerce")
    strm_hec_intp_gdf[sp_field] = None
    strm_hec_intp_gdf[sp_field] = (
        9810
        * strm_hec_intp_gdf["SlopeAVG"]
        * strm_hec_intp_gdf[disch_field]
    )

# ORIGINAL RROIT SCRIPT
# CALCULATE SPECIFIC STREAM POWER (SSP) = SP/WIDTH
for p in hec_profile_list_drop:
    ssp_field = "SSP" + str(p)
    w_field = "iTopWCh" + str(p)
    sp_field = "SP" + str(p)
    strm_hec_intp_gdf[ssp_field] = None
    strm_hec_intp_gdf[ssp_field] = (
        strm_hec_intp_gdf[sp_field] / strm_hec_intp_gdf[w_field]
    )

# CALCULATE SSPR = URBAN SSP/ RURAL SSP: IN THIS CASE SSP2 REPRESENTS RURAL CONDITION
if "SSP2" in strm_hec_intp_gdf.columns:
    for p in hec_profile_list_drop:
        rural_ssp_field = "SSP2"
        urban_ssp_field = "SSP" + str(p)
        sspr_field = "SSPR" + str(p)
        strm_hec_intp_gdf[sspr_field] = None
        strm_hec_intp_gdf[sspr_field] = (
            strm_hec_intp_gdf[urban_ssp_field]
            / strm_hec_intp_gdf[rural_ssp_field]
        )

# CALCULATE SSPR = SSP2/RURALSSP AND COMPARE 2-YEAR URBAN TO RURAL
if "SSP2" in strm_hec_intp_gdf.columns and "RuralSSP" in strm_hec_intp_gdf.columns:
    sspr_q2_field = "SSPR"
    strm_hec_intp_gdf[sspr_q2_field] = None
    strm_hec_intp_gdf["SSP2"] = strm_hec_intp_gdf["SSP2"].astype(float)
    strm_hec_intp_gdf["RuralSSP"] = strm_hec_intp_gdf["RuralSSP"].astype(float)
    strm_hec_intp_gdf.loc[:, sspr_q2_field] = strm_hec_intp_gdf["SSP2"] / strm_hec_intp_gdf["RuralSSP"]

# THIS IS WITH THE TEST WHERE IF ITOPWCH IS LESS THAN EMPIRICAL THEN CREATE A NEW COLUMN
# CALCULATE SPECIFIC STREAM POWER (SSP) = SP/WIDTH;
for p in hec_profile_list_drop:
    ssp_field = "ModSSP" + str(p)
    w_field = "ModW" + str(p)
    sp_field = "SP" + str(p)
    strm_hec_intp_gdf[ssp_field] = None

    mask = strm_hec_intp_gdf[w_field] > 0
    strm_hec_intp_gdf.loc[mask, ssp_field] = (
        strm_hec_intp_gdf.loc[mask, sp_field]
        / strm_hec_intp_gdf.loc[mask, w_field]
    )
    strm_hec_intp_gdf.loc[~mask, ssp_field] = 0

    strm_hec_intp_gdf[ssp_field] = pd.to_numeric(
        strm_hec_intp_gdf[ssp_field], errors="coerce"
    ).astype(float)

# CALCULATE SSPR = URBAN SSP/ RURAL SSP: IN THIS CASE SSP2 REPRESENTS RURAL CONDITION
# COMPARE EACH STM EVENT WITH RURAL SSP, 2 YR RURAL WITH 2 YR URBAN
# COMPARE THE SSP FOR ALL EVENTS TO THE 2 YR AND COMPARE 2YR TO THE RURAL
if "ModSSP2" in strm_hec_intp_gdf.columns:
    for p in hec_profile_list_drop:
        rural_ssp_field = "ModSSP2"  # MEANS COMPARING TO THE NORMAL CONDITION (I.E MOST FREQUENT 2 YR)
        urban_ssp_field = "ModSSP" + str(p)
        sspr_field = "ModSSPR" + str(p)
        strm_hec_intp_gdf[sspr_field] = None

        mask2 = strm_hec_intp_gdf[rural_ssp_field] > 0
        strm_hec_intp_gdf.loc[mask2, sspr_field] = (
            strm_hec_intp_gdf.loc[mask2, urban_ssp_field]
            / strm_hec_intp_gdf.loc[mask2, rural_ssp_field]
        )
        strm_hec_intp_gdf.loc[~mask2, sspr_field] = 0

        strm_hec_intp_gdf[sspr_field] = pd.to_numeric(
            strm_hec_intp_gdf[sspr_field], errors="coerce"
        ).astype(float)

    # CALCULATE SSPR = SSP2/SSPR,  AND COMPARE 2YR URBAN TO THE RURAL
    if "ModSSP2" in strm_hec_intp_gdf.columns:
        sspr_q2_field = "ModSSPR"
        strm_hec_intp_gdf[sspr_q2_field] = None
        strm_hec_intp_gdf[sspr_q2_field] = (
            strm_hec_intp_gdf["ModSSP2"] / strm_hec_intp_gdf["RuralSSP"]
        )

        strm_hec_intp_gdf[sspr_q2_field] = pd.to_numeric(
            strm_hec_intp_gdf[sspr_q2_field], errors="coerce"
        ).astype(float)

