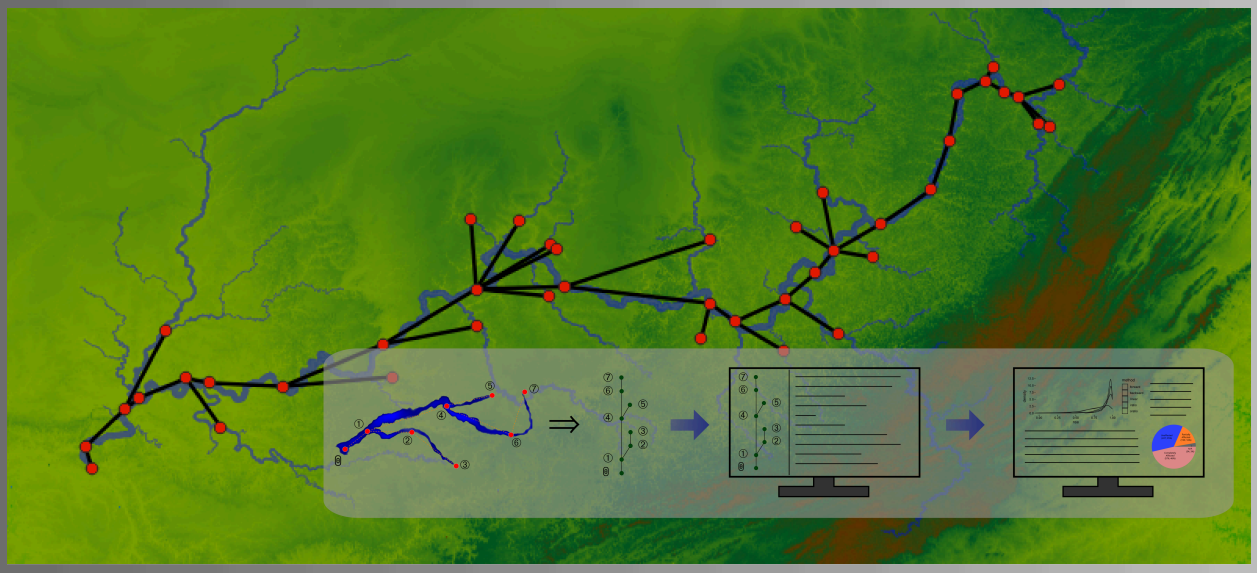
Network Analysis and Data Integration (NADI) System

User Manual



NADI Book Version: 0.7.0

Web Version: https://nadi-system.github.io/

Gaurav Atreya
2025-08-06

Contents

Preface

NADI is currently under active development. As such does not have stable API yet, and many of the concepts explained in this book might not work yet.

If you still want to use it for your projects, please do them with the knowledge that the API might change in next versions, and you might have to keep it updated until the system is stable. If you have any problems with the program, or would like some new features, please make an github issue, we will try to accomodate it if it fits within the scope of the program.

The PDF version of this book (Experimental)

The PDF Book is generated using automated script, so it might have some problems, please refer to the web version of the NADI Book if there are problematic/incomplete contents.

Acknowledgements

Acknowledgements:

Thank you everyone who has been consistently testing this software throughout the development and providing feedbacks. Specially the members of Water System Analysis Lab in University of Cincinnati.

Funding:

Grant: #W912HZ-24-2-0049 Investigators: Ray, Patrick 09-30-2024 – 09-29-2025 U.S. Army Corps of Engineers Advanced Software Tools for Network Analysis and Data Integration (NADI) 74263.03 Hold Level:Federal

1. Why NADI?

1.1. What is NADI System?

Network Analysis amd Data Integration (NADI) System is a system of programs made to make network based data analysis easier and more accessible.

It consists of multiple tools, that perform two important functions, network detection and network analysis. First part is done throuh the **Geographic Information (GIS) Tool**, while the second part is done using a **Domain Specific Programming Language (DSPL)** called NADI Task system.

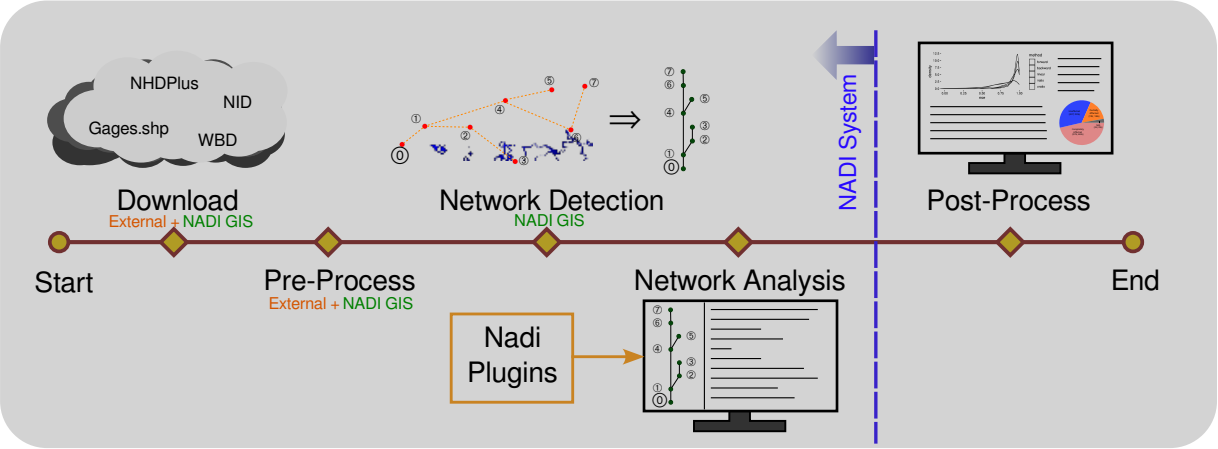


Figure 1: NADI System Workflow

1.2. Why use NADI System?

Hydrologic modeling involves the integration of diverse data to simulate complex (and often poorly understood) hydrological processes. The analysis of complex hydrological processes often requires using domain specific calculations, and the visual representation requires the creation of custom maps and plots. Both of which can be a repetitive and error-prone processes, diverting time from data interpretation and scientific inquiry. Efficient methods are needed to automate these tasks, allowing researchers to focus on higher-level analysis and translation of their findings.

Current solution to that problem is to either use general purpose programming languages like Python, R, Julia, etc., or use domain specific software packages to increase the reliability of the tasks. Domain Specific Programming Languages (DSPLs) like the NADI Task system provides better syntax for domain specific tasks, while also are general purpose enough for users to extend it for their use cases. NADI System is trying to be the software framework that can connect those two by integrating with various softwares and providing a intuitive way to do network based data analysis.

Some example functionality of NADI system includes:

- Detection of upstream/downstream relationships from stream network,
- Network based programming using an extensible custom programming language,

- Interactive plots and reports generation,
- Import/export from/to various GIS data formats, etc.

1.2.1. Network Based Data Analysis

If you have data that are network based, like in case of data related to points in a river. NADI provides a text representation of the network that can be manually created with any text editor, or through NADI GIS tool.

Some domains where the data are network based (directed tree graph) are:

- River networks,
- File/Directory structures,
- Human Resources in a Company,
- Decision Tree / Policy Tree,
- Modeling work with dependencies to component models,

1.2.2. Task System

The Domain Specific Programming Language (DSPL) developed for network analysis in NADI makes network analysis simple and intutive. So, it is easier to understand, interpret and catch mistakes. While the NADI IDE has network visualization tools built in that can help you visualiza the network attributes for visual analysis.

For example, implementing “cumulative sum of streamflow” in nadi:

```
node<inputsfirst>.cum_sf = node.streamflow + sum(inputs.streamflow);
```

The trying to do this in Python while making sure input nodes are run before the output. So you might have to write a recursive algorithm like this:

```
def cum_sf(node):
  node.cum_sf = node.streamflow + sum([cum_sf(i) for i in node.inputs()])
  return node.cum_sf

cum_sf(network.outlet())
```

While a common mistake people might make is to write a simple loop like this:

```
for node in network.nodes():
    node.cum_sf = node.streamflow + sum(
      [i.streamflow for i in node.inputs()]
  )
```

Which doesn't make sure input nodes are run before output in this case, and can error out when some variables are not present. NADI provides special syntax for cases where you can make sure variables exist before running something.

1.2.3. Extensibility

NADI has two types of plugin systems, which means users can write their own analysis in any programming language and have it interact with NADI through attributes, or they can write it in rust and have even more direct interaction.

2. Who this book is for

If you are a developer and want an API documentation for the data types used in the NADI Rust library refer to the [docs.rs `nadi_core` page](https://docs.rs/nadi_core/latest/nadi_core/). But if you want to understand the core concepts, use cases, and examples then proceed.

This book has sections explaining the concepts of the NADI system, its developmental notes, user guide and developer guide.

Hence it can be useful for people who:

- Want to understand the concepts used in NADI,
- Want to use NADI system for their use case,
- Want to develop plugin system for NADI,
- Want to contribute to the NADI system packages, etc.

Although not intended, it might include resources and links to other materials related to Rust concepts, Geographical Information System (GIS) concepts, Hydrology concepts, etc. that people could potentially benefit from.

Please note that, this book assumes you have some knowledge of programming, like control flow, loops, functions, variables, etc. As as well some knowledge of GIS if you are

reading network detection + GIS visualization specific functions. Those concepts will not be covered in this book. And if you are reading the developer reference or compiled plugin development, some knowledge of rust is assumed, but this is not necessary for the users of NADI that will only write code in NADI.

3. How to use this book

You can read this book sequentially to understand the concepts used in the NADI system. And then go through the references sections for a specific use cases you want to get into the details of.

- If you are in a hurry, but this is your first time reading this book,

at least read the Core Concepts, then refer to the section you are interested in.

- If you want to know a specific details, click on the search icon at the top left of the book to get the search bar. You can search text there and visit the pages.
- If you want to learn about the QGIS Plugin, NADI GIS, goto “Network Detection (GIS)” section.
- If you want to learn about the Domain Specific Programming Language (DSL), refer to the “Network Analysis (DSL)” section.
- Learn by Example contains some simple examples you can follow to learn the basic syntax of the Task System.
- If you want more detailed examples of the use of Task System, refer to the chapters in “Example Research Problems” section on the sidebar.
- If you want reference for functions used in Task System goto “Plugin Functions”, the Internal Plugins section contains details on the functions that are available with nadi system, while the external plugins are plugins that were loaded while this book was compiled.

If you have suggestions on the formatting, or arrangement of chapters in this book, please make an issue on the GitHub repository for this book.

3.1. Code Blocks

The code blocks will have example codes for various languages, most common will be `string template`, `task`, `python`, and `rust` codes.

String template and task have custom syntax highlights that is intended to make it easier for the reader to understand different semantic blocks.

For `task` scripts/functions, if relevant to the topic, they might have `Results` block following immediately showing the results of the execution.

For example:

```
network load_file("./data/mississippi.net")
node[ohio] render("{_NAME:case(title)} River")
```

Results:

```
{
  ohio = "Ohio River"
}
```

Task and Rust code block might also include lines that are needed to get the results, but hidden due to being irrelevant to the discussion. In those cases you can use the eye icon on the top right side of the code blocks to make them visible. Similarly use the copy icon to copy the visible code into clipboard.

3.2. String Template Syntax Highlight

The syntax highlight here in this book makes it so that any unknown transformers will be marked for easy detection to mistakes.

```
This shows var = {var:unknown()}, {_var:case(title)}
```

Besides this, the syntax highlight can help you detect the variables part (within `{}`), lisp expression (within `=()`), or commands (within `$()`) in the template.

Note: commands are disabled, so they won't run during template rendering process. But if you are rendering a template to run as a command, then they will be executed during that process.

4. How to Cite

4.1. How to Cite

The sections below show you a bibliography entry in ASCE format, and BibTeX format that you can copy.

4.1.1. Journal Papers: TODO

The papers are currently still being worked on, and will be added here when they are published.

4.1.2. This book

You can cite the link to this book as follows Make sure to replace Accessed Data by today's date.

Atreya, G. 2025. "Network Analysis and Data Integration (NADI)." Accessed May 1, 2025. https://nadi-system.github.io/.

```
@misc{PrefaceNetworkAnalysis,
  title = {Network {{Analysis}} and {{Data Integration}} ({{NADI}})},
  author = {Atreya, Gaurav},
  year = {2025},
  url = {https://nadi-system.github.io/},
  urldate = {2025-05-02}
}
```

4.2. Works using NADI System

Atreya, G., G. Mandavya, and P. Ray. 2024. "Which came first? Streamgages or Dams: Diving into the History of Unaltered River Flow Data with a Novel Analytical tool." H51L-0865.

```
@inproceedings{atreyaWhichCameFirst2024,
  title = {Which Came First? {{Streamgages}} or {{Dams}}: {{Diving}} into the
```

```
{{History}} of {{Unaltered River Flow Data}} with a {{Novel Analytical}} Tool},
  shorttitle = {Which Came First?},
  booktitle = {{{AGU Fall Meeting Abstracts}}},
  author = {Atreya, Gaurav and Mandavya, Garima and Ray, Patrick},
  year = {2024},
  month = dec,
  volume = {2024},
  pages = {H51L-0865},
  urldate = {2025-06-03},
  annotation = {ADS Bibcode: 2024AGUFMH51L.0865A}
}
```

NADI System Setup

5. Introduction

5.1. Network Analysis and Data Integration (NADI)

NADI is group of software packages that facilitate network analysis and do data analysis on data related to network/nodes.

NADI System consists of:

Tool	Description
NADI GIS	Geographic Information (GIS) Tool for Network Detection
NADI Task System	Domain Specific Programming Language
NADI Plugins	Plugins that provide the functions in Task System
NADI library	Rust and Python library to use in your programs
NADI CLI	Command Line Interface to run NADI Tasks
NADI IDE	Integrated Development Environment to write/ run NADI Tasks

The github repositories consisting of source codes:

Repo	Tool
nadi-gis	Nadi GIS
nadi-system	Nadi CLI/ IDE/ Core
nadi-plugins-rust	Sample Plugins
nadi-book	Source for this NADI Book

5.1.1. Workflow

A Typical workflow in NADI System consists of the follwing 4 processes:

1. Download Data
2. Pre-Process Data
3. Network Detection (using NADI GIS)
4. Network Analysis (Using NADI System's DSL, and Plugins)
5. Post Process

The figure below shows the order of how the components of the NADI System (blue) is used along side external tools (black).

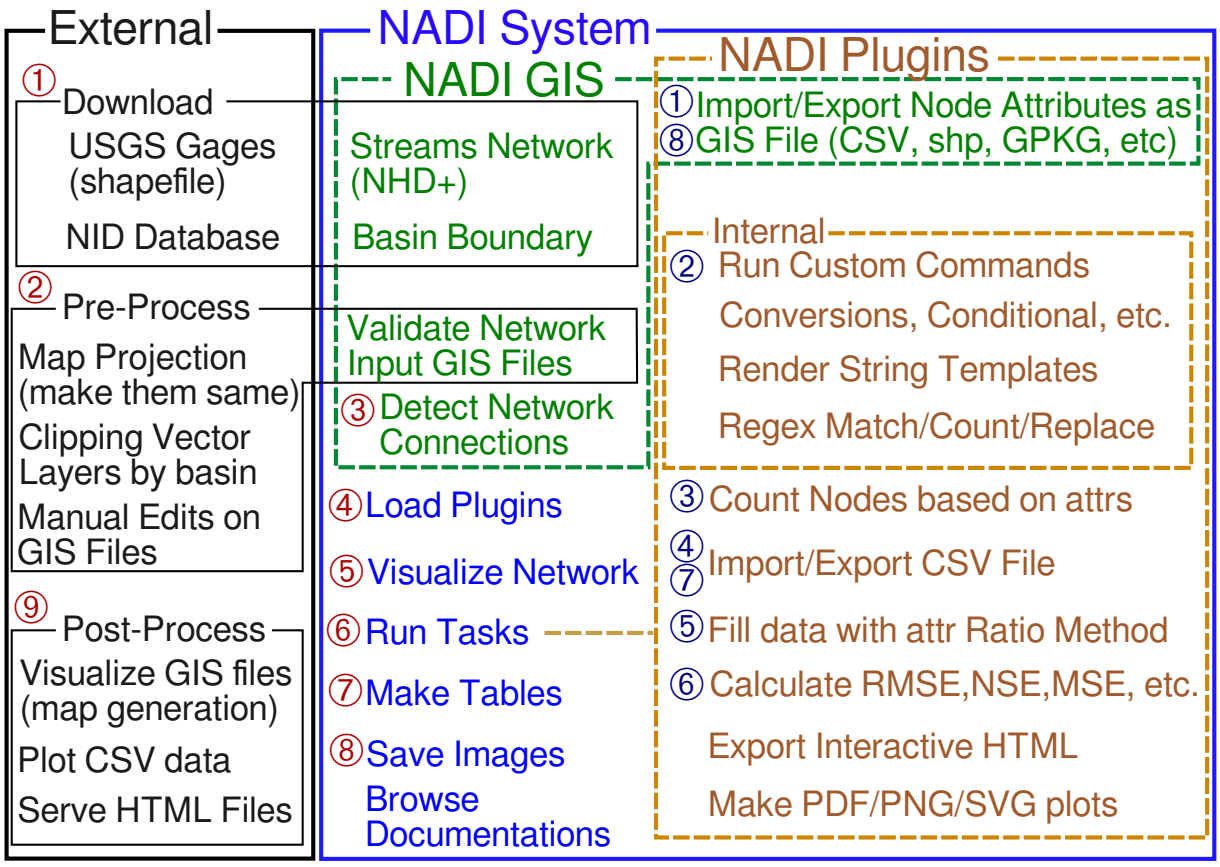


Figure 2: NADI System Workflow

Here the numbers in red circles are the order of use for different tools. Here the “Run Tasks” step represents running the NADI DSL Code in the System, so it is further divided into the tasks inside the code. The numbers on the blue circles show a typical use case of the DSL to perform a research work.

For exact details on what a typical research workflow involving the NADI DSL is, refer to the examples.

5.1.2. NADI GIS

Geographic Information (GIS) Tool for Network Detection. The main purpose of the NADI GIS is to find the network connectivity between a set of points using a stream network (which can be developed from elevation models, or downloaded from national databases).

NADI GIS can be used as a terminal command or QGIS plugin, refer to installation section for how to install it.

5.1.3. NADI Task System

Task System is a Domain Specific Programming Language (DSL) that is designed for river network analysis. This is the main core of the network analysis. This is included when you install NADI as a library, CLI or GUI.

5.1.4. NADI Plugins

The functions available to call in the task system comes from plugins. There are many internal plugins with core functions already available, while users can load their own plugins for other functions.

Refer to the plugins section of the book for more details on how to use plugins, how to write them and what to keep in mind while using them.

5.1.5. NADI libraries

Rust and Python library to use in your programs. Rust library `nadi_core` is available to download/use from `cargo` with the command `cargo add nadi_core`.

While Python library requires you to clone the repo and build it with `maturin` (for now). Future plan for it includes publishing it in pypi.

5.1.5.1. Rust Libraries

If you are not writing your own rust programs or plugins, you can skip this section.

There are three rust libraries:

Library	Use
`nadi_core`	Core library with data types, and plugin structure
`nadi_plugin`	Rust Procedural macro library to write nadi plugins
`string_template_plus`	Library for string templates with variables

Everything is loaded by `nadi_core` so you don't need to load them separately.