        # SET PRJ AND SAVE FILE
        strm_hec_intp_gdf.crs = output_crs
        strm_hec_intp_gdf_calc_path = os.path.join(watershed_outputs_id,"segment_dem_hec_streampow_prj.shp")
        strm_hec_intp_gdf.to_file(
            strm_hec_intp_gdf_calc_path, driver="ESRI Shapefile"
        )

        # WRITE TO A GEOPACKAGE
        layer_name = 'segment_dem_hec_streampow_prj'
        strm_hec_intp_gdf.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------------------
## RENAME TO HECRAS STREAM POWER FINAL
## END OF HECRAS CROSS SECTION ANALYSIS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------------
        gdf_hecfinal = gpd.read_file(
            strm_hec_intp_gdf_calc_path
        )  # READ THE ORIGINAL SHAPEFILE
        copy_hecfinal = (
            str(watershed_outputs_id) + "\Final_HECRAS_Stream_Power_Output.shp"
        )  # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
        gdf_hecfinal.to_file(copy_hecfinal, driver="ESRI Shapefile")
## ---------------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT HECRAS BRIDGE DATA TO SEGMENTS
## ---------------------------------------------------------------------------------------------------------------------------------------------------------------
        if bridge_analysis == "Y":
            # TRANSFORM HEC CSV INTO COMPATIBLE FORMAT
            # READ FILE AS DATAFRAME BY SKIPPING 9 ROWS AND SPECIFY COLUMN HEADERS
            df_bridge = pd.read_csv(bridge_hec_csv, header=[0, 1])

            # REMOVE UNNAMED CELLS IN COLUMN NAMES
```

```
# FOR NEW VERSION OF PANDAS 0.21 AND OLD
for i, columns in enumerate(df_bridge.columns.levels):
    columns_new = columns.tolist()
    for j, row in enumerate(columns_new):
        if "Unnamed: " in row:
            columns_new[j] = ""
    if (
        pd.__version__ < "0.21.0"
    ):  # HTTPS://STACKOVERFLOW.COM/A/48186976/716469
        df_bridge.columns.set_levels(columns_new, level=i, inplace=True)
    else:
        df_bridge = df_bridge.rename(
            columns=dict(zip(columns.tolist(), columns_new)), level=i
        )

# COMBINE BOTH ROWS AS HEADERS
col_raw = df_bridge.columns
col_raw.tolist()
# NEW_COL = PD.INDEX([E[0] + E[1] FOR E IN COL_RAW.TOLIST()]) FOR GDB CLASS, LONG NAMES ARE ALLOWED
(I.E. "QCHANNEL_M3PERS")
new_col = pd.Index(
    [e[0] for e in col_raw.tolist()]
)  # ONLY NAME, NOT METRIC FOR SHAPEFILE
df_bridge.columns = new_col

# TRANSFORM FIELD NAME BY REPLACE SPACE AND SYMBOLS WITH UNDERSCORE
df_bridge.columns = df_bridge.columns.str.replace(" ", "")
df_bridge.columns = df_bridge.columns.str.replace("(", "")
df_bridge.columns = df_bridge.columns.str.replace(")", "")
df_bridge.columns = df_bridge.columns.str.replace("/", "per")
df_bridge.columns = df_bridge.columns.str.replace(".", "")
df_bridge.columns = df_bridge.columns.str.replace("_", "")
df_bridge.columns = df_bridge.columns.str.lower()

# SELECT COLUMNS NEEDED
df_bridge = df_bridge[
    [
        "reach",
        "riversta",
        "profile",
        "qtotal",
        "qchannelupstream",
        "qbridge",
        "deckwidth",
        "topwchnlupstream",
        "scourtype",
    ]
]

# SHORTEN COLUMN NAME : ONLY FOR SHAPEFILE
df_bridge.rename(
    columns={
        "reach": "Reach",
        "riversta": "RSta",
        "profile": "Profile",
        "qtotal": "QTot",
        "qchannelupstream": "QChUp",
        "qbridge": "QBr",
        "deckwidth": "DeckW",
        "topwchnlupstream": "TopWChUp",
        "scourtype": "scour"
    },
    inplace=True,
)

# TRANSFORM COLUMNS WITH PROFILE - KEEP ONLY NUMBER
df_bridge["Profile"].replace(
    regex=True, inplace=True, to_replace=r"\D", value=r""
)

# DELETE SPACE IN DATAFRAME WITH UNDERSCORE
df_bc = df_bridge.replace(" ", "", regex=True)

# DROP ROWS WITH NONE
df_bc = df_bc.dropna(axis=0, how="all")

# DROP ROWS WITH NONE UNDER PROFILE
df_bc = df_bc.dropna(subset=["Profile"], axis=0, how="all")

# SAVE THE LIST OF BRIDGE IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
bridge_profile_list = list(df_bc["Profile"])
bridge_profile_list_drop = list(dict.fromkeys(bridge_profile_list))

# PIVOT TABLE TO TRANSFORM PROFILE FROM ROW WISE TO COLUMN WISE
df_piv_b = pd.pivot_table(df_bc, index="RSta", columns="Profile")

# GROUP STACKED COLUMNS
df_piv_b.columns = df_piv_b.columns.map("".join)

# SAVE TRANSFORMED DATA WITH NEW FIELDS
hec_csv_bf_path = os.path.join(watershed_outputs_id,"hecras_bridge_formatted.csv")
df_piv_b.to_csv(hec_csv_bf_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT TRANSFORMED BRIDGE DATA TO GDB TABLE. TO CALCULATE BRIDGE SCOUR AND BRIDGE OVERTOPPING,
THE DATA FOR THE UPSTREAM RIVER STATION AND AT THE BRIDGE ARE REQUIRED.
## ---------------------------------------------------------------------------------------------------------------------------------------------
# CONVERT CSV TO GDB TABLE
gdf_bridge = pd.read_csv(hec_csv_bf_path)

# ADD NEW COLUMNS
bridge_scour_label_list = []
bridge_overtop_label_list = []

for y in bridge_profile_list_drop:
    new_lab = "SBr" + str(y)
    bridge_scour_label_list.append(new_lab)
    print(new_lab)
    # CREATE EMPTY FIELD
    gdf_bridge[new_lab] = ""
    # APPLY EQUATION
    q_br_lab = "QBr" + str(y)
    print(q_br_lab)
    qchUp_br_lab = "QChUp" + str(y)
    print(qchUp_br_lab)
    topWChUp_br_lab = "TopWChUp" + str(y)
    print(topWChUp_br_lab)
    deck_lab = "DeckW" + str(y)
    print(deck_lab)

    # CHECK THE SPECIFIC COLUMN (E.G., "SCOURTYPE_COLUMN") FOR THE VALUE 1 BEFORE PERFORMING THE
CALCULATION
    scourtype_column = "scour" + str(y) # REPLACE THIS WITH THE ACTUAL COLUMN NAME FOR SCOURTYPE

    # APPLY THE CALCULATION ONLY WHEN SCOURTYPE IS EQUAL TO 1
    scourtype_1_mask = gdf_bridge[scourtype_column] == 1
    gdf_bridge.loc[scourtype_1_mask, new_lab] = (
        (gdf_bridge.loc[scourtype_1_mask, q_br_lab] / gdf_bridge.loc[scourtype_1_mask, qchUp_br_lab]) ** (6 / 7)
    ) * ((gdf_bridge.loc[scourtype_1_mask, topWChUp_br_lab] / gdf_bridge.loc[scourtype_1_mask, deck_lab]) ** 0.59)


for y in bridge_profile_list_drop:
    new_lab = "OBr" + str(y)
    bridge_overtop_label_list.append(new_lab)
    print(new_lab)
    gdf_bridge[new_lab] = ""
    # APPLY EQUATION
    qtot_lab = "QTot" + str(y)
    qbridge_lab = "QBr" + str(y)
    # CALCULATE
```

```
            gdf_bridge[new_lab] = gdf_bridge[qtot_lab] / gdf_bridge[qbridge_lab]

        # SUBSET ONLY CALCULATED VALUES, ADD RIVER STA AND SAVE
        merged_bridge_ratios_lab = (
            bridge_scour_label_list + bridge_overtop_label_list
        )
        merged_bridge_ratios_lab.insert(0, "RSta")
        merged_bridge_ratios_df = gdf_bridge[merged_bridge_ratios_lab]

        # CREATE A COPY OF TABLE
        gdf_bridge_path = os.path.join(watershed_outputs_id,"bridge_scour_overtop_ratios_only.csv")
        merged_bridge_ratios_df.to_csv(gdf_bridge_path)
## ---------------------------------------------------------------------------------------------------------------------
## CONNECT BRIDGE CALCULATIONS TO A COPY OF RIVER STATIONS POINTS FILE
## ---------------------------------------------------------------------------------------------------------------------
        # MAKE A COPY OF POINTS FILE
        bridge_point_df = geopandas.read_file(bridge_point)
        bridge_point_df = bridge_point_df[["RIVER_STA", "geometry"]]

        # JOIN BRIDGE CALCULATIONS TO HEC POINTS SHAPEFILE
        bridge_pts_joined = pd.merge(
            bridge_point_df,
            merged_bridge_ratios_df,
            how="inner",
            left_on=["RIVER_STA"],
            right_on=["RSta"],
        )
## ---------------------------------------------------------------------------------------------------------------------
## CONNECT BRIDGE POINTS TO STREAM REACH
## ---------------------------------------------------------------------------------------------------------------------
        # OPEN A COPY OF SEGMENTS CENTROID FILE
        gdf_centroid_c = geopandas.read_file(seg_centroid_path)
        gdf_segments_c = geopandas.read_file(
            stream_segments_attribute_df_path_c
        )
        gdf_hec45m_2_seg = geopandas.read_file(copy_hecfinal)

        # OPEN CENTROID FILE
        gdf_centroid = geopandas.read_file(seg_centroid_path)

        # FUNCTION TO FIND NEAREST SEGMENT'S CENTROID TO HECDATA POINT
        def nearest(
            row,
            geom_union,
            df1,
            df2,
            geom1_col="geometry",
            geom2_col="geometry",
            src_column=None,
        ):
            """Find the nearest point and return the corresponding
            value from specified column."""
            # FIND THE GEOMETRY THAT IS CLOSEST
            nearest = (
                df2[geom2_col] == nearest_points(row[geom1_col], geom_union)[1]
            )
            # GET THE CORRESPONDING VALUE FROM DF2 (MATCHING IS BASED ON THE GEOMETRY)
            value = df2.loc[nearest, src_column].iloc[0]
            return value