5.1.5.2. NADI Python

While using NADI from python library, you only have access to nadi data types (Node, Network, etc), and the plugin functions, which are enough for most cases as python language syntax, variables, loops etc will give you a lot of flexibility on how to do your own analysis. The python module is structured as follows:

```
nadi [contains Node, Network, etc]
 +-- functions
 | +-- node [contains node functions]
 | +-- network [contains network functions]
 | +-- env [contains env functions]
 +-- plugins
   +-- <plugin> [each plugin will be added here]
   | +-- node [contains node functions]
   | +-- network [contains network functions]
   | +-- env [contains env functions]
   +-- <next-plugin> and so on ...
```

The functions are available directly through `functions` submodule, or through each plugin in `plugins` submodule. An example python script looks like this:

```python
import nadi
import nadi.functions as fn

net = nadi.Network("data/ohio.network")
for node in net.nodes:
    try:
        _ = int(node.name)
        node.is_usgs = True
    # this just shows how nadi functions can be called from python
    # for simple functions please use the python native functions
        print(fn.node.render(node, "Node {_NAME} is USGS Site"))
    except ValueError:
        node.is_usgs = False
```

This code shows how to load a network, how to loop through the nodes, and use python logic, or use nadi functions for the node and assign attributes.

More detail on how to use NADI from python will be explained in NADI Python chapter.

5.1.6. NADI CLI

Command Line Interface to run NADI Tasks.

This can run nadi task files, syntax highlight them for verifying them, generate markdown documentations for the plugins. The documentations included in this book (Function List and each plugin's page like Attributes Plugin `attributes`) are generated with that. The documentation on each plugin functions comes from their docstrings in the code, please refer to how to write plugins section of the book for details on that.

The available options are shown below.

```
Usage: nadi [OPTIONS] [TASK_FILE]

Arguments:
  [TASK_FILE]  Tasks file to run; if `--stdin` is also provided this runs before
stdin

Options:
  -C, --completion <FUNC_TYPE>   list all functions and exit for completions
[possible values: node, network, env]
  -c, --fncode <FUNCTION>        print code for a function
  -f, --fnhelp <FUNCTION>        print help for a function
  -g, --generate-doc <DOC_DIR>   Generate markdown doc for all plugins and functions
  -l, --list-functions           list all functions and exit
  -n, --network <NETWORK_FILE>   network file to load before executing tasks
  -p, --print-tasks              print tasks before running
  -P, --new-plugin <NEW_PLUGIN>  Create the files for a new nadi_plugin
  -N, --nadi-core <NADI_CORE>    Path to the nadi_core library for the new
nadi_plugin
  -s, --show                     Show the tasks file, do not do anything
  -S, --stdin                    Use stdin for the tasks; reads the whole stdin
before execution
  -r, --repl                     Open the REPL (interactive session) before exiting
  -t, --task <TASK_STR>          Run given string as task before running the file
  -h, --help                     Print help
  -V, --version                  Print version
```

5.1.7. NADI IDE

NADI Integrated Development Environment (IDE) is a Graphical User Interface (GUI) for the users to write/ run NADI Tasks.

As seen in the image below, IDE consists of multiple components arranged in a tiling manner. You can drag them to move them around and build your own layout. When you start IDE it suggests you some layouts and what to open. You can use the buttons on the top right of each pane to:

- change pane type
- vertically split current pane
- horizontally split current pane
- fullscreen current page/ restore layout if it's fullscreen
- close current pane

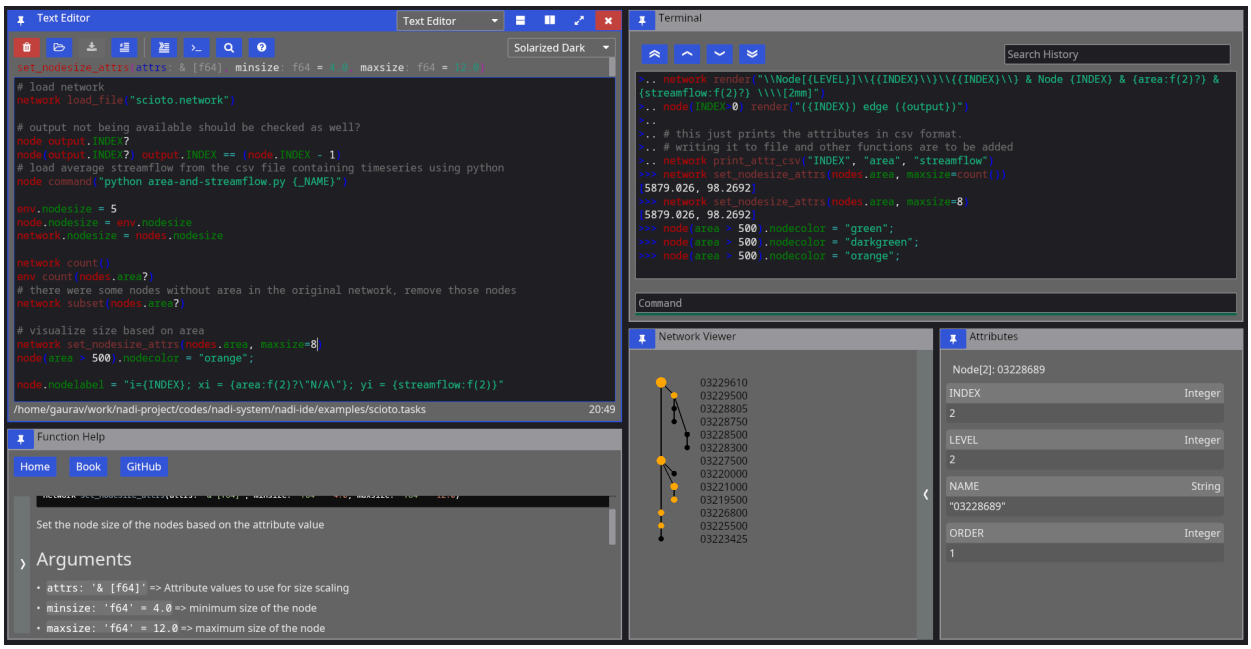


Figure 3: Screenshot of NADI IDE

It has the following components:

5.1.7.1. Text Editor

Open text files, edit and save them.

It comes with syntax highlighting for most languages. And custom highlight for tasks and network files.

For Tasks file, it can also show you function signatures on top so you can write tasks easily, knowing what arguments the function needs and what the default values are.

While open inside IDE, it can also run the tasks by sending them to the terminal, or search help documentations on functions. Hover over the buttons on the top row to see which button does what, and the keyboard shortcut to use them as well.

5.1.7.2. Terminal

Terminal is there so you can run NADI in a interactive session. Read Eval Print Loop (REPL) of NADI here is meant mostly to be used inside the IDE to evaluate the tasks from editor, but you can open it independently as well.

5.1.7.3. Function Help

This is a GUI with the list of all available plugin functions. You can expand the sidebar on left to search and browse functions. You can filter by type of function (node, network, env) with the buttons. When you click a function you can read its documentation on the right side.

Capabilities of the `iced` GUI libraries are limited right now, so you cannot select or copy text from the help. Please refer to the documentation online to do that. Or generate the documentation locally using `nadi-cli` tool.

5.1.7.4. Network Viewer

This is a pane where network is visualized, this is a very basic visualization to see the connections and is not optimized for drawing. Please avoid using this pane (making it visible) in case of large networks as it takes a lot of computation to draw this each frame.

5.1.7.5. Attribute Browser

When you click on a node on Network Viewer it will open/update showing the attributes of that node. There is no way to edit the attributes from here, which is intensional design as attributes should be assigned from tasks so that they are reproducible. For temporary assignments use the terminal.

5.1.7.6. SVG Viewer

This is a basic utility that can open a SVG file from disk and visualize it. You can click the refresh button to re-read the same file. This is intended for a quick way to check the SVG saved/exported from tasks. This is not a full fledge SVG renderer, so open them in image viewers or browsers to see how it looks.

5.2. Trivia

- Nadi means River in Nepali (and probably in many south asian languages).
- First prototype of NADI was Not Available Data Integration, as it

was meant to be an algorithm to fill data gaps using network information, but it was modified to be more generic for many network related analysis.

6. Installation

6.1. Installation

NADI System is a suite of software packages each have different installation methods. Some of the packages are uploaded to `crates.io` (rust) and `pypi` (python). For others, you can either get the compiled binaries from the Releases page of the github repo [windows]. Or you can get the source code using `git`, and using `cargo` build the packages [all OS].

Program	Linux	Windows	Mac	Android (Termux)
NADI CLI	yes	yes	yes	yes
NADI IDE	yes	yes	yes	no
mdbook nadi	untested	yes	untested	yes
`nadi-py`	yes	yes	yes	yes
QGIS Plugin	yes	yes	untested	no

Untested means it should work in theory, but I have not tested it.

6.1.1. Packages

For `nadi-py` you can use `pip`:

```
pip install nadi-py
```

For `nadi-cli` you can use `cargo`:

```
cargo install nadi
```

6.2. Downloading Binaries

Goto the repo of each component and refer to the releases section for binaries of different versions.

- nadi-system binaries
- nadi-gis binaries
- plugins binaries

To setup the nadi-systm to load the plugins you have to place them inside the directory included in the `NADI_PLUGIN_DIRS` environmental variable. Refer to your Operating System's documentation on how to set environemental variables.

The binaries should be able to run directly without needing extra steps. If you get a security warnings because the binaries are not signed, you might have to ignore it.

For QGIS Plugin, you can install it from Plugins Wizard on QGIS if you turn on experiemental plugins. But the `nadi-gis` binary should be available from PATH (i.e. you can call it from terminal).

6.3. Building from Source

This is currently the preferred way of installing `nadi-system` (and `nadi-gis` for Linux and MacOS). Although it includes a bit more steps this makes sure the compiled program is compatible with your OS.

6.3.1. Prerequisites

The prerequisites for building from source are:

- `git` [Optional]: to clone the repo, you can directly download zip from github
- `cargo`: To build the binaries from source.
- `gdal` [Optional]: Only for `nadi_gis` binary and plugin.

To install `git` refer to the instructions for your operating system from the official page.

For `cargo` follow the instructions to install rust toolsets for your operating system from the official page

Installing `gdal` can be little complicated for windows. For Linux, use your package manager to install `gdal` and/or `gdal-dev` package. Mac users can also install `gdal` using `homebrew`. For

windows, follow the instructions from official website, after installation you might have to make some changes to environmental variables to let `cargo` know where your `gdal` binaries/ header files are for the compilation to be successful. More details will be provided in the NADI GIS section.

If you use Linux or Mac (with homebrew), then the installation of prerequisites should be easy. But if you do not have the confidence to setup `gdal` for compiling `nadi_gis` use the binaries provided for them from the previous steps.

6.3.2. NADI System

It will build the binaries for `nadi`, `nadi-ide`, `nadi-help`, `nadi-editor`, etc. `nadi` is the command line interface to run nadi tasks, parse/validate syntax etc. While `nadi-ide` is the program to graphically develop nadi tasks and run them.

Assuming you have `git` and `cargo`,

```
git clone https://github.com/Nadi-System/nadi-system
cd nadi-system
cargo build --release
```

To run one of the binary from nadi system, use the command `cargo run` with binary name.

For example, the following will run the `nadi-ide`:

```
cargo run --release --bin nadi-ide
```

The compiled binaries will be saved in the `target/release` directory, you can copy them and distribute it. The binaries do not need any other files to run.

The plugins files if present in the system are automatically loaded from `NADI_PLUGIN_DIRS` environmental variable. Look into installing the plugin section below.

Note: all programs will compile and run in Windows, Linux, and MacOS, while only `nadi-cli` and `mdbook-nadi` will run in Android (tmux). `nadi-ide` and family need the GUI libraries that are not available for android (tmux) yet.

6.3.3. NADI GIS

NADI GIS uses `gdal` to read/write GIS files, so it needs to be installed. Please refer to gdal installation documentation for that.

6.3.3.1. Windows

First download compiled gdal from here:

- https://www.gisinternals.com/sdk.php

Then download clang from here:
- https://github.com/llvm/llvm-project/releases

Extract it into a folder, and then set environmental variables to point to that:

- `GDAL_VERSION`: Version of gdal e.g. '3.10.0'
- `LIBCLANG_PATH`: Path to the `lib` directory of `clang`
- `GDAL_HOME`: Path to the `gdal` that has the subdirectories like `bin`, `lib`, etc.

You can also follow the errors from the rust compilers as you compile to set the correct variables.

Finally you can get the source code and compile `nadi-gis` with the following command:

```
git clone https://github.com/Nadi-System/nadi-gis
cd nadi-gis
cargo build --release
```

This will generate the `nadi-gis` binary and `gis.dll` plugin in the `target/release` folder, they need to be run along side the `gdal` shared libraries (`.dlls`). Place the binaries in the same folder as the `dlls` from `gdal` and run it. To use the `gis.dll` plugin from `nadi`, `nadi-ide`, etc. same thing applies there, those binaries should be run with the `gdal`'s `dlls` to be able to load the `gis` plugin.

6.3.3.2. Linux and Mac

Assuming you have `git`, `cargo`, and `gdal` installed in your system you can build it like this:

```
git clone https://github.com/Nadi-System/nadi-gis
cd nadi-gis
cargo build --release --features bindgen
```

The `bindgen` feature will link the `nadi-gis` binary with the `gdal` from your system. So that you do not have to distribute `gdal` with the binary for your OS.

If you do not have `gdal` installed in your system, then you can still build the nadi-gis without the `bindgen` feature. This will still require `gdal` to be available and distributed with the binary.

```
cargo build --release
```

6.3.3.3. QGIS Plugin

The `nadi-gis` repo also contains the QGIS plugin that can be installed to run it through QGIS. The plugin will use the `nadi-gis` binary in your `PATH` if available. And it also contains the nadi plugin that can be loaded into the nadi system to import/export GIS files into/from the system.

You can download the zip file for plugin from releases page, and use the "Install from Zip" option on QGIS plugins tab. Or copy the `nadi` directory inside `qgis` to your python plugin directory for qgis.

Refer to the QGIS plugins page for more instructions. In future we are planning on publishing the plugin so that you can simply add it from QGIS without downloading from here.

6.3.3.4. NADI GIS Plugin

The NADI plugin on this repo provides the functions to import attributes, geometries from GIS files, and export them into GIS files.

6.3.4. NADI Plugins

Out of the two types of plugins, the executable plugins are just simple commands, they do not need to be installed along side NADI System, just make sure the executables that you are using from NADI System can be found in path. A simple way to verify that is to try to run that from terminal and see if it works.

The compiled plugins can be loaded by setting the `NADI_PLUGIN_DIRS` environmental variable. The environment variable should be the path to the folder containing the nadi plugins

(in `.dll`, `.so`, or `.dylib` formats for windows, linux and mac). You can write your own plugins based on our examples and compile them.

Officially available plugins are in the `nadi-plugins-rust` directory.

Assuming you have `git` and `cargo`,

```
git clone https://github.com/Nadi-System/nadi-plugins-rust
cd nadi-gis
cargo build --release
```

The plugins will be inside the `target/release` directory. Copy them to the `NADI_PLUGIN_DIRS` directory for nadi to load them.

You can take any one of the plugins as an example to build your own, or following the plugin development instructions from the plugins chapter.

7. Plugins

Plugins allow users to extend the use case of the NADI System by adding more functions or scripts. User are expected to only use plugins from trusted sources, or develop it in-house. Although the compiled plugin functions have their code exposed in their documentation for the transparency purposes even if the source code is not available, always make sure the plugin you run are not malicious.

There are two types of nadi plugins. Compiled plugins (shared libraries) are loaded dynamically from shared libraries, while executable plugins are called as shell commands. Refer to Plugins section of core concepts for more details.

7.1. Compiled Plugins

Compiled plugins are shared libraries (`.so` in linux, `.dll` in windows, and `.dylib` on macOS). They can be generated by compiling the nadi plugin in rust, or you can download the correct plugin for your OS and `nadi_core` version from the plugin repositories. It is recommended to only use plugins from trusted source.

To setup the nadi-systm to load the compiled plugins you have to place them inside the directory included in the `NADI_PLUGIN_DIRS` environmental variable. Refer to your Operating System's documentation on how to set environemental variables.

The compiled plugins are loaded when NADI is starting up, there is no way to `hot load` or `reload` the plugins, so you need to reopen the nadi program itself (CLI, IDE, etc) if you want to load new/updated plugin functions.

Once the plugins are loaded, the functions are directly available from the nadi task system, they'll act similar to the internal plugin functions.

7.2. Executable Plugins

Executable plugins are terminal commands, you set it up as you'd set any other terminal programs, by making sure the program is in `PATH` and can be executed from terminal. Linux and Mac do them mostly by default, while in Windows you might have to check the box saying something along the lines of "include this in path" during installation, or manually edit the `PATH` in "Environment Variables".

For example, if you want to call python scripts, make sure you can run `python --version` in terminal and get a response.

You can also check it using the `command` function:

```
network command("python --version", echo=true)
network command("Rscript --version", echo=true)
network command("julia --version", echo=true)
```

Results:

```
$ python --version
Python 3.13.5

$ Rscript --version
Rscript (R) version 4.5.1 (2025-06-13)

$ julia --version
julia version 1.11.6
```

Here we can see, the commands that ran successfully and returned a version are valid.

To write scripts and run them from nadi refer to Executable Plugins section on Plugin Developer Guide.

Network Detection (GIS)

8. NADI GIS

8.1. NADI GIS

NADI GIS is available as a CLI tool and QGIS plugin, the CLI tool has the following functions:

```
Usage: nadi-gis [OPTIONS] <COMMAND>

Commands:
  nid      Download the National Inventory of Dams dataset
  usgs     Download data from USGS NHD+
  layers   Show list of layers in a GIS file
  check    Check the stream network to see outlet, branches, etc
  order    Order the streams, adds order attribute to each segment
  network  Find the network information from streams file between points
  help     Print this message or the help of the given subcommand(s)

Options:
  -q, --quiet  Don't print the stderr outputs
  -h, --help   Print help
```

The important functions are:

- Download NID and USGS NHD+ data,
- Check stream network for validity of DAG (Directed Acyclic Graph) required for NADI,
- Stream ordering for visual purposes,
- Network detection between points of interest using the stream network

You can use the help command for each one of the subcommand for more help. For example, `usgs` subcommand's help using `nadi-gis help usgs` gets us:

```
Download data from USGS NHD+

Usage: nadi-gis usgs [OPTIONS] --site-no <SITE_NO>

Options:
  -s, --site-no <SITE_NO>
```

```
        USGS Site number (separate by ',' for multiple)

-d, --data <DATA>
        Type of data (u/d/t/b/n)

        [upstream (u), downstream (d), tributaries (t), basin (b), nwis-site (n)]

        [default: b]

-u, --url
        Display the url and exit (no download)

-v, --verbose
        Display the progress

-o, --output-dir <OUTPUT_DIR>
        [default: .]

-h, --help
        Print help (see a summary with '-h')
```

8.2. NADI QGIS

The QGIS plugin for nadi has a subset of the CLI functionality. It can be accessed from the Processing Toolbox.

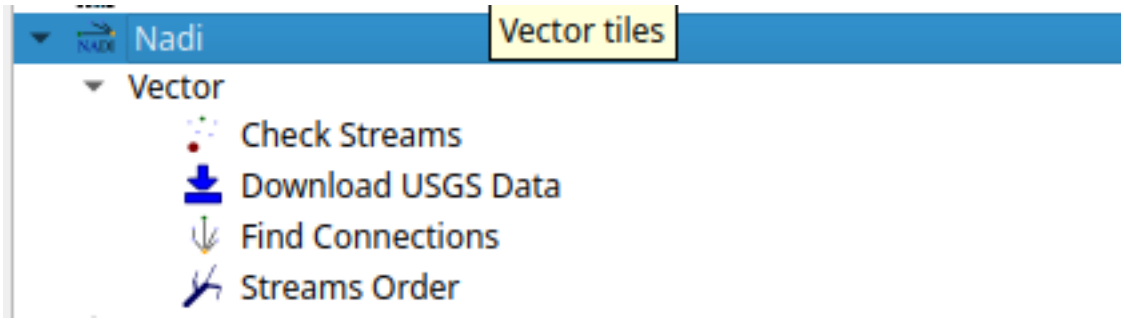


Figure 4: QGIS Processing Toolbox

You can run the tools from there and use the layers in QGIS as inputs. The QGIS plugin will first try to find `nadi-gis` binary on your `PATH` and use it, if not it'll try to use the binary provided with the plugins. It is preferred to have `nadi-gis` available in `PATH` and running without errors.