        # CREATE MULTIPOINT: CENTROID OF SEGMENTS
        unary_union = gdf_centroid_c.unary_union

        # FIND ID OF LINE CLOSEST TO HEC POINT
        bridge_pts_joined["NearSegID"] = bridge_pts_joined.apply(
            nearest,
            geom_union=unary_union,
            df1=bridge_pts_joined,
            df2=gdf_centroid,
            src_column="JoinID",
            axis=1,
        )

        # ONLY 1 HEC POINT CAN BE ATTACHED TO A SEGMENT. CHOOSE ONLY 1 POINT PER SEGMENT.
        bridge_pts_joined_sub = bridge_pts_joined.drop_duplicates(
            subset="NearSegID", keep="first", inplace=False
        )

        # JOIN HEC POINTS TO RAW STREAM SEGMENTS, MUST DROP GEOMETRY BEFORE MERGING
        bridge_pts_joined_sub = bridge_pts_joined_sub.drop(columns="geometry")
        bridge_2_seg = pd.merge(
            gdf_segments_c,
            bridge_pts_joined_sub,
            how="left",
            left_on=["JoinID"],
            right_on=["NearSegID"],
        )

        bridge_2_seg_path = os.path.join(watershed_outputs_id,"bridge_joined2_segments.shp")
        bridge_2_seg.to_file(bridge_2_seg_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'bridge_joined2_segments'
        bridge_2_seg.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------
## JOIN BRIDGE CALCULATIONS TO FINAL STREAM SEGMENTS
## ---------------------------------------------------------------------------------------------------------------------
        # JOIN ONLY BRIDGE CALCULATIONS TO FINAL RESULTS
        bridge_pts_joined_sub = bridge_pts_joined_sub.drop(columns="RIVER_STA")
        bridge_pts_joined_sub.rename(
            columns={"RSta": "RStaBr", "NearSegID": "NearSegBr"}, inplace=True
        )
        bridge_2_seg_hec = pd.merge(
            gdf_hec45m_2_seg,
            bridge_pts_joined_sub,
            how="left",
            left_on=["JoinID"],
            right_on=["NearSegBr"],
        )

        # SET PROJECTION AND SAVE
        bridge_2_seg_hec.crs = output_crs
        bridge_2_seg_hec_path = os.path.join(watershed_outputs_id,"segment_dem_hec_streampow_bridge_prj.shp")
        bridge_2_seg_hec.to_file(bridge_2_seg_hec_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'segment_dem_hec_streampow_bridge_prj'
        bridge_2_seg_hec.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------
## RENAME TO BRIDGE STREAM POWER FINAL
## END OF BRIDGE ANALYSIS
## ---------------------------------------------------------------------------------------------------------------------
        gdf_bridgefinal = gpd.read_file(
            bridge_2_seg_hec_path
        ) # READ THE ORIGINAL SHAPEFILE
        copy_bridgefinal = os.path.join(watershed_outputs_id,"Final_Bridge_Stream_Power_Output.shp")
        # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
        gdf_bridgefinal.to_file(copy_bridgefinal, driver="ESRI Shapefile")
## ---------------------------------------------------------------------------------------------------------------------
## CONNECT CULVERT HECRAS DATA TO SEGMENTS
## ---------------------------------------------------------------------------------------------------------------------
    if culvert_analysis == "Y":
        # TRANSFORM HECRASFILE FOR FLOOD EVENTS (.CSV) INTO COMPATIBLE FORMAT
        # READ FILE AS DATAFRAME (DF) BY SKIPPING 9 ROWS AND SPECIFY COLUMN HEADERS
        cul_df = pd.read_csv(culv_hec_csv, header=[0, 1])

        # REMOVE UNNAMED CELLS IN COLUMN NAMES
        # FOR NEW VERSION OF PANDAS 0.21 AND OLD
        for i, columns in enumerate(cul_df.columns.levels):
            columns_new = columns.tolist()
```

```
            for j, row in enumerate(columns_new):
                if "Unnamed: " in row:
                    columns_new[j] = ""
            if (
                pd.__version__ < "0.21.0"
            ):  # HTTPS://STACKOVERFLOW.COM/A/48186976/716469
                cul_df.columns.set_levels(columns_new, level=i, inplace=True)
            else:
                cul_df = cul_df.rename(
                    columns=dict(zip(columns.tolist(), columns_new)), level=i
                )

        # COMBINE BOTH ROWS AS HEADERS
        col_raw = cul_df.columns
        col_raw.tolist()
        # NEW_COL = PD.INDEX([E[0] + E[1] FOR E IN COL_RAW.TOLIST()]) FOR GDB CLASS, LONG NAMES ARE ALLOWED
(I.E. "QCHANNEL_M3PERS")
        new_col = pd.Index(
            [e[0] for e in col_raw.tolist()]
        )  # ONLY NAME, NOT METRIC FOR SHAPEFILE
        cul_df.columns = new_col