9. Example

The examples here will be given using QGIS plugin, and using the CLI tool both. CLI tool is great for quickly running things, and doing things in batch, while QGIS plugin will be better on visualization and manual fixes using other GIS tools.

If you want a video demostration, there is a Demo Video on YouTube.

9.1. Using QGIS Plugin

First **downloading the data** is done through the `Download USGS Data` tool. As shown in the screenshot below, input the USGS site ID and the data type you want to download.

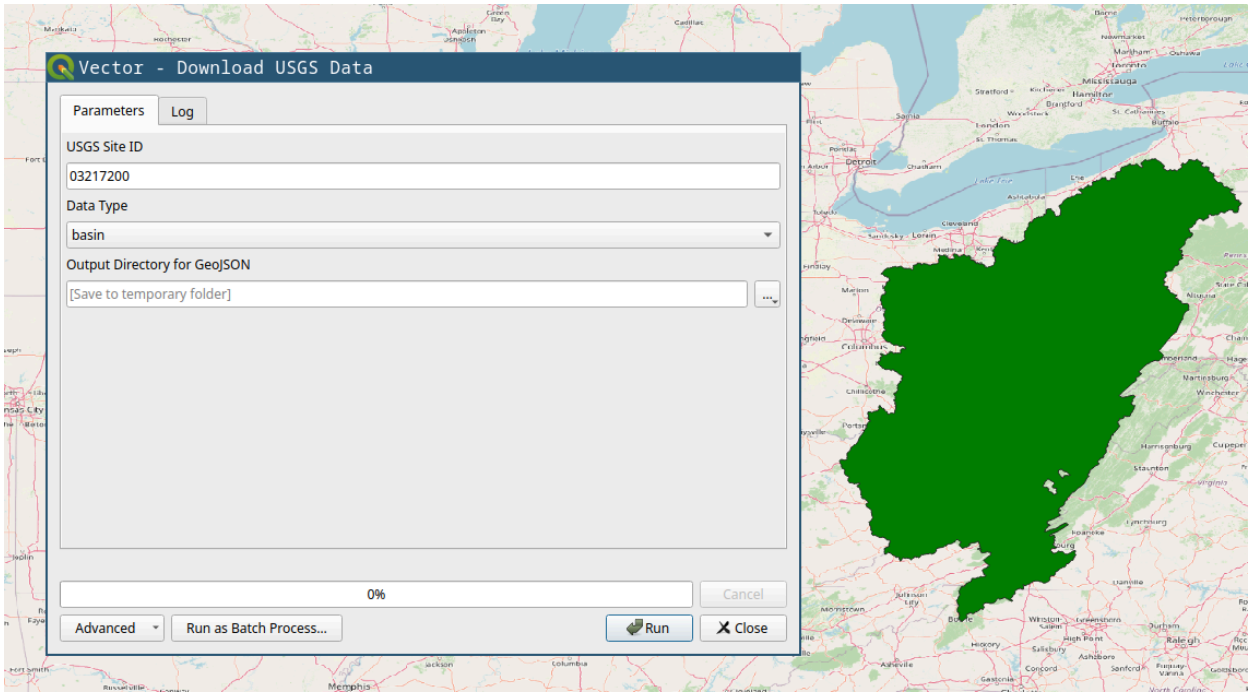


Figure 5: QGIS Download

You will need, `tributaries` for the upstream tributaries for network, and `nwis-site` will download the USGS NWIS sites upstream of the location. We will use those two for the example. If you have national data from other sources, you can use the basin polygon to crop them.

Stream Order tool is mostly for visual purposes. The figure below shows the results from stream order on right compared to the raw download on left.

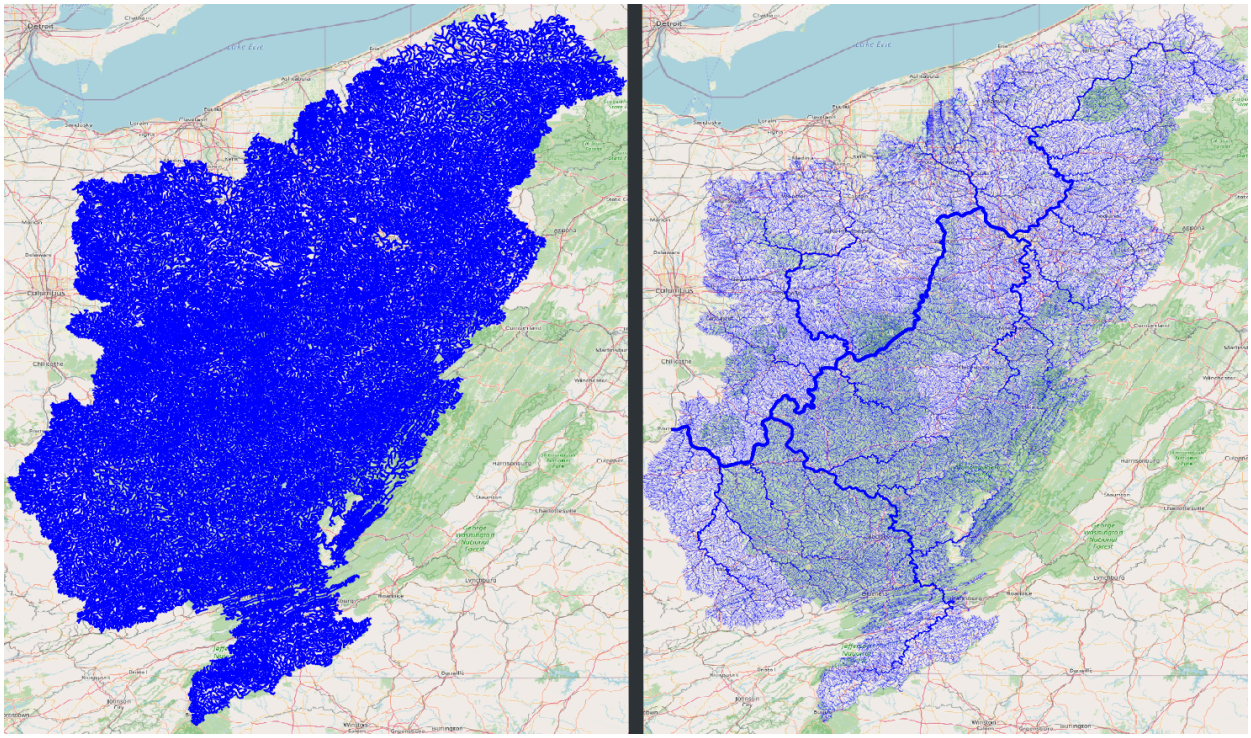


Figure 6: Stream Order Result

After you have streams (tributaries), you can use the **Check Streams** tool to see if there are any errors. It will give all the nodes and their categories, you can filter them to see if it has branches, or if it has more than one outlet. The figure below shows the branches with red dot. If we zoom in we can see how the bifurcation on the stream is detected, and how stream order calculation is confused there.

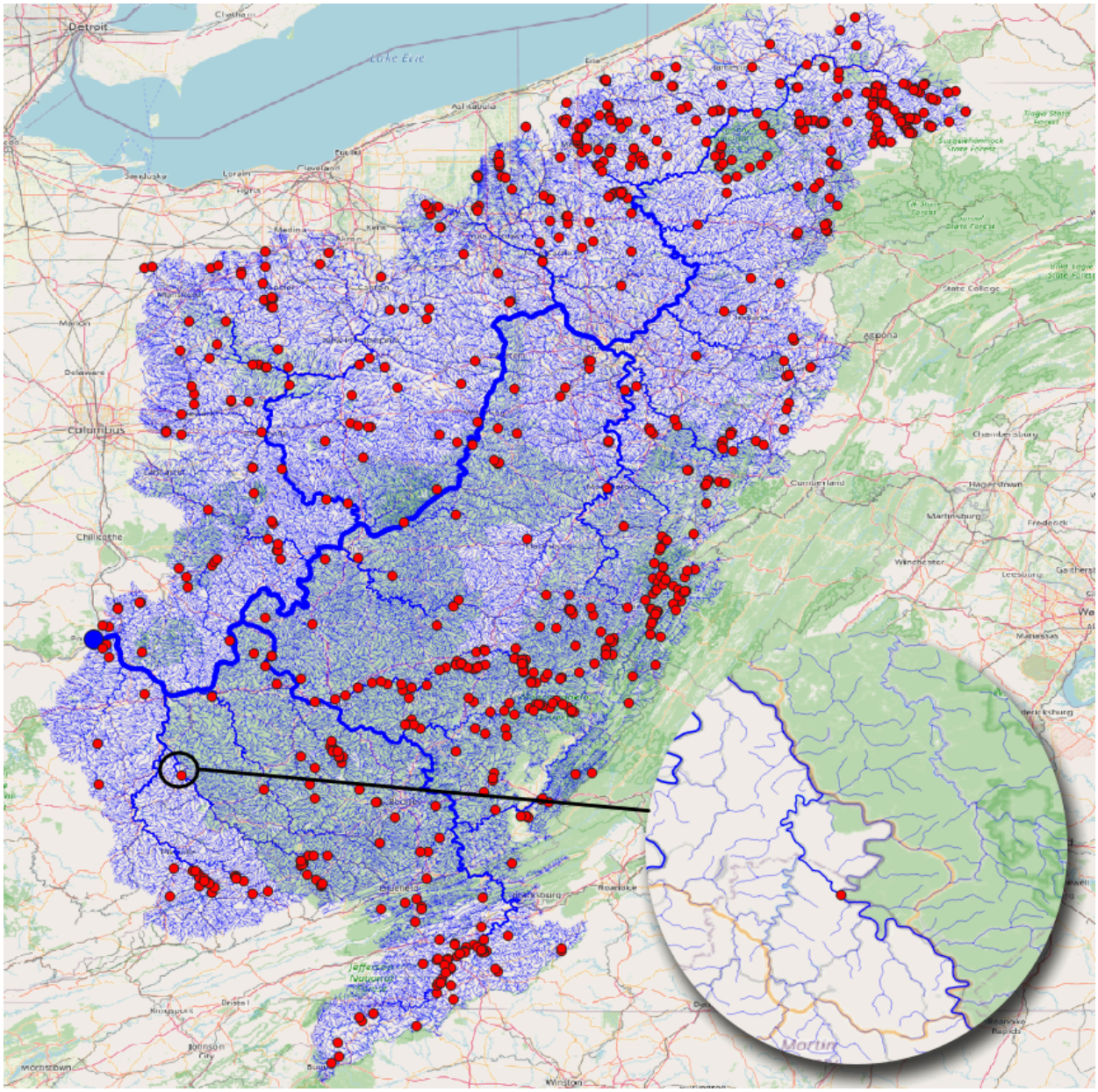


Figure 7: Check Streams Result

Find Connections tool will find the connection between the points using the stream network. The results below shows the tool being run on the NWIS points.

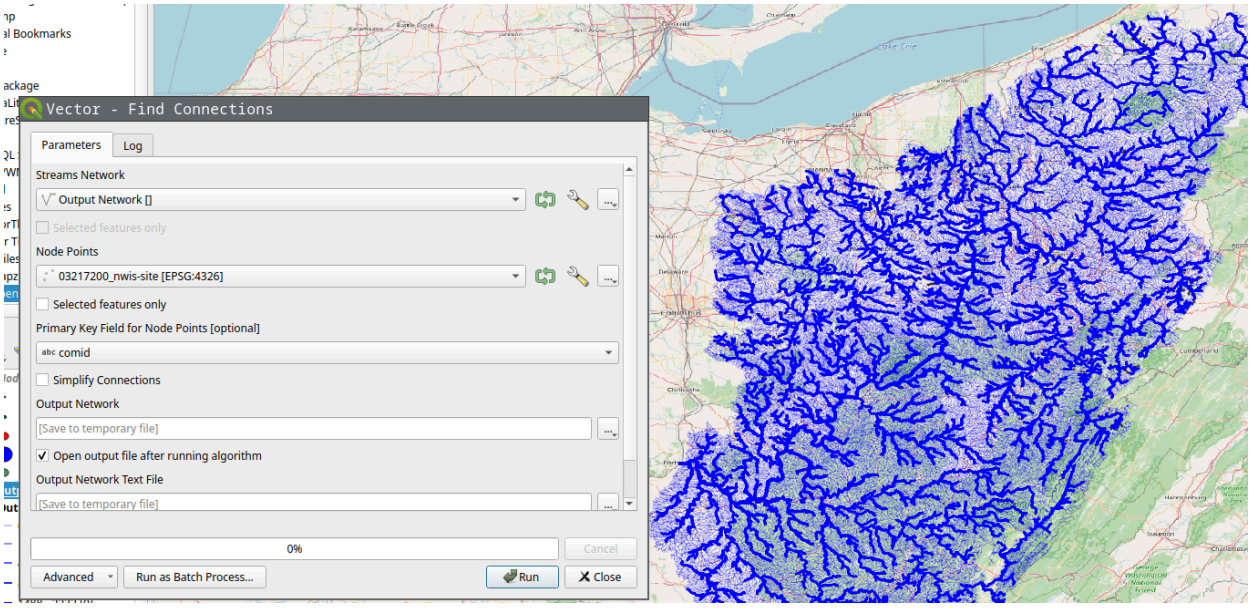


Figure 8: Find Connections Result

If we select **simplify** option, it'll only save the start and end point of the connection instead of the whole stream.

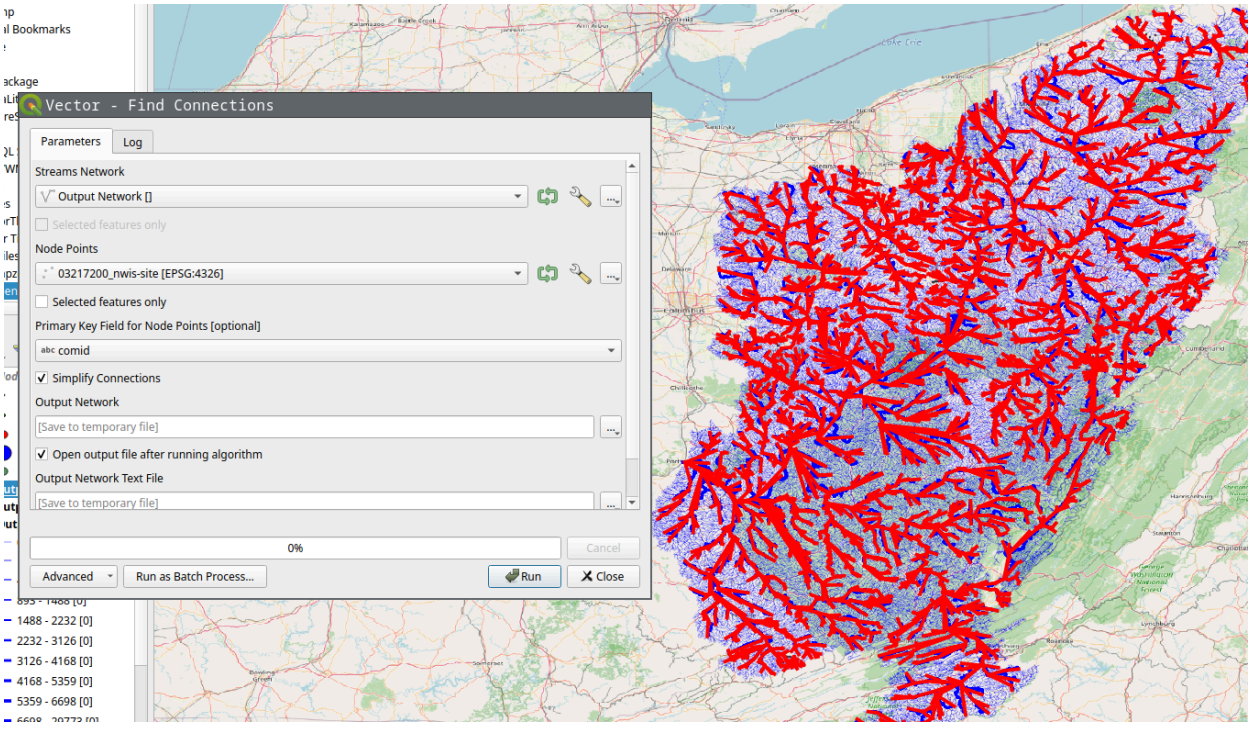


Figure 9: Find Connections Result Alt

Of course you can run **Stream Order** on the results to get a more aesthetically pleasing result.

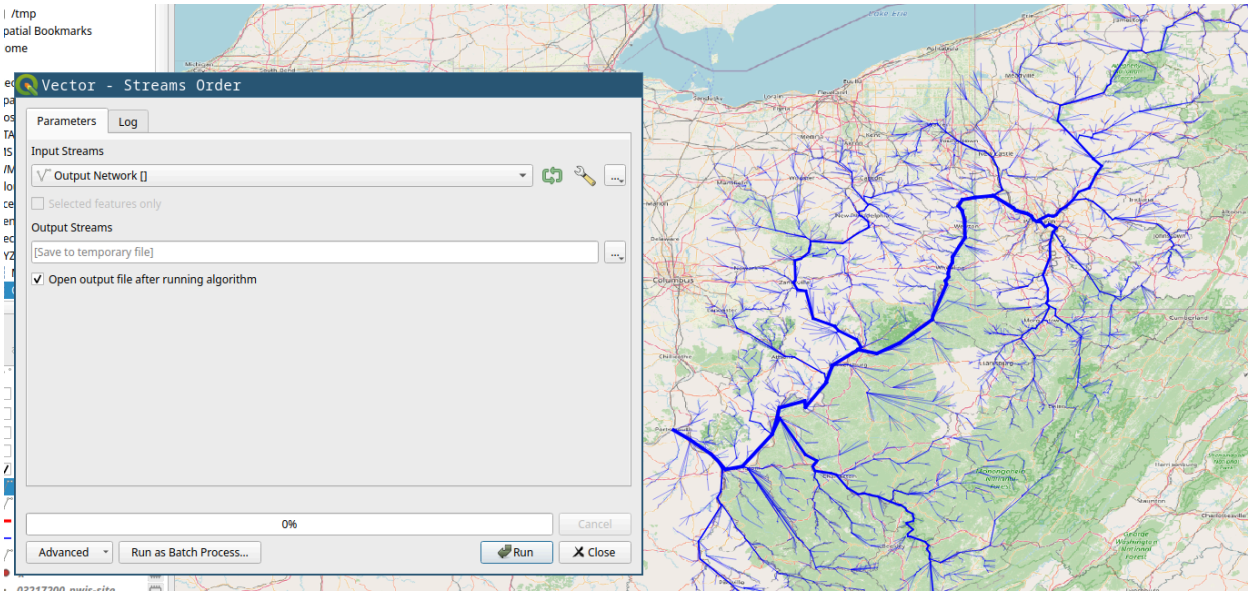


Figure 10: Find Connections Result with Order

9.2. Using CLI

An example of running `nadi-gis` using CLI can be done in the following steps:

9.2.1. Download data

We'll download the streamlines and the NWIS Sites from USGS for station 03217200 (Ohio River at Portsmouth, OH).

```
nadi-gis usgs -s 03217200 -d n -d t -o output/
```

This will download two files:

```
output/03217200_nwis-site.json  output/03217200_tributaries.json
```

Now we can use `check` command to see if there are any problems with the streams.

```
nadi-gis check output/03217200_tributaries.json
```

That gives us the following output:

```
Invalid Streams File: Branches (826)
* Outlet: 1
* Branch: 826
* Confluence: 30321
* Origin: 29591
```

We can generate a GIS file to locate the branches and see if those are significant. Refer to the `help` for `check` or use the QGIS plugin.

And to find the connections, we use `network` subcommand like this:

```
nadi-gis network -i output/03217200_nwis-site.json output/03217200_tributaries.json
```

Output:

```
Outlet: 3221 (-82.996916801, 38.727624498) -> None
3847 -> 3199
2656 -> 2644
399 -> 1212
2965 -> 3942
2817 -> 6236
5708 -> 4733
2631 -> 5741
201 -> 2101
2066 -> 2317
3770 -> 1045
... and so on
```

Since this is not as useful, we can use the flags in the `network` subcommand to use a different id, and save the results to a network file.

First we can use `layers` subcommand to see the available fields in the file:

```
nadi-gis layers output/03217200_nwis-site.json -a
```

which gives us:

```
03217200_nwis-site
  - Fields:
```

```
    + "type" (String)
    + "source" (String)
    + "sourceName" (String)
    + "identifier" (String)
    + "name" (String)
    + "uri" (String)
    + "comid" (String)
    + "reachcode" (String)
    + "measure" (String)
    + "navigation" (String)
```

Using `comid` as the id for points, and saving the results:

```
nadi-gis network -i output/03217200_nwis-site.json output/03217200_tributaries.json
-p comid -o output/03217200.network
```

The `output/03217200.network` file will have the connections like:

```
15410797 -> 15411587
6889212 -> 6890126
8980342 -> 10220188
19440469 -> 19442989
19390000 -> 19389366
6929652 -> 6929644
... and so on
```

Make sure you use a field with unique name, and valid identifier in NADI System.

Network Analysis (DSL)

10. Getting Started

10.1. Getting Started

This section walks you throuh the process of using NADI system through the CLI and IDE.

Network analysis is done through the Domain Specific Programming Language. This is the main feature of NADI.

You can run the DSL, called tasks through the CLI, or through the IDE. Refer to the Installation for how to install them.

Recommended way to develop/write nadi tasks is with IDE, while CLI can be used to automate/batch run it once the tasks are finalized.

10.1.1. Command Line Interface (CLI)

NADI CLI is available as `nadi` command when you have it installed. You can run it in interactive mode in a Read Eval Print Loop (REPL), or provide a file to run.

So assuming you have the following contents in `sample.tasks` file:

```
network load_str("a -> b")
node array(LEVEL, ORDER)
```

You run it with:

```
nadi sample.tasks
```

You should get the following output:

```
{
  b = [0, 2],
  a = [0, 1]
}
```

If you run nadi with `--repl` or `-r` flag, then it'll open a REPL. If you have a tasks file provided like before, it'll run the tasks in the file before entering a REPL, otherwise it'll start from empty context.

For more details on other use of `nadi` command. Refer to the output of `nadi --help`. It has options for,

- Evalute some tasks before running the file/repl for setting context,
- Print/Generate help for functions,
- Inspect code for functions,
- Autocomplete nadi functions,
- Generate a template for nadi plugin code, and more.

10.1.2. Integrated Development Environment (IDE)

NADI IDE contains multiple components to make it easier to write, edit, and run tasks, as well as visualize the network and browse the function documentation.

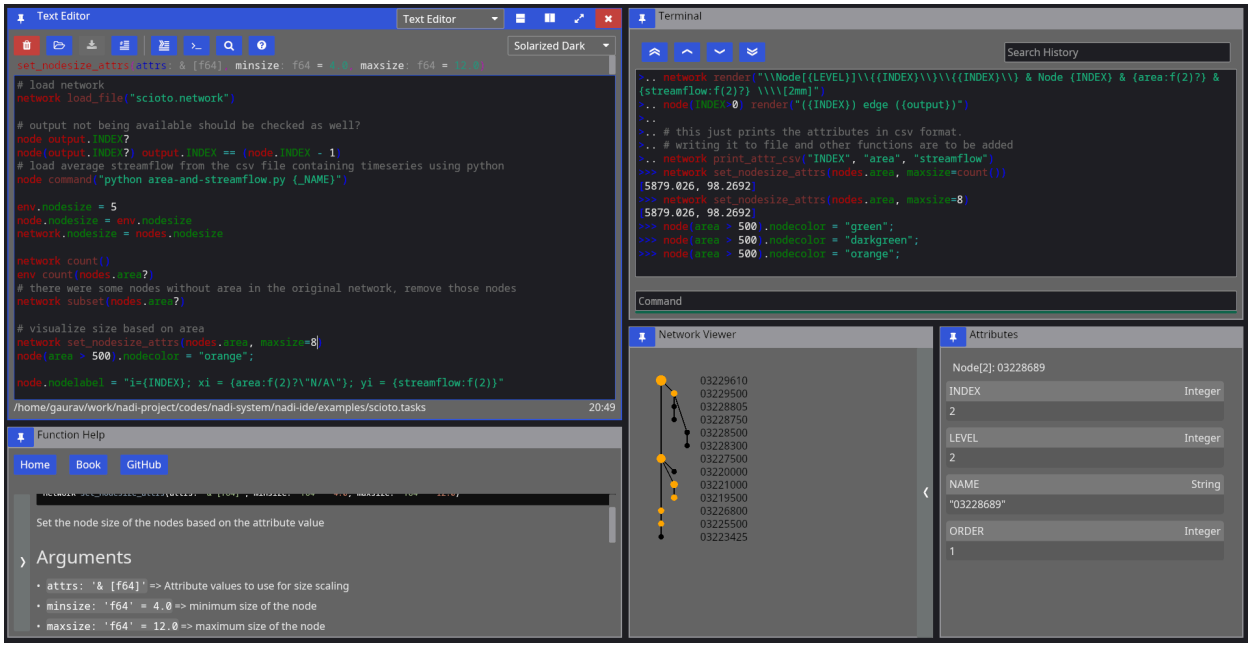


Figure 11: IDE

As you can see, the main UI is divided into multiple panes, which are independent components that you can resize, arrange the way you want.

First, when you start NADI IDE, you come up with this view:

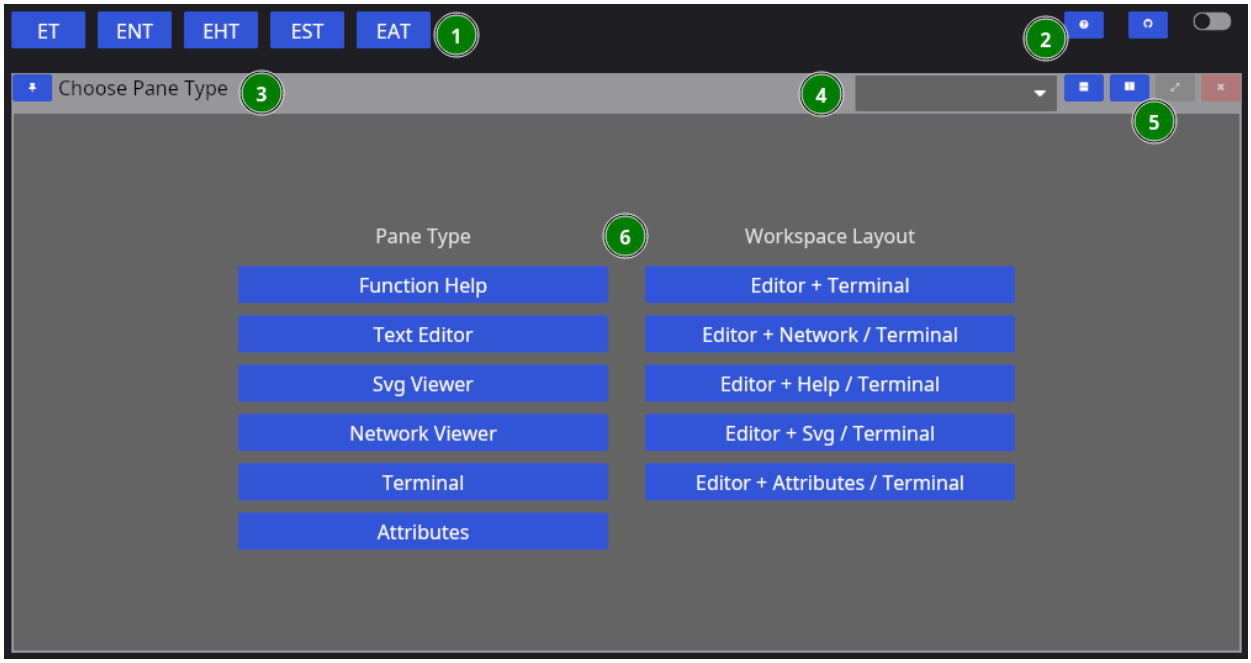


Figure 12: IDE Greeting

1. Different layout options
2. Global options
3. Pane Title Bar
4. Dropdown to change the Pane Type
5. Pane options (horizontal/vertical split, fullscreen, close)
6. Pane Contents

The layout of each pane are similar, while each one will have their own contents. For example, “Function Help” contains the help for the plugin functions. We will go over each one in a separate chapter.

You can choose a singular pane type here, or chose a combination of panes that feels more useful. Some panes like terminal/attribute views also spawn automatically (if not already in view) when you try to run tasks, or click nodes.

Now, let’s choose “Text Editor” for now, then we’re greeted with this view:

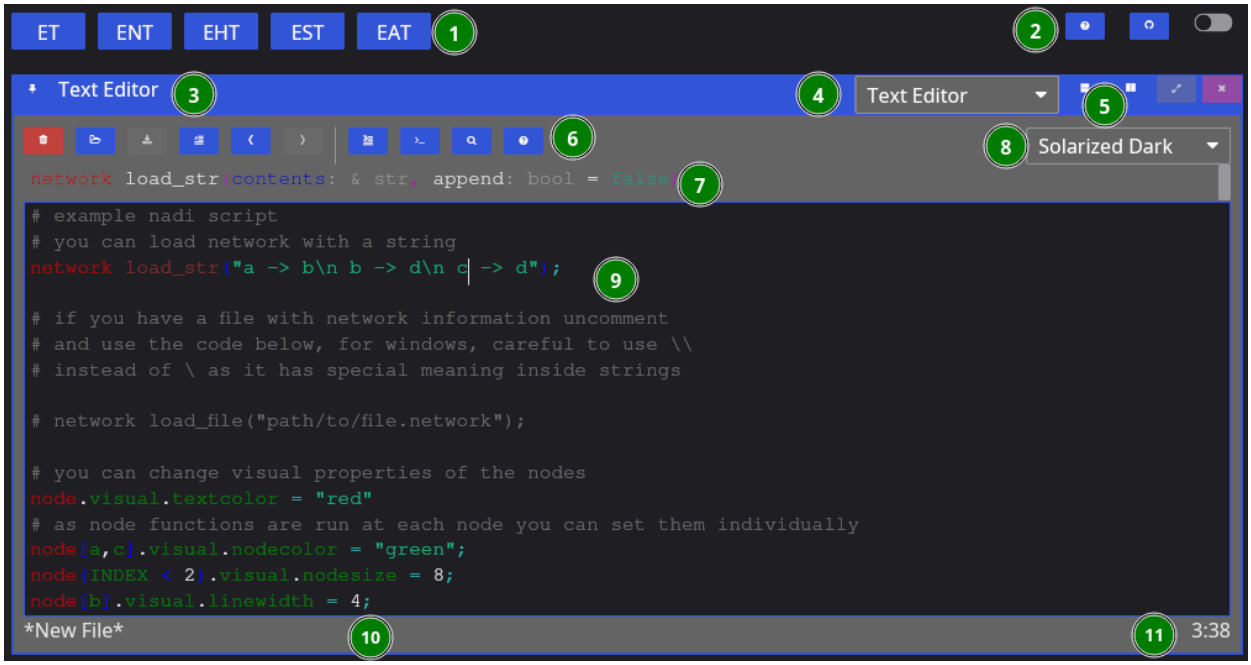


Figure 13: IDE Editor

Now, aside from the global components, we have:

6. Editor Tools,
7. Function Signature: Only shows when cursor is on a function,
8. Editor Theme: Only works for non-NADI formats like Python, C, Rust, R, etc.
9. Tasks File Contents: File contents with syntax highlighting,
10. Current File Path: Where the file is opened from and will be saved, and
11. Line/Column of Cursor.

As for the editor tools: you can hover over them to know their names, and shortcuts. We'll explain them shortly below in the left to right order,

Button	**Key**	**Function**
New File	`Ctrl` + `n`	Remove the contents, and the filepath for new file
Open	`Ctrl` + `o`	Browse and open a new file
Save	`Ctrl` + `s`	Save the file, browse for new file if path is not given
Toggle Comment	`Alt` + `;`	Comment or uncomment the current selection
Undo	`Ctrl` + `z`	Undo the last edit (edits are saved periodically)

Button	Key	Function
Redo	`Ctrl` + `y`	Redo the last undo (redo vanishes if you edit after undo)
Run Line/Selection	`Ctrl` + `Enter`	Run the current line, or the selection in terminal
Run Buffer	`Ctrl` + `Shift` + `Enter`	Run the whole buffer (file contents) in terminal
Search		Search the selection in function help
Help		Visit the function help for current function

When yo run a line using “Run Line/Selection” it should spawn a terminal pane on the right side

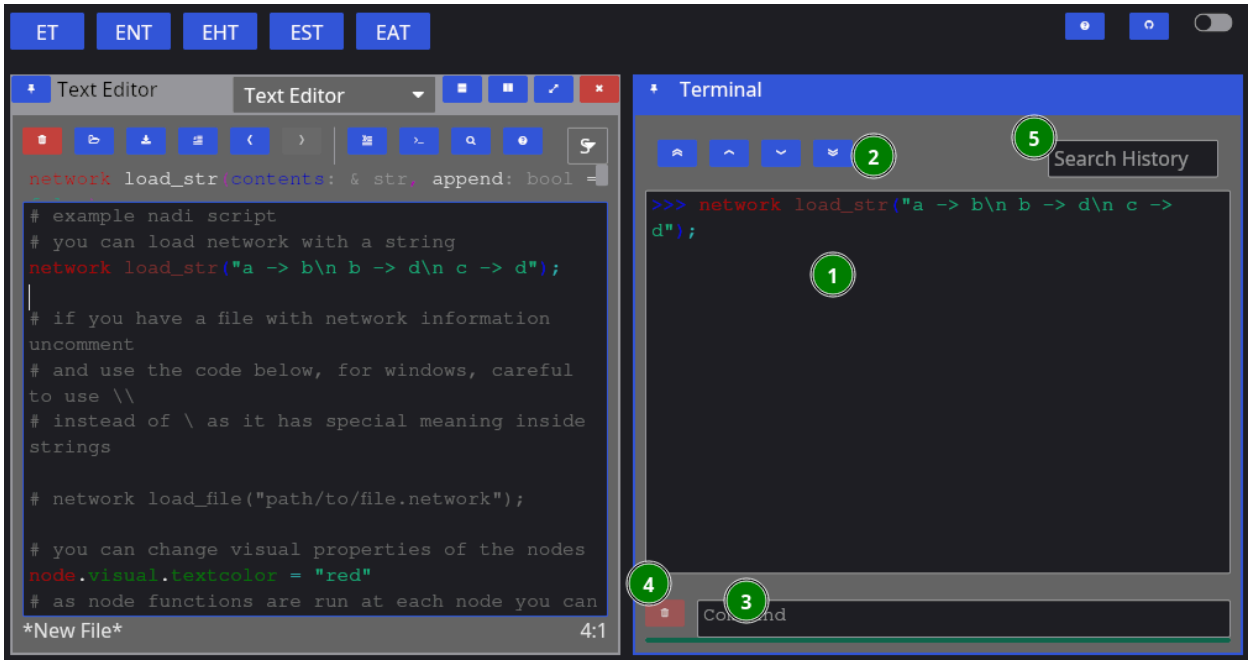


Figure 14: IDE Terminal

Here we have:

1. Terminal Contents: tasks that were run, and their output,
2. Terminal Navigation: Buttons to goto top/bottom or page up/down,
3. Command Entry: You can directly enter commands here instead of from editor (they won’t be saved),
4. Clear: Clear the current command entry,
5. History: Search here for previous commands, clicking them will copy it to the Command Entry

Upto here will be enough to run tasks. But the advantage of the IDE with a Graphical User Interface (GUI) comes in the form of network visualization. If you open the Network Viewer pane, either through the dropdown to change one of the pane. Or split the panes and open Network Viewer there, you can see the network visualized.

The figure below shows the network viewer, and the attribute viewer (it spawns when you click on a node in the network) on the bottom right.

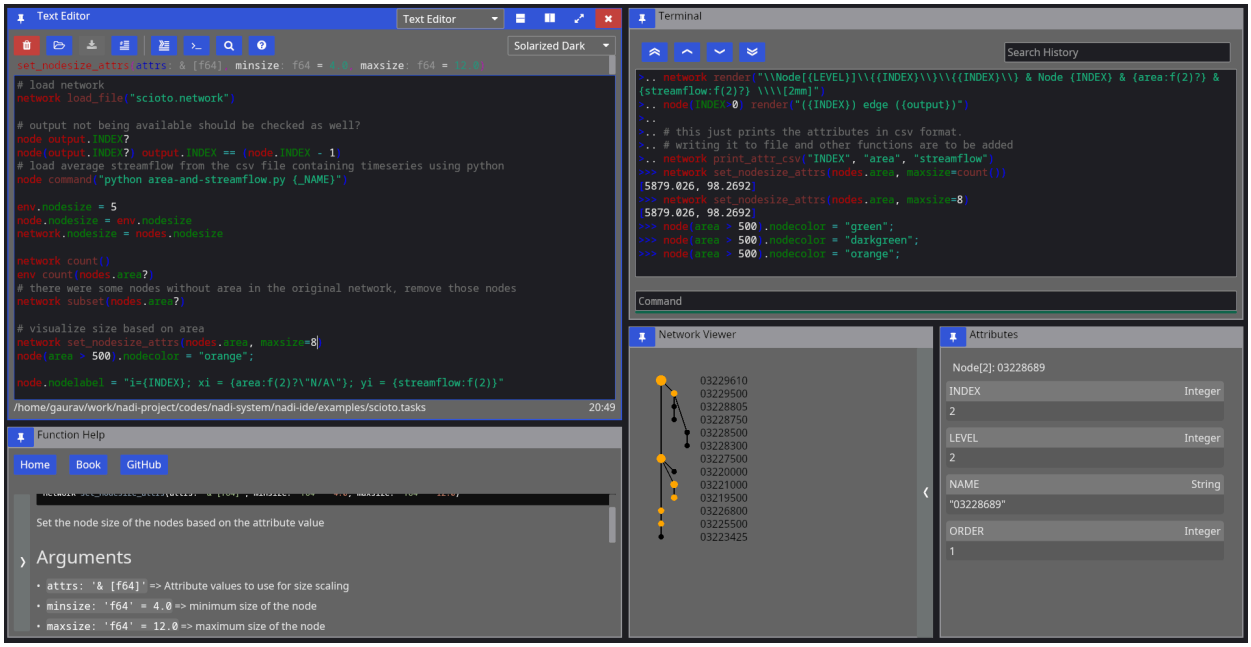


Figure 15: IDE

You can click on the "<" button on the right side of the network viewer to expand the sidebar that has details on how to change the visual attributes of the nodes. You can run the examples in the code for testing them out as well.

Extra components of the IDE includes the Function Help that you can use to browse the plugin functions' documentation, and SVG viewer that you can use to open SVG files. The SVG viewer is very primitive and only meant to be a quick check/update while you're fine tuning an exported SVG file, use dedicated softwares and web browsers for actual viewing.