        # TRANSFORM FIELD NAME BY REPLACE SPACE AND SYMBOLS WITH UNDERSCORE
        cul_df.columns = cul_df.columns.str.replace(" ", "")
        cul_df.columns = cul_df.columns.str.replace("(", "")
        cul_df.columns = cul_df.columns.str.replace(")", "")
        cul_df.columns = cul_df.columns.str.replace("/", "per")
        cul_df.columns = cul_df.columns.str.replace(".", "")
        cul_df.columns = cul_df.columns.str.replace("_", "")

        # SELECT ONLY NEEDED ROWS
        cul_df = cul_df[
            ["RiverSta", "ID", "Profile", "QCulvGroup", "CulvSpan", "QTotal"]
        ]

        # SHORTEN COLUMN NAME : ONLY FOR SHAPEFILE
        cul_df.rename(
            columns={
                "RiverSta": "RSta",
                "ID": "ID",
                "Profile": "Profile",
                "QCulvGroup": "QCul",
                "CulvSpan": "DCul",
                "QTotal": "QTot",
            },
            inplace=True,
        )

        # TRANSFORM COLUMNS WITH PROFILE - KEEP ONLY NUMBER
        # HEC_DF["PROFILE"] = HEC_DF["PROFILE"].STR.REPLACE(R'\D', R'') # NOT IN USE
        cul_df["Profile"].replace(
            regex=True, inplace=True, to_replace=r"\D", value=r""
        )

        # REPLACE ALL SPACE IN HEC_DF
        cul_df = cul_df.replace(" ", "", regex=True)

        # DROP ROWS WITH NONE
        cul_df = cul_df.dropna(axis=0, how="all")

        # DROP ROWS WITH NONE UNDER PROFILE
        cul_df = cul_df.dropna(subset=["Profile"], axis=0, how="all")

        # REPLACE ALL SPACE IN HEC_DF
        cul_df = cul_df.replace("_", "", regex=True)

        # SAVE THE LIST OF CULVERT IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
        culv_profile_list = list(cul_df["Profile"])
        culv_profile_list_drop = list(dict.fromkeys(culv_profile_list))

        # PIVOT TABLE BY MAKING THE PROFILE (I.E. STORM EVENT) DATA INTO COLUMNS
        # CONCATENATE THE RSTA WITH THE CULVERT ID
        cul_df["ID"] = cul_df["ID"].astype("int64")
        cul_df["CulvID"] = cul_df["RSta"].astype(str) + cul_df["ID"].astype(str)

        # SUBSET ONLY THE FIELDS NEEDED FOR PIVOTING
        cul_df_sub = cul_df[["CulvID", "Profile", "QCul", "DCul", "QTot"]]

        # HEC_DF_PIV = HEC_DF.PIVOT(INDEX='RSTA', COLUMNS = 'PROFILE')
        cul_df_piv = pd.pivot_table(
            cul_df_sub, index="CulvID", columns="Profile"
        )  # OLD CODE

        # GROUP STACKED COLUMNS
        cul_df_piv.columns = cul_df_piv.columns.map("".join)

        # FORMAT DF BY REMOVING DUPLICATES FOR JOINING TO PIVOTED TABLE
        cul_df_sub2 = cul_df[["RSta", "ID", "CulvID"]]
        cul_df_sub2_noduplicate = cul_df_sub2.drop_duplicates(
            ["RSta", "CulvID"], keep="last"
        )

        # ADD RIVER STATION TO PIVOTED DATA
        cul_df_piv_all = pd.merge(
            cul_df_piv,
            cul_df_sub2_noduplicate,
            how="left",
            left_on="CulvID",
            right_on="CulvID",
        )

        # PIVOT TABLE BY ATTACHING ID OF CULVERT (E.G. 1,2,3) TO COLUMN NAMES
        # SUBSET ONLY FIELDS NEEDED FOR PIVOTING
        cul_df_piv_all_sub = cul_df_piv_all.drop(["CulvID"], axis=1)
        cul_df_piv_all_sub = cul_df_piv_all_sub.fillna(0.0)

        # CONVERT ID TO STRING BEFORE ATTACHING TO COLUMN NAME
        cul_df_piv_all_sub["ID"] = cul_df_piv_all_sub["ID"].astype("int64")

        # SAVE THE LIST OF CULVERT IDS FOUND AT A RIVER STATION. THIS NUMBER WILL BE USED FOR FURTHER
CALCULATIONS
        culv_id_list = list(cul_df_piv_all_sub["ID"])
        culv_id_list_drop = list(dict.fromkeys(culv_id_list))

        # CONVERT COLUMN WITH CULVERT ID TO BE ATTACHED TO COLUMN NAME
        cul_df_piv_all_sub["ID"] = cul_df_piv_all_sub["ID"].astype(str)
        cul_df_piv_all_sub["ID"] = "_" + cul_df_piv_all_sub["ID"].astype(str)

        # PIVOT TABLE BASED ON ID
        cul_df_piv_all_sub_piv = pd.pivot_table(
            cul_df_piv_all_sub, index="RSta", columns="ID"
        )  # OLD CODE

        # GROUP STACKED COLUMNS
        cul_df_piv_all_sub_piv.columns = cul_df_piv_all_sub_piv.columns.map(
            "".join
        )