The function help looks like below:

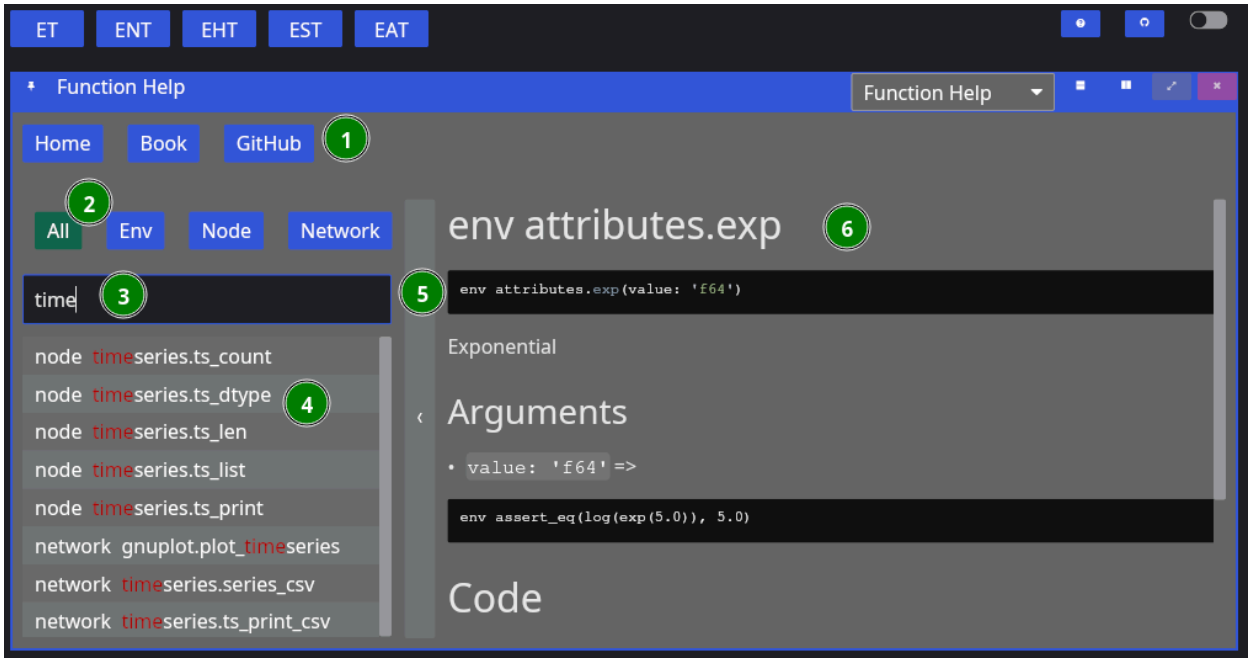


Figure 16: IDE Help

The figure contains,

1. Buttons for Reset, and links to NADI Book and Github,
2. Buttons to filter search by function type,
3. Text Entry to search the function,
4. List of plugin functions,
5. Bar to collapse/expand the function search sidebar, and
6. Plugin function documentation content.

Currently you cannot copy the code or any contents from the documentation, it is a limitation of the GUI library that might be fixed in the future. If you want to copy the text, refer to the web version of the function help (in NADI Book). Or use the `help [type] <function>` task. For example:

```
help node render
```

Results:

```
node render (template: '& Template', safe: 'bool' = false)
Render the template based on the node attributes
# Arguments
- `template: & Template` String template to render
```

```
- `safe: bool` [def = false] if render fails keep it as it is instead of exiting


For more details on the template system. Refer to the String
Template section of the NADI book.

```task
network load_str("a -> b")
node.x = 13
node assert_eq(render("abc {x}"), "abc 13")
```
```

You can copy the example code from the output in the terminal.

That should be enough for you to be able to use the GUI, in the next section we'll talk about the core concepts in NADI system that you need to understand, and then we'll show some examples you can run using what you learned in this chapter.

11. Core Concepts

This section contains a brief explanation of core concepts.

The main concepts that you need to know are:

- **Attributes** are values, it can be float, integer, boolean, strings, or list of attributes, or a map of attributes (key=value),
- **Nodes** are points in the network, they can have attributes, input nodes and an output node,
- **Network** is a collection of nodes, network can also have attributes, Network used in the NADI System can have only one outlet, so a 'ROOT' node is added if there are multiple outlet. And loading a network that is not a directed tree is undefined behaviour.
- **Expression** is something that can be evaluated or executed, it consists of literal values (attributes), variables (node, network, env variables that could hold attributes), function calls, or a mathmatical or logical operation.
- **Functions** in nadi are of 3 types, env functions are normal functions that take values and run, network functions take values and run on the network, while node functions run at each node (they also provide a way to subset which nodes to run it on).

- **Task** is an execution body of the task system. It can be of env, network or node type. It can be conditional (If-Else) or loop (While) consisting of more tasks inside it. Task can assign values to the env/network/node attributes, or call mutable functions on the top level.

- **String Template**: Some functions take string inputs that are interpreted dynamically to represent different strings based on variables.

- **Plugins** provide the functions used by the nadi task system. There are internal plugins and external plugins. Internal plugins comes with the installation, while external plugins are loaded from dynamic libraries.

11.1. Keywords

Keyword	Description
node	the node task type, function or variable
network/net	the network task type, function or variable
env	the environment task type, function or variable
exit	exit the program
end	end the execution of tasks without exiting
help	display help for functions
inputs	get node variables or function output for input nodes of a node
output	get node variable or function output for output node of a node
nodes	get node variable or function output for all nodes in the network
if	if statement for conditional task/expression
else	else statement for conditional task/expression
while	while statement for loop task
in	binary operator to check if something is in another (list/string)
match	binary operator to check patterns on string (regex)

And here are some keywords reserved for future:

Keyword	Description
function/func	user defined functions
map	map values in an array/attrmap to a function
attrs	attributes of the env/node/network
loop	loop task
for	for loop task for looping through array/attrmap

11.2. Symbols

Some special symbols and their functions are listed below:

- `.` dot accessor for variables, e.g. `node.var`, `node.var.another`, etc.,
- `?` variable check (only used after a variable) e.g. `node.var?` evaluates to a `true` or `false`,
- `()`, `{}`, `[]` brackets for functions/expressions, attrmaps and arrays,
- `;` suppress current task output (only used at the end of a task),
- `+`, `-`, `*`, `/`, and `//` are mathematical operators,
- `&`, `|`, and `!` are logical operators,
- `>`, `<`, `>=`, `<=`, and `==` are comparision operators,
- `->` path operator (only used in node propagation, or network),
- `=` is assignment operator,
- `#` starts a comment,

There might be more functions of each symbol depending on the context.

Continue with the chapters for details on each concept. Or skip ahead to Learn by Examples if you want to jump into the examples.

11.3. Task

11.3.1. Task

Task is an execution body in the task system. There are different types of tasks, specially environment, network and node type tasks, and there can be conditional tasks that only execute based on a condition or loops.

Some examples of different tasks are given below to show a general overview, but the concepts inside the tasks system will be introduced as we progress through the chapters,

Environment tasks that can evaluate expressions, assign variables, or call functions:

```
env 1 + 2 * 8
env render("my name is {_name}", name="John")
env.x = 12 > 2;
env.x
```

Results:

```
17
"my name is John"
true
```

network task loading a network, and node task getting node attributes:

```
network load_str("a->b\nb->c")
node.NAME
```

Results:

```
{
  c = "c",
  b = "b",
  a = "a"
}
```

Conditional and Loop task

```
if ( !val? | (val > 5) ) {
  # if val is not defined or greater than 5, set it to 0
    env.val = 0
}
while (val < 5) {
  env.val = env.val + 1
}
```

Results:

```
0 -> 1
1 -> 2
2 -> 3
3 -> 4
4 -> 5
```

Tasks system acts like a scripting language for nadi system. A Task consists of getting/evaluating/setting attributes in environment, network or nodes. The value that can be evaluated are expressions that consists of literal values, variables, or function calls that can either be a

environment, node or a network function. Functions are unique based on their names, and can have default values if users do not pass all arguments.

The code examples throughout this book, that are being used to generate network diagrams, tables, etc are run using the task system.

Here is an example contents of a more complex task file, do not concern with what each task does, we will go through them in other chapters.

```
# sample .tasks file which is like a script with functions
node<inputsfirst> print_attrs("uniqueID")
node show_node()
network save_graphviz("/tmp/test.gv")
node<inputsfirst>.cum_val = node.val + sum(inputs.cum_val);

node[WV04113,WV04112,WV04112] print_attr_toml("testattr2")
node render("{NAME} {uniqueID} {_Dam_Height_(Ft)?}")
node list_attr("; ")
# some functions can take variable number of inputs
network calc_attr_errors(
    "Dam_Height_(Ft)",
    "Hydraulic_Height_(Ft)",
    "rmse", "nse", "abserr"
)
node sum_safe("Latitude")
node<inputsfirst> render("Hi {SUM_ATTR}")
# multiple line for function arguments
network save_table(
  "test.table",
  "/tmp/test.tex",
  true,
  radius=0.2,
  start = 2012-19-20,
  end = 2012-19-23 12:04
  )
node.testattr = 2
node set_attrs_render(testattr2 = "{testattr:calc(+2)}")
node[WV04112] render("{testattr} {testattr2}")

# here we use a complicated template that can do basic logic handling
node set_attrs_render(
    testattr2 = "=(if (and (st+has 'Latitude) (> (st+num 'Latitude) 39)) 'true
'false)"
)
# same thing can be done if you need more flexibility in variable names
```

```
node load_toml_string(
    "testattr2 = =(if (and (st+has 'Latitude) (> (st+num 'Latitude) 39)) 'true
'false)"
)
# selecting a list of nodes to run a function
node[
  # comment here?
    WV04113,
    WV04112
] print_attr_toml("testattr2")
# selecting a path
node[WV04112 -> WV04113] render("=(> 2 3)")
```

11.4. Attributes

Attributes are TOML like values. They can be one of the following types:

Type Name	Rust Type	Description
Bool	`bool`	Boolean values (`true` or `false`)
String	`RString`	Quoted String Values
Integer	`i64`	Integer values (numbers)
Float	`f64`	Float values (numbers with decimals)
Date	`Date`	Date (`yyyy-mm-dd` formatted)
Time	`Time`	Time (`HH:MM`, `HH:MM:SS` formatted)
DateTime	`DateTime`	Date and Time separed by or `T`
Array	`RVec<Attribute>`	List of any attribute values
Table/AttrMap	`AttrMap`	Key Value pairs of any attribute values

You can write attributes directly into the task system to assign them, use them in functions. You can also load attributes from a file into the env/node/network.

If you want to assign a attribute inside the task system, you can do it like this:

```
env.river = "Ohio River"
env.river
```

Results:

```
"Ohio River"
```

Example Attribute File that can be loaded:

```
river = "Ohio River"
outlet = "Smithland Lock and Dam"
outlet_is_gage = true
outlet_site_no = ""
streamflow_start = 1930-06-07
mean_streamflow = 123456.0
obs_7q10 = 19405.3
nat_7q10 = 12335.9
num_dams_gages = 2348
```

Here loading the files we can see only ohio has the attributes loaded

```
 network load_file("./data/mississippi.net")
node[ohio] load_attrs("./data/attrs/{_NAME}.toml")
node.outlet
```

Results:

```
{
  lower-mississippi = <None>,
  upper-mississippi = <None>,
  missouri = <None>,
  arkansas = <None>,
  red = <None>,
  ohio = "Smithland Lock and Dam",
  tenessee = <None>
}
```

With plugins, you can load attributes from different file types.

11.5. Node

A Node is a point in network. A Node can have multiple input nodes and only one output node. And a Node can also have multiple attributes identifiable with their unique name, along with timeseries values also identifiable with their names.

If you understand graph theory, then node in nadi network is the same as a node in a graph.

Nodes in NADI Network are identified by their name, that is loaded from the network file. Node names are string values, even if they are integer or float, they are read and internally stored as strings. If the node name contains characters outside of alphanumeric and underscore (_), it has to be quoted.

i.e. valid names like `123` or `node_1` can appear unquoted or quoted, but names like `node-123` needs to be quoted: `"node-123"`.

```
network load_str("
123 -> node_1
node_1 -> \"node-123\"
")
node.NAME
```

Results:

```
{
  node-123 = "node-123",
  node_1 = "node_1",
  123 = "123"
}
```

If the node name `node-123` appearing on the output is unquoted, it's a bug that will be fixed in the next version. It is harmless as long as you don't try to copy the output as a valid attribute value to load back into the system.

If you do not quote the name, you'll get an error:

```
network load_str("123 -> node-1")
node.NAME
```

Error:

```
Error in function load_str: Error: Parse Error at Line 1 Column 8
  123 -> node-1
         ^ Incomplete Path; expected node here
```

11.6. Network

11.6.1. Network

A Network is a collection of nodes. The network can also have attributes associated with it. The connection information is stored within the nodes itself. But Network will have nodes ordered based on their connection information. So that when you loop from node from first to last, you will always find output node before its input nodes.

A condition a nadi network is that it can only be a directed graph with tree structure.

Example Network file:

```
# network consists of edges where input node goes to output node
# each line is of the format: input -> output
tenessee -> ohio
# if your node name has characters outside of a-zA-Z_, you need to
# quote them as strings
ohio -> "lower-mississippi"
"upper-mississippi" -> "lower-mississippi"
missouri -> "lower-mississippi"
arkansas -> "lower-mississippi"
red -> "lower-mississippi"
```

The given network can be loaded and visualized using `svg_save` function.

```
network load_file("./data/mississippi.net")
network command("mkdir -p output")
network svg_save(
   "./output/network-mississippi.svg",
  label="[{INDEX}] {_NAME:repl(-, ):case(title)}",
  bgcolor="gray"
)
```

Results:

Figure 17:

You can assign different graphical properties through node properties.

```
network load_file("./data/mississippi.net")
node[red].visual.nodecolor = "red";
node[ohio].visual.linecolor = "blue";
node[ohio].visual.linewidth = 3;
node["upper-mississippi", red].visual.nodesize = 8;
node[red].visual.nodeshape = "triangle";
node["upper-mississippi"].visual.nodeshape = "ellipse:0.5";
network svg_save(
   "./output/network-mississippi-colors.svg",
   label="[{INDEX}] {_NAME:repl(-, ):case(title)}",
  bgcolor="gray"
)
```

Results:

Figure 18:

11.7. Expression

Expressions are airthmetic or logical operations. They can appear inside the conditional statements, or as input to a task, or nested in other expression or function calls.

Expressions are defined into the following categories:

11.7.1. Literal Values

```
env [1, true, "no maybe"]
```

Results:

```
[1, true, "no maybe"]
```

11.7.2. Variable

```
env.value = [1, true, "no maybe"];
env.value
```

Results:

```
[1, true, "no maybe"]
```

Variables also have a “check” mode, where it returns true if variable exists, false if it does not.

```
env.value = [1, true, "no maybe"];
env value?
env other_var?
```

Results:

```
true
false
```

You can also use use varible from node, or network in other context. For example:

```
env.value = [1, true, "no maybe"];
network echo(json(env.value))
```

Results:

```
[1, true, "no maybe"]
```

Special variable types like `nodes`, `inputs`, `output` are available besides `env`, `network` and `node` based on what type of task the expression is on.

You will learn more about this on Cross Context Functions and Variables chapter.

11.7.3. Unary Operator

```
env !true
env - 12.0
```

Results:

```
false
-12
```

11.7.4. Binary Operator

```
env (12 > 34) & true
env "x" in "xyz"
env 12 in [123, true]
env "my name is" match "^my.*"
```

Results:

```
false
true
false
true
```

11.7.5. If Else

```
env if(!true) {"if true"} else {"if false"}
```

Results:

```
"if false"
```

11.7.6. Function

```
env.value = [1, true, "no maybe"];
env get(value, 2)
```

Results:

```
"no maybe"
```

Out of all expressions, only the function is not garanteed to return a value. If you are using a function expression and expect a value and it does not return it, it'll be a runtime error.

```
env echo("Hello world!") + 12
```

Error:

```
Hello world!

Function echo did not return a value
```

Special function types like `nodes`, `inputs`, `output` are available besides `env`, `network` and `node` based on what type of task the expression is on.

You will learn more about this on Cross Context Functions and Variables chapter.

11.8. String Template

String templates are strings with dynamic components that can be rendered for each node based on the node attributes.

A simple template can be like below:

```
Hi, my name is {name}, my address is {address?"N/A"}.
I wrote this document on {%A}, exact date: {%Y-%m-%d}.
```

Results (with: name=John; address=123 Road, USA):

```
Hi, my name is John, my address is 123 Road, USA.
I wrote this document on Wednesday, exact date: 2025-08-06.
```

With more complicated templates, we would be able to generate documents with text and images based on the node attributes as well.

For example the following template can be used to generate a table.

```
| Name              | Index   |
|-------------------|---------|
<!-- ---8<--- -->
| {_NAME:case(up)} | {INDEX} |
<!-- ---8<--- -->
```

```
network load_file("./data/mississippi.net");
network echo(render_template("./data/example.template"))
```

Results:

Name	Index
LOWER-MISSISSIPPI	0
UPPER-MISSISSIPPI	1
MISSOURI	2
ARKANSAS	3
RED	4
OHIO	5
TENESSEE	6

Of course, there are better ways to generate table than this, but this shows how flexible the template system is.

11.9. Node Function

11.9.1. Node Function

Node function runs on each node. It takes arguments and keyword arguments.

For example following node function takes multiple attribute names and prints them. The signature of the node function is `print_attrs(*args)`.

```
network load_file("./data/mississippi.net")
node print_attrs("INDEX", name=false)
```

Results:

```
INDEX = 0
INDEX = 1
INDEX = 2
INDEX = 3
INDEX = 4
INDEX = 5
INDEX = 6
```

Only the `NAME` is printed as they do not have any other attributes.

11.9.2. Node Function Propagation

Propagation of the node functions refer to the order and selection of nodes to run the function on.

By default node function is run at each node. But that might not be the intended use of the function, for example you might want to:

- run the function only on nodes that satisfy a condition,
- run the funciton only on a group of nodes,

Or you might want to change the order of the node function execution. Like running input nodes before the output node, if your function/analysis needs that.

These things are done with 3 syntax in the node function:

11.9.2.1. Order

You can run functions in different orders:

- `sequential`/`seq` => based on node index
- `inverse`/`inv` => inverse based on node index
- `inputsfirst`/`inp` => input nodes before output
- `outputfirst`/`out` => output node before inputs

```
network load_str("a -> b\n b ->d \n c -> d \n d -> e")
node<seq> array(INDEX, ORDER)
node<inv> array(INDEX, ORDER)
```

Results:

```
{
  e = [0, 5],
  d = [1, 4],
  c = [2, 1],
  b = [3, 2],
  a = [4, 1]
}
{
  a = [4, 1],
  b = [3, 2],
```

```
  c = [2, 1],
  d = [1, 4],
  e = [0, 5]
}
```

Currently, `inp` and `inv` are equivalent, while `seq` and `out` are also equivalent. But when the parallization is added in the future versions, they will be implemented differently. So for backward compatibility, if you function needs to be run in a certain way, always use that one.

Here an example showing how to calculate the order of the node.

```
network load_str("a -> b\n b ->d \n c -> d \n d -> e")
node<inp>.val = sum(inputs.val) + 1;
node array(val, ORDER)
```

Results:

```
{
  e = [5, 5],
  d = [4, 4],
  c = [1, 1],
  b = [2, 2],
  a = [1, 1]
}
```

If you do not use the `inputsfirst` propagation here, you get an error because the `val` attribute doesn't exist in `inputs`, and if you did have that variable already, it would be old data instead of the recent results from your expression.

```
network load_str("a -> b\n b ->d \n c -> d \n d -> e")
node.val = sum(inputs.val) + 1;
node array(val, ORDER)
```

Error:

```
Attribute not found
```

NOTE: I need to work on better error messaging. It is in TODO list for the next major release.

11.9.2.2. Node List/Path

You can selectively run only a few nodes based on a list, or a path.

Given this network:

Figure 19: Network Diagram

11.9.2.2.1. List of Nodes

List of node contains a separated list of node names or quoted string if the name is not valid identifier/number inside `[]`.

```
network load_file("./data/mississippi.net")
node[tenessee,"lower-mississippi"] print_attrs("NAME")
```

Results:

```
NAME = "lower-mississippi"
NAME = "tenessee"
```

11.9.2.2.2. Path of Nodes

Path of node has the same syntax as a path used in the network file. It has starting node and end node. Instead of it representing a single edge like in network file, it represents all the nodes that are between those two (inclusive).

```
network load_file("./data/mississippi.net")
node[tenessee -> "lower-mississippi"] print_attrs("NAME")
```

Results:

```
NAME = "tenessee"
NAME = "ohio"
NAME = "lower-mississippi"
```

As we can see in the diagram, the path from tenessee to lower mississippi includes the ohio node.

11.9.2.3. Logical Condition

Logical condition is used by putting an expression that evaluates to a boolean value inside the `()`.

```
network load_file("./data/mississippi.net")
node("mississippi" in NAME) print_attrs("NAME")
```

Results:

```
NAME = "lower-mississippi"
NAME = "upper-mississippi"
```

11.9.2.4. Combination

You can combine the three different types of propagations in a single task using the syntax `node<...>[...](...)`, they must come in that sequence as the order is decided first, then the list/path is taken, and finally the condition is evaluated to boolean before selecting the nodes.

For example:

```
network load_file("./data/mississippi.net")
node[tenessee -> "lower-mississippi"]("mississippi" in NAME) INDEX
node<inv>[tenessee -> "lower-mississippi"] INDEX
node<inv>[tenessee -> "lower-mississippi"](ORDER > 1) INDEX
```

Results:

```
{
  lower-mississippi = 0
```

```
}
{
  lower-mississippi = 0,
  ohio = 5,
  tenessee = 6
}
{
  lower-mississippi = 0,
  ohio = 5
}
```

11.10. Network Function

Network function runs on the network as a whole. It takes arguments and keyword arguments. Few network functions we have been using throughout the examples are `load_file`, `load_str` and `svg_save`:

```
network load_file("./data/mississippi.net")
network command("mkdir -p output")
network svg_save(
   "./output/network-mississippi-sdf.svg",
  label="[{INDEX}] {_NAME:repl(-, ):case(title)}",
  bgcolor="gray"
)
```

Results:

Figure 20:

The examples below use the graphviz plugin. Make sure you have it loaded. Refer to the plugins section to learn how to load the plugins in to the NADI System.

For example following network function takes file path as input to save the network in graphviz format:

```
save_graphviz(
  outfile [PathBuf],
  name [String] = "network",
  global_attrs [String] = "",
  node_attr [Option < Template >],
  edge_attr [Option < Template >]
)
```

Note that, if the arguments have default values, or are optional, then you do not need to provide them.

For example, you can simply call the above function like this.

```
network load_file("./data/mississippi.net")
network save_graphviz("./output/test.gv")
network clip()
# the path link are relative to /src
network echo("./output/test.gv")
```

Results:

```
digraph network {

"upper-mississippi" -> "lower-mississippi"
"missouri" -> "lower-mississippi"
"arkansas" -> "lower-mississippi"
"red" -> "lower-mississippi"
"ohio" -> "lower-mississippi"
"tenessee" -> "ohio"
}
```

With extra commands you can also convert it into an image

```
network load_file("./data/mississippi.net")
network save_graphviz("./output/test.gv")
network command("dot -Tsvg ./output/test.gv -o ./output/test.svg")
network clip()
# the link path needs to be relative to this file
network echo("../output/test.svg")
```

Results:

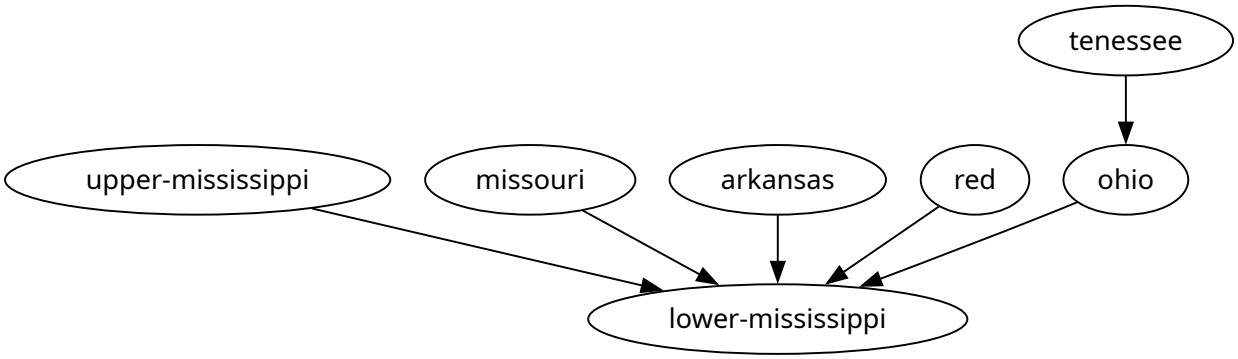


Figure 21:

11.11. Cross Context Functions and Variables

You can access variable and call functions based on their default context (e.g. node variable/function in a node task). Additionally, you can also access the variables or call functions in select few other context.

By default, if a function is not available, node/network task calls the environment function of the same name.

For example, here the `sum` and `array` functions are environment functions, while the `count` is a network function. When you use NADI IDE, it'll show you which function is actually being called at the top of the editor.

```
network load_str("a->b")
network sum(array(count(), 1))
```

Results:

```
3
```

Besides this, you can manually call cross context variable/functions in the following ways:

11.11.1. Env and Network Variables/Functions

You can use `env` and `network` variables anywhere in the task system with the dot syntax.

```
 network load_str("a->b")
env.var = 12;
network.sth = true;

env render("this is {x}", x = network.sth)
network str(env.var)
node array(network.sth, env.var, node.NAME)
```

Results:

```
"this is true"
"12"
{
  b = [true, 12, "b"],
  a = [true, 12, "a"]
}
```

Similary, env and network functions can be called anywhere. These functions cannot be mutable functions (change network internally).

Taking the previous example, if we use env function `count`, we get an error as the function arguments are different.

```
network load_str("a->b")
node network.count()
network sum(array(env.count(), 1))
```

Error:

```
{
  b = 2,
  a = 2
}
Error in function count: Argument 1 (vars [& [bool]]) is required
```

11.11.2. Node Variables/Functions

You can use `node`, `inputs`, `output` and `nodes` keywords to access node variables and functions from different contexts. `nodes` is valid in all tasks, while the other 3 are only valid in a node task and refer to the current node, input nodes and output node respectively.

```
network load_file("./data/mississippi.net")
env count(nodes._)
node inputs.NAME
```

Results:

```
7
{
  lower-mississippi = ["ohio", "upper-mississippi", "missouri", "arkansas", "red"],
  upper-mississippi = [],
  missouri = [],
  arkansas = [],
  red = [],
  ohio = ["tenessee"],
  tenessee = []
}
```

You can call node functions not just for the node in the context, but also for input nodes, and output node:

Please note that the root node (outlet) of the network doesn't have output node, so we need to skip that, which can be done through the `output._?` which is checking for the dummy variable _ in `output`, which is true if the node has an output.

```
network load_file("./data/mississippi.net")
node[tenessee -> "lower-mississippi"] inputs.render("{_NAME}")
node[tenessee -> "lower-mississippi"](output._?) output.render("{_NAME}")
```

Results:

```
{
  tenessee = [],
  ohio = ["tenessee"],
  lower-mississippi = ["ohio", "upper-mississippi", "missouri", "arkansas", "red"]
```

```
}
{
  tenessee = "ohio",
  ohio = "lower-mississippi"
}
```

You can also use `nodes` keyword to call the function on each node, it can be used anywhere, but is useful for env and network tasks.

```
network load_file("./data/mississippi.net")

env nodes.render("Node [{INDEX}] {_NAME}")
```

Results:

```
["Node [0] lower-mississippi", "Node [1] upper-mississippi", "Node [2] missouri",
"Node [3] arkansas", "Node [4] red", "Node [5] ohio", "Node [6] tenessee"]
```

11.12. Plugins

11.12.1. Plugins

Plugins are the main source of functions in the task system. There are two types of plugins,

- internal plugins: come with your nadi distribution, and
- external plugins: you can install them using shared libraries.

The functions available from the internal plugins are available in any format. You can access the functions with their name, or with `plugin.name` syntax.

External plugins are loaded from the directory in `NADI_PLUGIN_DIRS` environmental variable. If the shared library in that directory does not match the specification of NADI plugin, is compiled using different version, or compiled with different internal data structures, it will print warning and skip that file.

11.12.2. Security

Plugins can run arbitrary code, so users are expected to be careful while loading plugins by making sure it is not malicious.

Rust plugins give option to look at the code used to build the function, but it is only intended to be helpful, a malicious actor can still obfuscate, omit, or show different code than the actual function.

Furthermore, plugins can also replace functions if they have same name, look out for the warnings while running nadi.

Expect these Changes in the Future Versions to make the plugins system more secure in terms of function replacement problem:

- internal plugin names will be uppercase, and external lowercase, so that external plugins are not mistaken as internal ones, and no overwriting functions,
- external plugin functions always need to be accessed with dot syntax (`plugin.function(...)`). And so that external functions are not called accidently.

This still will not solve the first problem, so always be vigilant using plugins. It is recommended to only use plugins developed in house, or those that you have source code of and can compile it yourself.

11.13. Further Reading

11.13.1. Further Reading

If you need help on any functions. Use the `help` as a `task`. You can use `help node` or `help network` for specific help. You can also browse through the function help window in the `nadi-ide` for help related to each functions.

```
help node render
```

Results:

```
node render (template: '& Template', safe: 'bool' = false)
Render the template based on the node attributes
# Arguments
- `template: & Template` String template to render
- `safe: bool` [def = false] if render fails keep it as it is instead of exiting


For more details on the template system. Refer to the String
Template section of the NADI book.
```

```
```task
network load_str("a -> b")
node.x = 13
node assert_eq(render("abc {x}"), "abc 13")
```
```

Or you can use `nadi --fnhelp <function>` using the `nadi-cli`.

Now that you have the overview of the nadi system's data structures. We'll jump into the software structure and how to setup and use the system.

If you want more details on any of the data structures refer the Developer's references, or the library documentation.

12. Learn by Examples

This section teaches you the basics of the NADI Task System's syntax with small examples. If you want a higher level example that focuses on the actual research problem instead of syntax then refer to the "Example Research Problems" Section on the sidebar.

For example data download the zip file here

Topic	Learn About
Attributes	Setting and Getting Attributes
Control Flow	Control flow, if, else, while loops etc
Connections	Loading and modifying connections
Counting	Counting nodes in network, conditional
Cumulative	Calculating Network cumulative sums and those
Import Export	Importing and exporting multiple data formats
String Template	Using String Templates to do various things

12.1. Attributes

12.1.1. Attributes

There are 3 kind of attributes in nadi. Environment, Network and Node attributes. as their name suggests environment attributes are general attributes available in the current

context. Network attributes are associated with the currenly loaded network. and node attributes are associated with each nodes.

nadi has special syntax where you can get/set attributes for multiple nodes at once.

```
network load_str("a -> b\n b -> d\n c -> d\n");
# environmental attribute
env.someattr = 123;
env.other = 1998-12-21;
env array(someattr, other)
# network attribute
network.someattr = true;
network.someattr
# node attributes
node.someattr = "string val";

node.someattr
```

Results:

```
[123, 1998-12-21]
true
{
  d = "string val",
  c = "string val",
  b = "string val",
  a = "string val"
}
```

like you saw with the array function, variables used are inferred as the attributes of the current env/network/node task.

you can use attributes from outside of current task type in some cases like:

- env/network variables can be used anywhere
- node variables are valid in node tasks
- node tasks has special variables types like `inputs` and `output`

```
network load_str("a -> b\n b -> d\n c -> d\n");
# environmental attribute
env.someattr = 123;
env.other = 1998-12-21;
```

```
# network attribute
network.someattr = true;

# using network attr in env task
env array(network.someattr, other)

# using nodes in network task
network nodes.NAME
```

Results:

```
[true, 1998-12-21]
["d", "c", "b", "a"]
```

Similarly inputs:

```
network load_str("a -> b\n b -> d\n c -> d\n");

node inputs.NAME
```

Results:

```
{
  d = ["b", "c"],
  c = [],
  b = ["a"],
  a = []
}
```

Refer to the network diagram below to verify the output are correct:

```
network load_str("a -> b\n b -> d\n c -> d\n");
network svg_save("./output/attrs-simp.svg")
```

Results:

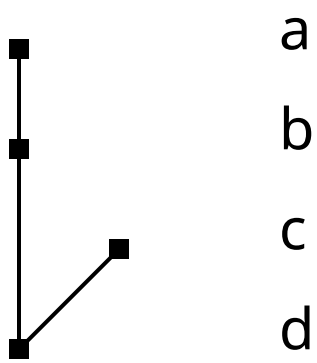


Figure 22:

12.2. Control Flow

Task has some basic control flow required to write programs. They are if-else branches and while loops.

12.2.1. Conditional (If-Else) Blocks

There are two kind of if-else branches. One is on an expression level. which means there has to be if and else branch both as it expects a return value. The following example shows the expression with if-else block.

```
env.newvar = if (12 > 90) {"yes"} else {"no"};
env.newvar
```

Results:

```
"no"
```

Trying to do it without else block will result in an parse error as the program will error with a syntax error, for example the code below is invalid

```
env.newvar = if (12 > 90) {"yes"};
env.newvar
```

That's when you can use the if-else block on the task level. This can be only if block as the execution blocks are tasks instead of expressions.

Here, since the condition is negative the task inside the block is never executed, hence `env.newvar` is empty.

```
if (12 > 90) {
  env.newvar = "yes";
}
env.newvar
```

Error:

```
EvalError: Attribute not found
```

12.2.2. While Loop

While loop runs the tasks inside the block repeatedly while the condition is satisfied. There is an iteration limit of 1,000,000 for now just in case people write infinite loop. This is arbritary.

```
env.somevar = 1;
while (somevar < 10) {
  env.somevar
  env.somevar = env.somevar + 1;
}
```

Results:

```
1
2
3
4
5
6
7
8
9
```

This can be used to repeat a set of tasks for a various reasons.

If your tasks take a long time to run, note that, the while loop needs to be completely run before the output can be processed and displayed, so that even if your output is not printed, it is being run. This will be fixed in the future version of the program.

12.3. Connections

Connections between the nodes is the most important part of nadi. you can load networks by loading a file or string. The network is a simple multiline text with one edge (`input -> output`) in each line. comments starting with `#` are supported.

12.3.1. Default is Empty Network

Tasks are run by default with an empty network. So you might still be able to work with network attributes, but the nodes will be empty. also note that when you load network it replaces the old one including the attributes.

```
network.someattr = 1234;
network.someattr
```

Results:

```
1234
```

But we can see the nodes are not there,

```
network count()
network nodes.NAME
```

Results:

```
0
[]
```

Trying to run node functions on the empty network means nothing is run

```
node render("{NAME}")
```

Results:

```
```

12.3.2. Loading Network from String

Here assume we have a network consisting of nodes of dams and gages like the following where dam nodes start with d and gages with g:

```
network load_str("
d1 -> d2
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
network svg_save(
   "./output/simple-count.svg",
  label="[{INDEX}] {_NAME}"
)
```

Results:

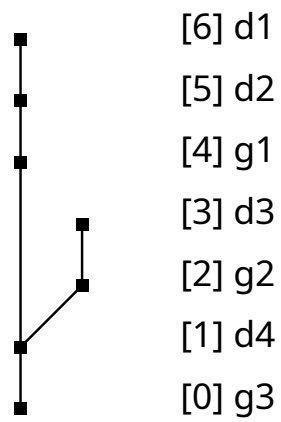


Figure 23:

12.3.3. Loading Network from a File

we can load a network from a file:

```
network load_file("./data/mississippi.net");
network svg_save(
   "./output/ex-network-conn.svg",
  label="[{INDEX}] {_NAME}"
)
```

Results:

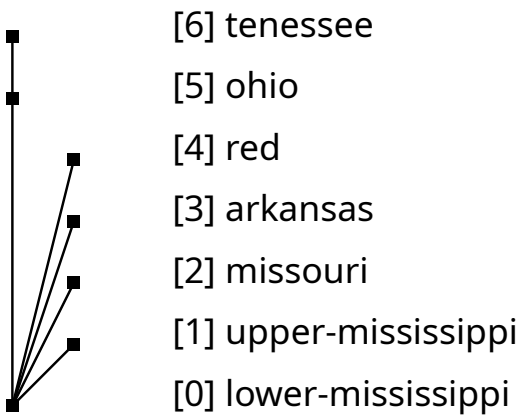


Figure 24:

12.3.4. Modifying the network

You can modify the network after loading it as well. The example below extracts just the nodes that are dams. Compare this with the previous network to see how the connections are retained during the subsets.

```
network load_str("
d1 -> d2
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node.is_dam = NAME match "^d[0-9]+";
network subset(nodes.is_dam);
network svg_save(
   "./output/simple-count-subset.svg",
  label="[{INDEX}] {_NAME}"
)
```

Results:

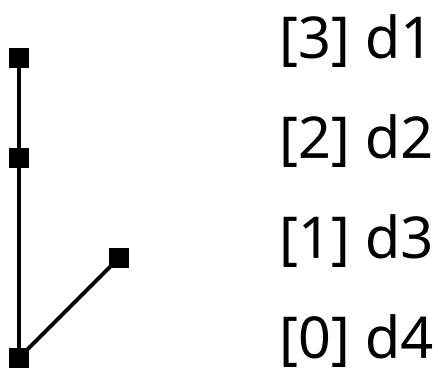


Figure 25:

This can be useful when you want to remove nodes that do not satisfy some selection criteria for your analysis without having to redo the network detection part.

12.4. Counting Nodes

Here assume we have a network consisting of nodes of dams and gages like the following where dam nodes start with d and gages with g:

```
network load_str("
d1 -> d2
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
network svg_save(
   "./output/simple-count.svg",
  label="[{INDEX}] {_NAME}"
)
```

Results:

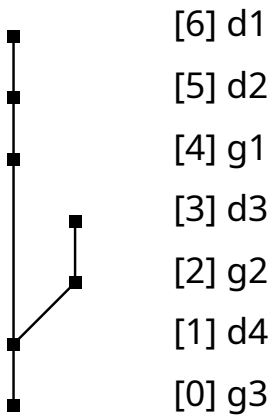


Figure 26:

Simply counting number of nodes, or certain types of nodes in a network is done through `count` function.

```
network load_str("
d1 -> d2
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node.g_node = NAME match "^g[0-9]+";
network count()
network count(nodes.g_node)
network count(nodes.g_node) / count()
```

Results:

```
7
3
0.42857142857142855
```

when you call a network function, you get one output, while a node function will give you the output for each node like here:

```
network load_str("
d1 -> d2
```

```
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node.g_node = NAME match "^g[0-9]+";
node.g_node
```

Results:

```
{
  g3 = true,
  d4 = false,
  g2 = true,
  d3 = false,
  g1 = true,
  d2 = false,
  d1 = false
}
```

Always be careful that node function is run for all the nodes separately, if you are running them without any variables from the node, then you can use network function, or environment function to get the results.