        # SAVE TRANSFORMED HEC RAS FILE
        cul_csv_f = os.path.join(watershed_outputs_id,"culvert_csv_formatted.csv")
        cul_df_piv_all_sub_piv.to_csv(cul_csv_f)
## ---------------------------------------------------------------------------------------------------------------------
## CONNECT TRANSFORMED CULVERT DATA TO GDB TABLE. TO CALCULATE CULVERT SCOUR AND CULVERT
OVERTOPPING, THE DATA FOR THE TOTAL DISCHARGE, DISCHARGE AT CULVERT AND CULVERT DIAMETER ARE
REQUIRED.
## ---------------------------------------------------------------------------------------------------------------------
        # ADD NEW COLUMNS TO CALCULATE CULVERT SCOUR AND OVERTOPPING
        culv_scour_label_list = []
        culv_overtop_label_list = []
```

```
        for i in culv_id_list_drop:
            for y in culv_profile_list_drop:
                new_lab = "SCul" + str(y) + "_" + str(i)
                culv_scour_label_list.append(new_lab)
                print(new_lab)
                # CREATE EMPTY FIELD
                cul_df_piv_all_sub_piv[new_lab] = ""
                # APPLY EQUATION
                qcul_lab = "QCul" + str(y) + "_" + str(i)
                print(qcul_lab)
                dcul_lab = "DCul" + str(y) + "_" + str(i)
                print(dcul_lab)
                # CALCULATE NEW FIELD
                g = 9.81
                denom = (g**0.5) * (cul_df_piv_all_sub_piv[dcul_lab] ** 2.5)
                brac = (cul_df_piv_all_sub_piv[qcul_lab] / denom) ** 0.45
                cul_df_piv_all_sub_piv[new_lab] = 2.07 * brac

        for i in culv_id_list_drop:
            for y in culv_profile_list_drop:
                new_lab = "OCul" + str(y) + "_" + str(i)
                culv_overtop_label_list.append(new_lab)
                # PRINT(NEW_LAB)
                cul_df_piv_all_sub_piv[new_lab] = ""
                # APPLY EQUATION
                qtot_lab = "QTot" + str(y) + "_" + str(i)
                qcul_lab = "QCul" + str(y) + "_" + str(i)
                # CALCULATE
                cul_df_piv_all_sub_piv[new_lab] = (
                    cul_df_piv_all_sub_piv[qtot_lab]
                    / cul_df_piv_all_sub_piv[qcul_lab]
                )

        # SUBSET ONLY CALCULATED VALUES, ADD RIVER STA AND SAVE
        merged_culv_ratios_lab = culv_scour_label_list + culv_overtop_label_list
        merged_culv_ratios_df = cul_df_piv_all_sub_piv[merged_culv_ratios_lab]

        # ADD THE COLUMN RSTA AND CONVERT THE INDEX TO INT
        merged_culv_ratios_df["RSta"] = merged_culv_ratios_df.index
        merged_culv_ratios_df = merged_culv_ratios_df.reset_index(drop=True)

        # SAVE TRANSFORMED DATA WITH NEW FIELDS
        merged_culv_ratios_path = os.path.join(watershed_outputs_id, "culv_scour_overtop_ratios_only.csv")
        merged_culv_ratios_df.to_csv(merged_culv_ratios_path)
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## RESHAPE POINTS FILE BY MERGING ALL STACKED POINTS (I.E. DIFFERENT CULVERTS AT 1 REACH) INTO ONE POINT
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # READ SHAPEFILE
        culv_shp = geopandas.read_file(culv_point)

        # DISSOLVE STACKED POINTS BY REACH ID
        culv_dissolved = culv_shp.dissolve(by="RIVER_STA")
        culv_dissolved["RIVER_STA"] = culv_dissolved.index

        # SAVE TRANSFORMED DATA WITH NEW FIELDS
        culv_dissolved_path = os.path.join(watershed_outputs_id,"stacked_culverts.shp")
        culv_dissolved.to_file(culv_dissolved_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'stacked_culverts'
        culv_dissolved.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT CULVERT CALCULATIONS TO A COPY OF RIVER STATIONS POINTS FILE
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # MAKE A COPY OF POINTS FILE
        culv_point_stack_df = geopandas.read_file(culv_dissolved_path)
        culv_point_stack_df = culv_point_stack_df[["RIVER_STA", "geometry"]]

        # JOIN CULVERT CALCULATIONS TO CULVERT POINTS SHAPEFILE
        culv_pts_joined = pd.merge(
            culv_point_stack_df,
            merged_culv_ratios_df,
            how="inner",
            left_on=["RIVER_STA"],
            right_on=["RSta"],
        )
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
## CONNECT CULVERT POINTS TO STREAM REACH
## ---------------------------------------------------------------------------------------------------------------------------------------------------------
        # OPEN A COPY OF SEGMENTS CENTROID FILE
        gdf_centroid_c = geopandas.read_file(seg_centroid_path)
        gdf_segments_c = geopandas.read_file(
            stream_segments_attribute_df_path_c
        )
        gdf_hec45m_2_seg = geopandas.read_file(copy_bridgefinal)