Counting the number of nodes upstream of each node gives us the order of the nodes.

```
network load_str("
d1 -> d2
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node<inputsfirst>.nodes_us = 1 + sum(inputs.nodes_us);
network svg_save(
   "./output/simple-count-1.svg",
  label="{_NAME} = {nodes_us}"
)
```

Results:

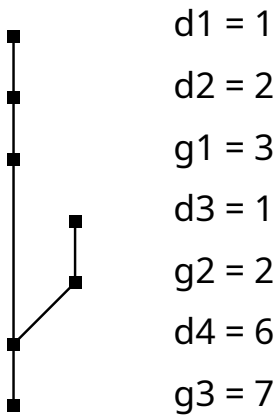


Figure 27:

We can add a condition and count the nodes that satisfy that condition only. Like counting the number of dams upstream of each node (including the node).

```
network load_str("
d1 -> d2
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node.is_dam = NAME match "^d[0-9]+";
node<inputsfirst>.dams_us = int(is_dam) + sum(inputs.dams_us);
network svg_save(
   "./output/simple-count-2.svg",
  label="{_NAME} = {dams_us}"
)
```

Results:

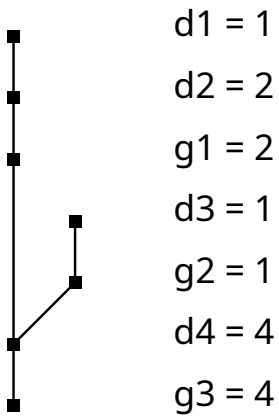


Figure 28:

You can similarly count the number of gages downstream. Here we need a conditional unlike in previous cases as not all nodes have output. In case of inputs, a leaf node would have no inputs but `sum([])` would still be a valid output of `0`. But for node without output nodes, the variable type `output` fails with `NoOutputNode` error, so we add a conditional check to avoid that.

```
network load_str("
d1 -> d2
d3 -> g2
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node.is_gage = NAME match "^g[0-9]+";
node<outputfirst>.gages_ds = int(is_gage) + if (output._?) {
  output.gages_ds
  } else {
  0
};
network svg_save(
   "./output/simple-count-3.svg",
  label="{_NAME} = {gages_ds}"
)
```

Results:

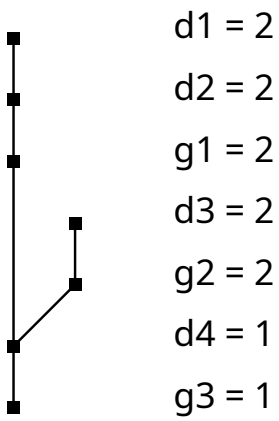


Figure 29:

Here the condition `(output._?)` checks if there is output on the node or not by checking for the dummy variable _ which is present in all nodes/network.

12.5. Cumulative Sum

Here we can use the stream ordering formula to calculate the stream order for each node:

```
network load_str("
d1 -> d2
d3 -> g1
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node<inputsfirst>.stream_ord = max(inputs.stream_ord, 1) + int(count(inputs._?) >
1);
network svg_save(
   "./output/cumulative-1.svg",
  label="{_NAME} = {stream_ord}"
)
```

Results:

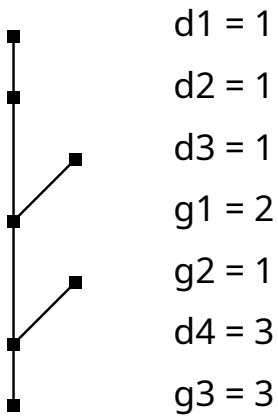


Figure 30:

The first part takes the maximum order of the input nodes, then the second part `int(count(inputs._?) > 1)` checks if there are more than one input, adding one to the order when multiple streams combine into one. You can use the funciton `inputs_count()` instead of `count(inputs._?)` to do the same thing.

That is the core of the NADI Task System, you can write functions that have their own logic and then load them into the system. You can then use the syntax and network based analysis methods of NADI using those functions.

And of course, we can visualize the different order of streams for easier understanding.

```
network load_str("
d1 -> d2
d3 -> g1
d2 -> g1
g1 -> d4
g2 -> d4
d4 -> g3
");
node<inputsfirst>.stream_ord = max(inputs.stream_ord, 1) + int(count(inputs._?) >
1);
node.visual.linewidth = stream_ord / 2;
node(stream_ord == 1).visual.linecolor = "green";
node(stream_ord == 2).visual.linecolor = "blue";
node(stream_ord == 3).visual.linecolor = "red";
network svg_save(
   "./output/cumulative-2.svg",
  label="{_NAME} = {stream_ord}"
)
```

Results:

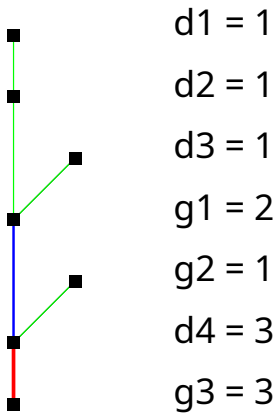


Figure 31:

12.6. Import Export Files

12.6.1. Import Export Files

Similar to how you can load network files, you can load attributes from files as well. Direct load of TOML format is supported from the internal plugins, while you might need external plugins for other formats.

`load_attrs` function takes a template, and reads a different files for each node to load the attributes from.

```
network load_file("data/ohio.network")
node attributes.load_attrs("data/attrs/{_NAME}.toml")
network svg_save(
  "output/ohio-import-export.svg",
  label="{_NAME} (A = {basin_area?:f(2)})",
  height=700,
  bgcolor="gray"
)
```

Results:

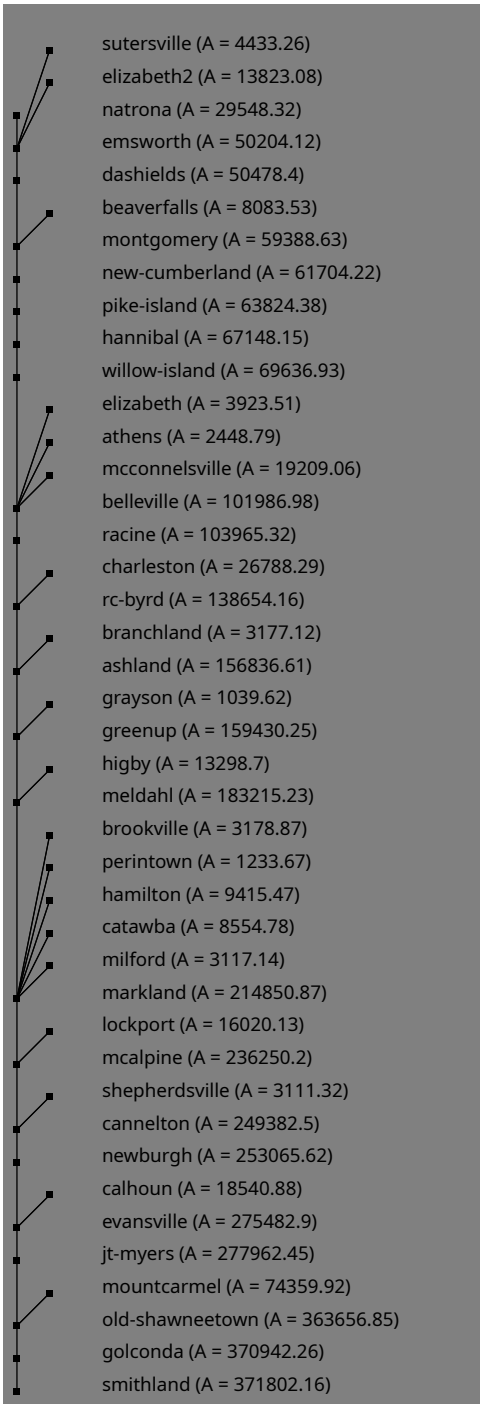


Figure 32:

You can use the render function to see if the files being loaded are correct. Here we can see the examples for the first 4 nodes:

```
network load_file("data/ohio.network")
node(INDEX<4) render("data/attrs/{_NAME}.toml")
```

Results:

```
{
  smithland = "data/attrs/smithland.toml",
  golconda = "data/attrs/golconda.toml",
  old-shawneetown = "data/attrs/old-shawneetown.toml",
  mountcarmel = "data/attrs/mountcarmel.toml"
}
```

You can also read a attributes from string, so you can combine that with `files.from_file` and load it.

```
network load_file("data/ohio.network")
env.somevalue = attributes.parse_attrmap(
  files.from_file("data/attrs/smithland.toml")
);
env.somevalue.basin_area
env.somevalue.length
```

Results:

```
371802.16
1675.95
```

You can export csv files

```
network load_file("data/ohio.network")
node attributes.load_attrs("data/attrs/{_NAME}.toml")
network table.save_csv("output/ohio-export.csv", ["NAME", "basin_area", "length"])
network command("cat output/ohio-export.csv | head", echo=true)
```

Results:

```
$ cat output/ohio-export.csv | head
NAME,basin_area,length
"smithland",371802.16,1675.95
"golconda",370942.26,1701.32
"old-shawneetown",363656.85,1772.27
"mountcarmel",74359.92,1918.08
"jt-myers",277962.45,1791.07
```

```
"evansville",275482.9,1878.29
"calhoun",18540.88,1992.5
"newburgh",253065.62,1903.58
"cannelton",249382.5,1993.72
```

12.6.1.1. GIS Files

The examples below require the `gis` external plugin from `nadi-gis` repository to work. Make sure you have the plugin file in the directory in your `NADI_PLUGIN_DIRS` environmental variable.

First we make a GIS file by exporting. The image below shows the resulting points (red) from the shapefile and connections (black) from the Geopackage file when we visualize this on QGIS (with background of Terrain and Ohio River tributaries).

```
network load_file("data/ohio.network")
node attributes.load_attrs("data/attrs/{_NAME}.toml")
node.geometry = render("POINT ({lon} {lat})");
network gis.gis_save_nodes(
  "output/ohio-nodes.shp",
  "geometry",
  {
    NAME = "String",
  basin_area = "Float",
  length = "Float"
  }
)
# Exporting the edges
network gis.gis_save_connections(
  "output/ohio-connections.gpkg",
  "geometry"
)
```

Results:

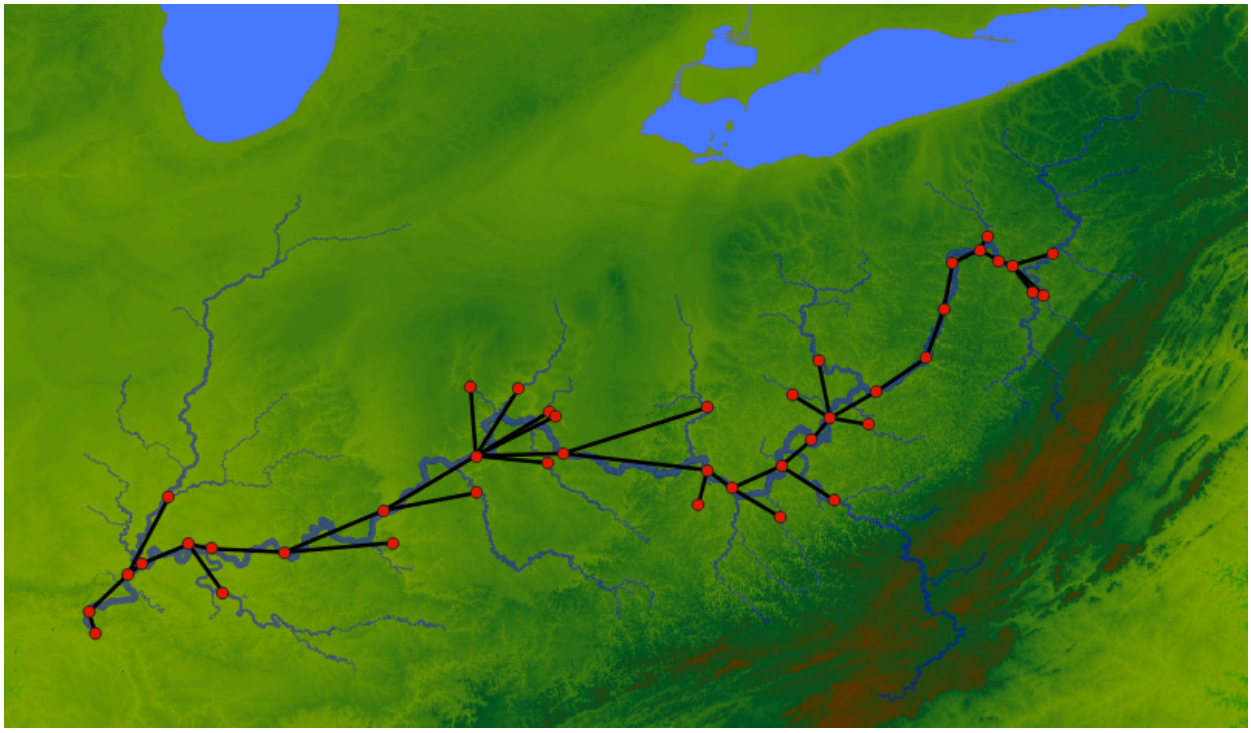


Figure 33:

The geometry attributes should be WKT String.

Now we are using the generated GIS files to load the network and the attributes:

```
network gis.gis_load_network("output/ohio-connections.gpkg", "start", "end")
network gis.gis_load_attrs("output/ohio-nodes.shp", "NAME")

network svg_save(
  "output/ohio-from-gis.svg",
  label="{_NAME} (A = {basin_area?:f(2)}; L = {length:f(1)})",
  height=700,
  bgcolor="gray"
)
```

Results:

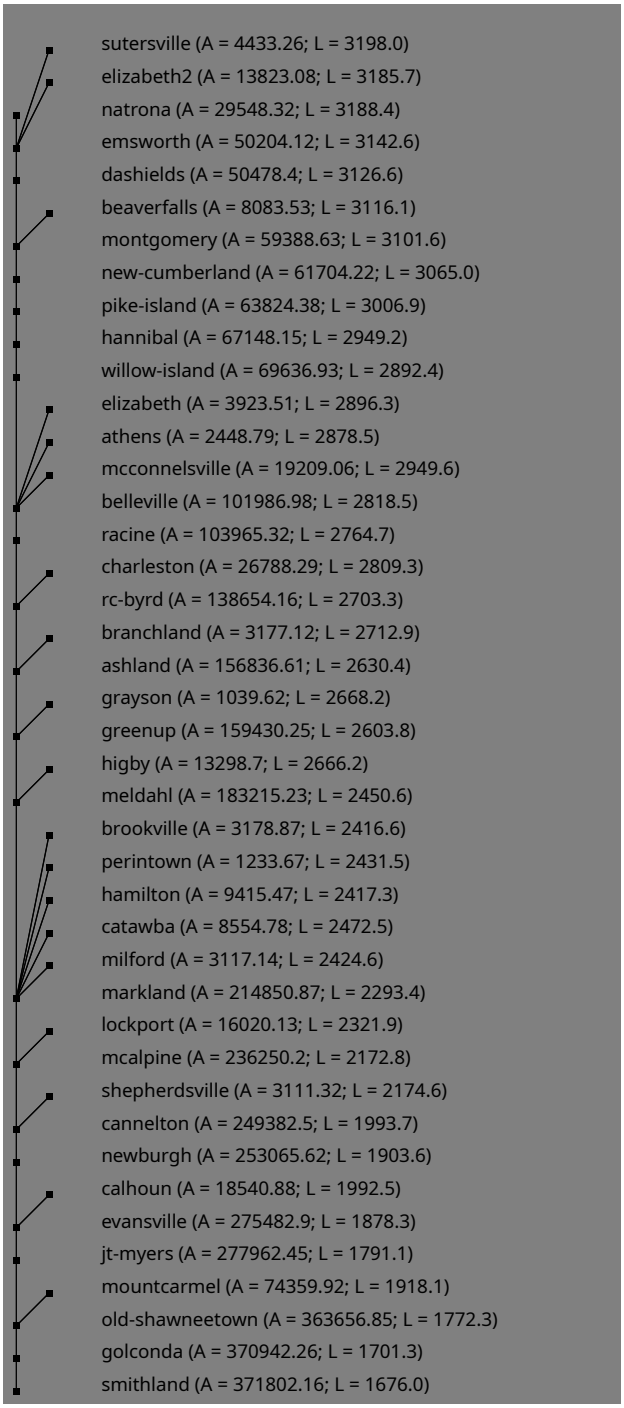


Figure 34:

As we can see the plugins make it easier to interoperate with a lot of different data formats. Here GIS plugin will support any file types supported by `gdal`. Similarly, other formats can be supported by writing plugins.

12.7. String Templates

13. NADI Extension Capabilities

NADI System can be extended for custom use cases with the following ways:

- LISP on String Template
- Task System
- Rust Library
- Python Library
- Plugin System

14. List of All Functions

All the functions available on this instance of nadi, are listed here.

14.1. Env Functions

Plugin	Function	Help
core	int	make an int from the value
debug	clip	Echo the ----8<---- line for clipping syntax
attributes	log	Logarithm of a value, natural if base not given
regex	str_split	Split the string with the given pattern
core	type_name	Type name of the arguments
core	min_num	Minimum of the variables
core	count	Count the number of true values in the array
core	range	Generate integer array, end is not included
core	assert_eq	Assert the two values are equal
debug	debug	Print the args and kwargs on this function
logic	gt	Greater than check
connections	root_node	default name used for ROOT node of the network
core	day	day from date/datetime
core	unique_str	Get a list of unique string values
attributes	sqrt	Square Root
attributes	float_mult	Float Multiplication (same as * operator)

Plugin	Function	Help
logic	ifelse	Simple if else condition
regex	str_replace	Replace the occurances of the given match
core	max_num	Minimum of the variables
core	json	format the attribute as a json string
core	year	year from date/datetime
attributes	powi	Integer power
core	isinf	check if a float is +/- infinity
core	str	make a string from value
regex	str_find	Find the given pattern in the value
dss	list_catalog	List the catalog of the dss file
core	float	make a float from value
core	attrmap	make an attrmap from the arguments
attributes	strmap	map values from the attribute based on the given table
files	from_file	Reads the file contents as string
logic	or	boolean or
core	sum	Sum of the variables
attributes	float_div	Float Division (same as / operator)
attributes	parse_attr	Parse attribute from string
core	month	month from date/datetime
logic	eq	Equality than check
core	isna	check if a float is nan
core	min	Minimum of the variables
core	prod	Product of the variables
logic	lt	Less than check
debug	echo	Echo the string to stdout or stderr
render	render	Render the template based on the node attributes
logic	not	boolean not
files	exists	Checks if the given path exists
regex	str_match	Check if the given pattern matches the value or not
core	append	append a value to an array
attributes	powf	Float power
regex	str_find_all	Find all the matches of the given pattern in the value
files	to_file	Writes the string to the file
core	assert_neq	Assert the two values are not equal
debug	sleep	sleep for given number of milliseconds

Plugin	Function	Help
regex	str_filter	Filter from the string list with only the values matching pattern
attributes	exp	Exponential
core	max	Maximum of the variables
core	concat	Concat the strings
core	array	make an array from the arguments
logic	all	check if all of the bool are true
regex	str_count	Count the number of matches of given pattern in the string
attributes	get	get the choosen attribute from Array or AttrMap
nadi_pdf	typst_compile	convert the typst content into pdf/svg/png
core	assert	Assert the condition is true
core	count_str	Get a count of unique string values
logic	any	check if any of the bool are true
attributes	parse_attrmap	Parse attribute map from string
logic	and	Boolean and
core	length	length of an array or hashmap

14.2. Node Functions

Plugin	Function	Help
timeseries	ts_list	List all timeseries in the node
timeseries	ts_dtype	Type name of the timeseries
attributes	set_attrs_ifelse	if else condition with multiple attributes
series	set_series	set the following series to the node
series	sr_to_array	Make an array from the series
core	inputs_count	Count the number of input nodes in the node
series	sr_mean	Type name of the series
render	render	Render the template based on the node attributes
files	exists	Checks if the given path exists when rendering the template
series	sr_sum	Sum of the series
errors	calc_ts_error	Calculate Error from two timeseries values in the node
core	output_attr	Get attributes of the output node
attributes	print_all_attrs	Print all attrs in a node