        # FUNCTION TO FIND NEAREST SEGMENT'S CENTROID TO HECDATA POINT
        def nearest(
            row,
            geom_union,
            df1,
            df2,
            geom1_col="geometry",
            geom2_col="geometry",
            src_column=None,
        ):
            """Find the nearest point and return the corresponding
            value from specified column."""
            # FIND THE GEOMETRY THAT IS CLOSEST
            nearest = (
                df2[geom2_col] == nearest_points(row[geom1_col], geom_union)[1]
            )
            # GET THE CORRESPONDING VALUE FROM DF2 (MATCHING IS BASED ON THE GEOMETRY)
            value = df2.loc[nearest, src_column].iloc[0]
            return value

        # CREATE MULTIPOINT: CENTROID OF SEGMENTS
        unary_union = gdf_centroid_c.unary_union

        # FIND ID OF LINE CLOSEST TO HEC POINT
        culv_pts_joined["NearSegID"] = culv_pts_joined.apply(
            nearest,
            geom_union=unary_union,
            df1=culv_pts_joined,
            df2=gdf_centroid_c,
            src_column="JoinID",
            axis=1,
        )

        # ONLY 1 HEC POINT CAN BE ATTACHED TO A SEGMENT. CHOOSE ONLY 1 POINT PER SEGMENT.
        culv_pts_joined_sub = culv_pts_joined.drop_duplicates(
            subset="NearSegID", keep="first", inplace=False
        )

        # JOIN HEC POINTS TO RAW STREAM SEGMENTS, MUST DROP GEOMETRY BEFORE MERGING
        culv_pts_joined_sub = culv_pts_joined_sub.drop(columns="geometry")
        culv_2_seg = pd.merge(
            gdf_segments_c,
            culv_pts_joined_sub,
            how="left",
            left_on=["JoinID"],
            right_on=["NearSegID"],
        )

        culv_2_seg_path = os.path.join(watershed_outputs_id,"culv_joined2_segments.shp")
        culv_2_seg.to_file(culv_2_seg_path, driver="ESRI Shapefile")

        # WRITE TO A GEOPACKAGE
        layer_name = 'culv_joined2_segments'
```

```
            culv_2_seg.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## JOIN CULVERT TO FINAL STREAM SEGMENTS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
            # JOIN ONLY CULVERT CALCULATIONS TO STREAM SEGMENTS WITH OTHER CALCULATIONS
            culv_pts_joined_sub = culv_pts_joined_sub.drop(columns="RIVER_STA")
            culv_pts_joined_sub.rename(
                columns={"RSta": "RStaCul", "NearSegID": "NearSegCul"}, inplace=True
            )
            culv_2_seg_hec = pd.merge(
                gdf_hec45m_2_seg,
                culv_pts_joined_sub,
                how="left",
                left_on=["JoinID"],
                right_on=["NearSegCul"],
            )

            culv_2_seg_hec_path = os.path.join(watershed_outputs_id,"segment_dem_hec_bridge_culvert_prj.shp")
            culv_2_seg_hec.to_file(culv_2_seg_hec_path, driver="ESRI Shapefile")

            # WRITE TO A GEOPACKAGE
            layer_name = 'segment_dem_hec_bridge_culvert_prj'
            culv_2_seg_hec.to_file(hec_analysis_file, layer=layer_name, driver='GPKG')
            # 1 MIN
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## RENAME TO CULVERT STREAM POWER FINAL
## END OF CULVERT ANALYSIS
## ------------------------------------------------------------------------------------------------------------------------------------------------------
            gdf_culfinal = gpd.read_file(
                culv_2_seg_hec_path
            )  # READ THE ORIGINAL SHAPEFILE
            copy_culfinal = os.path.join(watershed_outputs_id,"Final_Culvert_Stream_Power_Output.shp")
            # DEFINE THE NAME OF THE NEW SHAPEFILE AND FILEPATH
            gdf_culfinal.to_file(copy_culfinal, driver="ESRI Shapefile")
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        file_pattern = os.path.join(watershed_outputs_id, 'hec_points_joined2data.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'hec_points_within45m.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'hec45m_join2_segments.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'segment_dem_hec_connected_intp.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'segment_dem_hec_streampow_prj.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'segments_centroid.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_segments_attribute.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
        # DELETE ALL FILES ASSOCIATED WITH THE SHAPEFILE
        file_pattern = os.path.join(watershed_outputs_id, 'stream_segments_attribute_edited.*')
        file_list = glob.glob(file_pattern)
        for file_path in file_list:
            os.remove(file_path)
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## CREATING ONE SHAPEFILE OUT WITH ALL OF THE HECRAS FILES
## ------------------------------------------------------------------------------------------------------------------------------------------------------
        # DEFINE THE FILE PATHS OF THE SHAPEFILES TO BE MERGED
        file_paths = [copy_strm_net]

        if hec_analysis == "Y":
            file_paths.append(copy_hecfinal)

        if bridge_analysis == "Y":
            file_paths.append(copy_bridgefinal)

        if culvert_analysis == "Y":
            file_paths.append(copy_culfinal)

        # READ IN THE FIRST SHAPEFILE TO INITIALIZE THE MERGED GEODATAFRAME
        merged_gdf = gpd.read_file(file_paths[0])

        # LOOP THROUGH EACH FILE PATH AND MERGE THE SHAPEFILE DATA
        for path in file_paths[1:]:
            if os.path.isfile(path):
                try:
                    gdf = gpd.read_file(path)
                    merged_gdf = pd.concat([merged_gdf, gdf], ignore_index=True, sort=False)
                except:
                    print(f"Error reading file: {path}")
            else:
                print(f"File not found: {path}")

        # WRITE THE MERGED GEODATAFRAME TO A NEW SHAPEFILE
        allanalysisdata = str(watershed_outputs_id) + "\HECRAS_Data_Merged.shp"
        merged_gdf.to_file(allanalysisdata)
        print("Merged all HEC-RAS Outputs")

    # END1 = TIME.TIME()
    # PRINT("THE TIME OF EXECUTION OF HECRAS SINGLE IS :",(END1-START1), "SECONDS")
    print(counter)
    hec_analysis_file = None
## ------------------------------------------------------------------------------------------------------------------------------------------------------
## END OF MULTIPLE HECRAS ANALYSIS
end4 = time.time()
print("The time of execution of HECRAS Multiple is :", (end4 - start4), "seconds")

gc.collect()
##########################################################################################################################################
################################################
##########################################################################################################################################
################################################
## MERGE THE OUTPUT SHAPEFILES TOGETHER
def combine_shapefiles(main_folder, filenames, common_field, output_file):
    # GET ALL FILE PATHS IN THE MAIN FOLDER AND ITS SUBFOLDERS
    file_paths = []
    for filename in filenames:
        file_paths.extend(
            glob.glob(os.path.join(main_folder, "**", filename), recursive=True)
        )

    # INITIALIZE AN EMPTY GEODATAFRAME TO STORE THE COMBINED DATA
    combined_data = gpd.GeoDataFrame()

    common_crs = "EPSG:" + str(dem_espg2int)

    # ITERATE OVER EACH FILE PATH
    for file_path in file_paths:
```

```
        # PRINT THE FILE PATH
        print("Found file:", file_path)

        # READ THE SHAPEFILE INTO A GEODATAFRAME
        data = gpd.read_file(file_path)

        # PRINT THE CRS OF THE CURRENT FILE
        print(f"CRS of {file_path}: {data.crs}")

        # CHECK IF THE GEODATAFRAME HAS A CRS; IF NOT, SET IT TO THE COMMON CRS
        if data.crs is None:
            print(f"No CRS found for {file_path}. Setting CRS to {common_crs}.")
            data = data.set_crs(common_crs)
        elif data.crs != common_crs:
            # CONVERT THE CRS TO THE COMMON CRS IF NECESSARY
            data = data.to_crs(common_crs)
            print(f"Converted CRS of {file_path} to {common_crs}")

        # APPEND THE DATA TO THE COMBINED_DATA GEODATAFRAME
        combined_data = pd.concat([combined_data, data], ignore_index=True)


    # MERGE THE DATA BASED ON THE COMMON FIELD
    merged_data = combined_data.dissolve(by=common_field)

    # SAVE THE MERGED DATA TO A SHAPEFILE
    merged_data.to_file(output_file)
    print("Merged data saved to", output_file)

    # PRINT THE ENTIRE ARRAY OF FILE PATHS
    print("All file paths found:")
    for file_path in file_paths:
        print(file_path)

# SPECIFY THE MAIN FOLDER, FILENAMES, COMMON FIELD, AND OUTPUT FILE NAME
filenames = [
    "HECRAS_Data_Merged.shp",
    "Final_Land_Use_Stream_Power_Output.shp",
    "Final_Land_Use_EngSWM_Stream_Power_Output.shp"
]  # REPLACE WITH THE SPECIFIC FILENAMES YOU WANT TO SEARCH FOR
common_field = "pXY"  # REPLACE WITH THE ACTUAL COMMON COLUMN NAME
output_file = os.path.join(data_dir, "All Scenario Outputs Combined.shp")

# CALL THE FUNCTION TO COMBINE THE FILES
combine_shapefiles(data_dir, filenames, common_field, output_file)
#############################################################################################################################
#################################################
#############################################################################################################################
################################################
## TOTAL PROGRAM LENGTH
end = time.time()
print("The time of execution of SPINpy 1 is :", (end - start), "seconds")

if check_boundary == 0 and lu_analysis == "Y":
    print("The time of execution of LU Single is :", (end1 - start1), "seconds")

if check_boundary == 0 and hec_analysis == "Y":
    print("The time of execution of HECRAS Single is :", (end2 - start2), "seconds")

if check_boundary > 0 and lu_analysis == "Y":
    print("The time of execution of LU Multiple is :", (end3 - start3), "seconds")

if check_boundary > 0 and hec_analysis == "Y":
    print("The time of execution of HECRAS Multiple is :", (end4 - start4), "seconds")

profiler.disable()

# SAVE THE PROFILER OUTPUT TO A FILE
profileres = os.path.join(data_dir, "profile_results.txt")
with open(profileres, 'w') as f:
    stats = pstats.Stats(profiler, stream=f)
    stats.sort_stats(pstats.SortKey.CUMULATIVE)
    stats.print_stats()
```