Plugin	Function	Help
dams	count_node_if	Count the number of nodes upstream at each point that satisfies a certain condition
dams	min_year	Propagate the minimum year downstream
attributes	print_attrs	Print the given node attributes if present
command	command	Run the given template as a shell command.
series	sr_dtype	Type name of the series
streamflow	check_negative	Check the given streamflow timeseries for negative values
attributes	set_attrs	Set node attributes
timeseries	ts_count	Number of timeseries in the node
timeseries	ts_len	Length of the timeseries
attributes	load_toml_render	Set node attributes by loading a toml from rendered template
errors	calc_ts_errors	Calculate Error from two timeseries values in the node
attributes	has_attr	Check if the attribute is present
attributes	first_attr	Return the first Attribute that exists
timeseries	ts_print	Print the given timeseries values in csv format
print_node	print_node	Print the node with its inputs and outputs
core	has_outlet	Node has an outlet or not
core	inputs_attr	Get attributes of the input nodes
attributes	get_attr	Retrive attribute
datafill	load_csv_fill	
series	sr_list	List all series in the node
series	sr_len	Length of the series
command	run	Run the node as if it's a command if inputs are changed
datafill	datafill_experiment	
attributes	load_attrs	Loads attrs from file for all nodes based on the given template
series	sr_count	Number of series in the node
attributes	set_attrs_render	Set node attributes based on string templates

14.3. Network Functions

Plugin	Function	Help
connections	load_file	Load the given file into the network

Plugin	Function	Help
table	table_to_markdown	Render the Table as a rendered markdown
timeseries	series_csv	Write the given nodes to csv with given attributes and series
graphviz	save_graphviz	Save the network as a graphviz file
core	node_attr	Get the attr of the provided node
print_node	print_attr_csv	Print the given attributes in csv format with first column with node name
command	command	Run the given template as a shell command.
connections	subset_largest	Take a subset of network by only including the largest blob of connected nodes
render	render_template	Render a File template for the nodes in the whole network
gis	gis_load_network	Load network from a GIS file
attributes	set_attrs_render	Set network attributes based on string templates
connections	subset	Take a subset of network by only including the selected nodes
errors	calc_attr_error	Calculate Error from two attribute values in the network
timeseries	ts_print_csv	Save timeseries from all nodes into a single csv file
datafill	save_experiments_csv	Write the given nodes to csv with given attributes and experiment results
nadi_pdf	typst_table	Generate Typst code for given Table
connections	save_file	Save the network into the given file
render	render_nodes	Render each node of the network and combine to same variable
fancy_print	fancy_print	Fancy print a network
gis	gis_save_nodes	Save GIS file of the nodes
connections	load_edges	Load the given edges as a network
html	export_map	Exports the network as a HTML map
gis	gis_save_connections	Save GIS file of the connections
attributes	set_attrs	Set network attributes
gnuplot	plot_timeseries	Generate a gnuplot file that plots the timeseries data in the network
core	outlet	Get the name of the outlet node
table	save_csv	Save CSV

Plugin	Function	Help
visuals	set_nodesize_attrs	Set the node size of the nodes based on the attribute value
core	count	Count the number of nodes in the network
render	render	Render from network attributes
gis	gis_load_attrs	Load node attributes from a GIS file
connections	load_str	Load network from the given string
visuals	svg_save	Exports the network as a svg
command	parallel	Run the given template as a shell command for each nodes in the network in parallel.
connections	subset_from	Take a subset of network by taking the given node as new outlet

Example Research Problems

15. Effects of Dams

Ohio River is one of the largest river basin in the USA. The development of the basin and its river system started early, so we have a problem where many streamgages never recorded the river in their natural state. In these examples, we'll try to load a network, validate the network based on metadata, count the dams/gages, and get the year of earliest constructed dam upstream for further analysis.

The examples in this chapter loads the data for the dams from National Inventory of Dams (NID), and USGS gages from Gages.shp (EPA and USGS), and uses the network developed using the NHDPlus streamlines above the Ohio River at Smithland Streamgage.

The files used in this demo are follows:

Data	Filename	Source
NID Data	`nid-uniq.gpkg`	National Inventory of Dams
Gage Locations	`GageLoc.shp.zip`	US EPA & USGS
Gage Attributes	`gages-II.gpkg`	USGS
Gage Drainage	`usgs-drainage.csv`	Scrapped from USGS NWIS

Except NID data (due to large size), the others are also in the `src/data/ohio-river` directory on the repository of this book. Place the nid data into it before running the examples from this chapter. If you're using the repository, you also have to unzip the `gages-II.gpkg.zip` into `gages-II.gpkg`.

The Gage Drainage data is scrapped manually for some basins from the USGS website to increase the amount of gage we know the drainage area of, you can skip that in loading and still be able to do your analysis.

15.1. Validating Network

In this example we will use `nadi-gis` plugin to load and write GIS files, while using QGIS to generate the map visualization. The plugin is external, refer to the installation page to install it into the NADI System.

Different from other chapters, the code in this chapters are run in the order they are given, meaning each task code blocks are not independent.

15.1.1. Load Network and Attributes

We'll use the same network from the Dams count example.

```
network load_file("data/ohio-river/ohio.network")
network count()
network outlet()

node.is_usgs = NAME match "^[0-9]+";
node.is_dam = !is_usgs;
network count(nodes.is_usgs)
network count(nodes.is_dam)
```

Results:

```
5987
"03399800"
1806
4181
```

Then we load the node attributes from the gis files, we can see some of the variables are loaded from NID website.

```
network gis_load_attrs("data/ohio-river/nid-uniq.gpkg", "nidId")
network gis_load_attrs("data/ohio-river/GageLoc.shp.zip", "SOURCE_FEA")
network gis_load_attrs("data/ohio-river/gages-II.gpkg", "STAID")
network gis_load_attrs("data/ohio-river/usgs-drainage.csv", "SiteNumber")

node[03399800].SiteNumber # only USGS gages have it
node(INDEX < 5).yearCompleted # only NID dams have it
node(INDEX < 5).GEOM # all nodes have it although from different sources
```

Results:

```
{
  03399800 = <None>
}
```

```
{
  03399800 = <None>,
  IL50499 = 1980,
  IL00070 = 1960,
  IL00955 = 1977,
  IL40055 = 1980
}
{
  03399800 = "POINT (-88.4234349499133 37.1612042742605 0)",
  IL50499 = "POINT (-88.42921894 37.16622664)",
  IL00070 = "POINT (-88.57139 37.21975)",
  IL00955 = "POINT (-88.78582 37.38365)",
  IL40055 = "POINT (-88.77145 37.401709)"
}
```

Also note that, while loading GIS files it also loads the geometry into “GEOM” attribute (you can change it through `change` function argument).

15.1.2. Process Attributes

Since we need a quick and easy way to identify USGS gage and NID dam, we use regex to categorize them.

```
node.is_usgs = NAME match "^[0-9]+";
node.is_dam = !is_usgs;
network count(nodes.is_usgs)
network count(nodes.is_dam)
```

Results:

```
1806
4181
```

The basin area is stored as different variable in each dataset, we want to combine them into a single one.

```
node.ba = first_attr(["drainageArea", "DRAIN_SQKM", "Drainage"])
network count(nodes.ba?) / count()
```

Results:

```
0.7561382996492401
```

Even then we only have 75% of the nodes with basin area. Since the `DrainageArea` is in sqmiles

If we look at some examples here on two sources of basin area of USGS gages, we can see the problems:

```
node(is_usgs & Drainage? & DRAIN_SQKM? & INDEX < 100) array(Drainage, DRAIN_SQKM)
```

Results:

```
{
  03384450 = ["42.9", 111.1266],
  03383000 = ["255.0", 660.5415],
  03382100 = ["147.0", 367.272]
}
```

1. The `Drainage` value are `String`,
2. The units are different (confirmed from metadata of datasets).

So we reconcile that,

```
node(Drainage?).basin_area = float(Drainage);
node(DRAIN_SQKM?).basin_area = DRAIN_SQKM * 0.38610216;
```

Error:

```
Error in function float: cannot parse float from empty string
```

We got an error, seems like `Drainage` contains empty strings as well. That and the `String` part comes from loading a CSV without data types (`.csvt` file).

Let's fix the code for that, and also let's get the `basin_area` for dams.

```
node(Drainage? & Drainage match "[0-9]+(.[0-9]+)?").basin_area = float(Drainage);
node(DRAIN_SQKM?).basin_area = DRAIN_SQKM * 0.38610216;
node(drainageArea?).basin_area = float(drainageArea);
```

```
node(INDEX < 10).basin_area
network count(nodes.basin_area?) / count()
```

Results:

```
{
  03399800 = <None>,
  IL50499 = 144000,
  IL00070 = <None>,
  IL00955 = <None>,
  IL40055 = 0.08,
  IL00039 = <None>,
  03386500 = 9.853134072120001,
  IL00102 = 2,
  03386000 = <None>,
  03385500 = <None>
}
0.755971271087356
```

15.1.3. Check for Incorrect Basin Areas

Now we have all the different variables combined into a single one with same data type and unit. Let's do a quick analysis to see if there are points in the network where nodes have larger area than input nodes.

```
node.incorrect = basin_area < sum(inputs.basin_area);
network count(nodes.incorrect)
```

Error:

```
Attribute not found
```

The error here comes from the fact that not all nodes contain `basin_area` attribute. Since `basin_area?` will only check for the current node, we use a default value when the values are not available into a temporary variable.

```
node.temp_ba = basin_area ? 0.0;
node.incorrect = basin_area? & (basin_area < sum(inputs.temp_ba));
```

```
network count(nodes.incorrect)
network count(nodes.incorrect) / count()
```

Results:

```
547
0.09136462335059295
```

That's around 10% error rate.

Looking at the actual values, few of them seem to be due to minor errors in values, while most of them seem to be from large input area.

```
node(incorrect & INDEX < 400) array(basin_area, sum(inputs.temp_ba))
```

Results:

```
{
  IL00028 = [1.5, 30.036998660096],
  IL50443 = [0.04, 141009.11000000002],
  KY00192 = [0.07, 0.31],
  IN04021 = [0.08, 0.2],
  IN03653 = [0.04, 0.37],
  IL50585 = [0.03, 0.16],
  IL01085 = [240, 240.171142047264],
  IL00606 = [0.6, 57],
  IN00316 = [0.09, 0.15],
  IN00317 = [0.15, 0.7000000000000001],
  IN00319 = [0.7000000000000001, 43.053904369656],
  IN03161 = [0.06, 0.07],
  IN03160 = [0.07, 3.0100000000000002],
  IN04048 = [0.4, 0.53],
  IN00627 = [0.53, 1.1300000000000001],
  IN03433 = [0.03, 1.82],
  IN00617 = [0.27, 0.49],
  IN00480 = [0.56, 21.637975860936002],
  IN00618 = [0.41000000000000003, 1.2700000000000002],
  IN00098 = [0.33, 263.021357392616],
  IN00035 = [0.23, 168.12355234608],
  03343010 = [13732, 13755.171304576803],
  IN00528 = [0.21, 0.32],
```

```
  IN00151 = [1.18, 4.75],
  IN03337 = [1.1500000000000001, 42.9097178978],
  IN00107 = [0.55, 8.22],
  IN00209 = [1.8, 8.48],
  IN00091 = [0.6900000000000001, 0.87],
  IN00092 = [0.87, 8.38]
}
```

This could be due to the error in snapping to the streamlines. It is also possible that some basin area that are missing are contributing to the unaccounted area.

Let's try to fix them, or avoid points without data.

```
node.all_inputs_ba = all(inputs.basin_area?);
node.incorrect = basin_area? & all_inputs_ba & (
  basin_area < (0.975 * sum(inputs.basin_area))
);
network count(nodes.incorrect)
network count(nodes.incorrect) / count()
```

Results:

```
451
0.07532988140972106
```

Looks like 7.5% of the errors are reasonable errors from network detection.

```
network count(nodes.incorrect & nodes.is_usgs)
network count(nodes.incorrect & nodes.is_dam)
```

Results:

```
8
443
```

This shows that the majority of the errors come from the NID dams, which makes sense given the `GageLoc.shp` data has points indexed to the NHDPlus streamlines, while the NID Dams are not indexed to it. Furthermore, looking at the attributes in QGIS shows, many

dams are in locations without streamlines, and even then sometimes they have unusually large basin area values for their location.

15.1.4. Export to GIS for visualization

We can export this result to a GIS file and look into individual cases in QGIS.

```
network gis_save_nodes(
    "output/ohio-gages-check.gpkg",
  "GEOM",
  {
    NAME="String",
    is_usgs="String",
    incorrect="String",
    basin_area="Float"
  }
)
```

Results:

```
```

We also need connection information to cross reference, let's make them through some string manipulation. NADI GIS plugin uses WKT format for geometries, which are string values, so we can manipulate them using string functions.

Note: we should probably add a function for this in future. A WKT plugin to work with geometries.

```
node.xy_coords = str_find_all(node.GEOM, "-?[0-9]+[.][0-9]+");
node(output._?).out_coords = str_find_all(output.GEOM, "-?[0-9]+[.][0-9]+");
node(! output._?).conn_geom = GEOM
node(output._?).conn_geom = env.render(
  "LINESTRING ({_x1} {_y1}, {_x2} {_y2})",
  x1=get(node.xy_coords, 0),
  y1=get(node.xy_coords, 1),
  x2=get(node.out_coords, 0),
  y2=get(node.out_coords, 1)
)
node(output._? & (INDEX < 10)).conn_geom

network gis_save_connections("output/ohio-conn-all.gpkg", "conn_geom")
```

Results:

```
{
  IL50499 = "LINESTRING (-88.42921894 37.16622664, -88.4234349499133
37.1612042742605)",
  IL00070 = "LINESTRING (-88.57139 37.21975, -88.42921894 37.16622664)",
  IL00955 = "LINESTRING (-88.78582 37.38365, -88.42921894 37.16622664)",
  IL40055 = "LINESTRING (-88.77145 37.401709, -88.42921894 37.16622664)",
  IL00039 = "LINESTRING (-88.68815 37.38965, -88.42921894 37.16622664)",
  03386500 = "LINESTRING (-88.6736645 37.4156072, -88.68815 37.38965)",
  IL00102 = "LINESTRING (-88.66408 37.41518, -88.6736645 37.4156072)",
  03386000 = "LINESTRING (-88.6637356477429 37.4140878122472, -88.66408 37.41518)",
  03385500 = "LINESTRING (-88.6494395725886 37.4141446558289, -88.6637356477429
37.4140878122472)"
}
```

When we visualize the output in QGIS, we can see the nodes where the `basin_area` are not correct based on the network information.

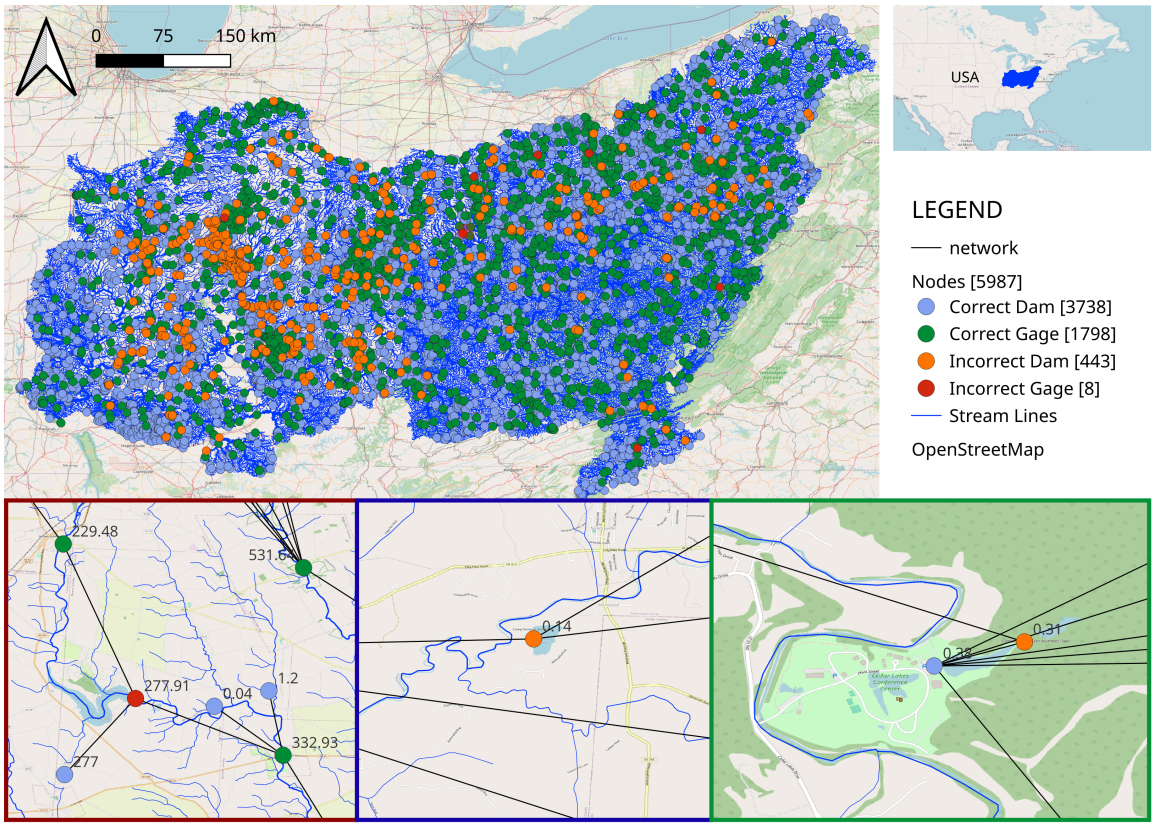


Figure 35: Correctness in QGIS

The zoomed in images at the bottom shows some of the reasons:

- [left] Incorrect Data in database (I think the dam coordinates are wrong on the blue dot with 277 basin area),
- [middle & right] Not enough streamlines for smaller details, and
- Nearest river detection algorithm problem from not considering river width.

The example for the last problem can be seen in the Video demostration for NADI QGIS Plugin.

15.2. Counting Ohio Dams

In the previous example we validated the network using basin_area. You can also use other properties like river mile to validate the network connection. Or to validate the metadata.

In this example, we'll be assuming there are some errors and continue with our analysis. Because the errors are mostly in the placement of dams, the count of dam upstream of a gage is still the same. For example:

If our network is:

```
gage1 -> dam1
dam1 -> gage2
```

instead of:

```
gage1 -> gage2
dam1 -> gage2
```

For both gage, number of dams upstream (or the first dam construction year upstead) is still the same, even though there is error on the same calculation in the dam.

15.2.1. Load Network and Attributes

First load the network

```
network load_file("data/ohio-river/ohio.network")
network count()
network outlet()
```

Results:

```
5987
"03399800"
```

15.2.2. Identify Dams and Gages

Since we need a quick and easy way to identify USGS gage and NID dam, we use regex to categorize them.

```
node.is_usgs = NAME match "^[0-9]+";
node.is_dam = !is_usgs;
network count(nodes.is_usgs)
network count(nodes.is_dam)
```

Results:

```
1806
4181
```

15.2.3. Count Dams/Gages Upstream

Now we simply count the number of dams and gages upstream recursively. We run it inputs first, so that the count begins with leaf nodes where we get 1 for the category we're counting and 0 otherwise, then we propagate that value downstream.

```
node<inp>.ngage = int(is_usgs) + sum(inputs.ngage);
node<inp>.ndam = int(!is_usgs) + sum(inputs.ndam);

node(INDEX < 10) array(ngage, ndam)
```

Results:

```
{
  03399800 = [1806, 4181],
  IL50499 = [1805, 4181],
  IL00070 = [0, 1],
  IL00955 = [0, 1],
  IL40055 = [0, 1],
```

```
  IL00039 = [3, 2],
  03386500 = [3, 1],
  IL00102 = [2, 1],
  03386000 = [2, 0],
  03385500 = [1, 0]
}
```

If we run this code without the `inputsfirst`/`inp` propagation, we get an error because the node's `inputs` will not have `ngage` and `ndam` values.

The counting is done, super simple right? NADI's ability to work with networks mean that some things are super simple compared to other languages.

15.2.4. GIS Visualization

Again, if we use `nadi-gis` plugin, then we can export this result and visualize it in QGIS.

```
network gis_save_nodes(
  "output/ohio-gages-count.gpkg",
  "GEOM",
  {
    NAME="String",
    is_usgs="String",
    ndam="Integer",
    ngage="Integer"
  }
)
```

After applying some filters and styles in QGIS, we can look at the numbers visually,

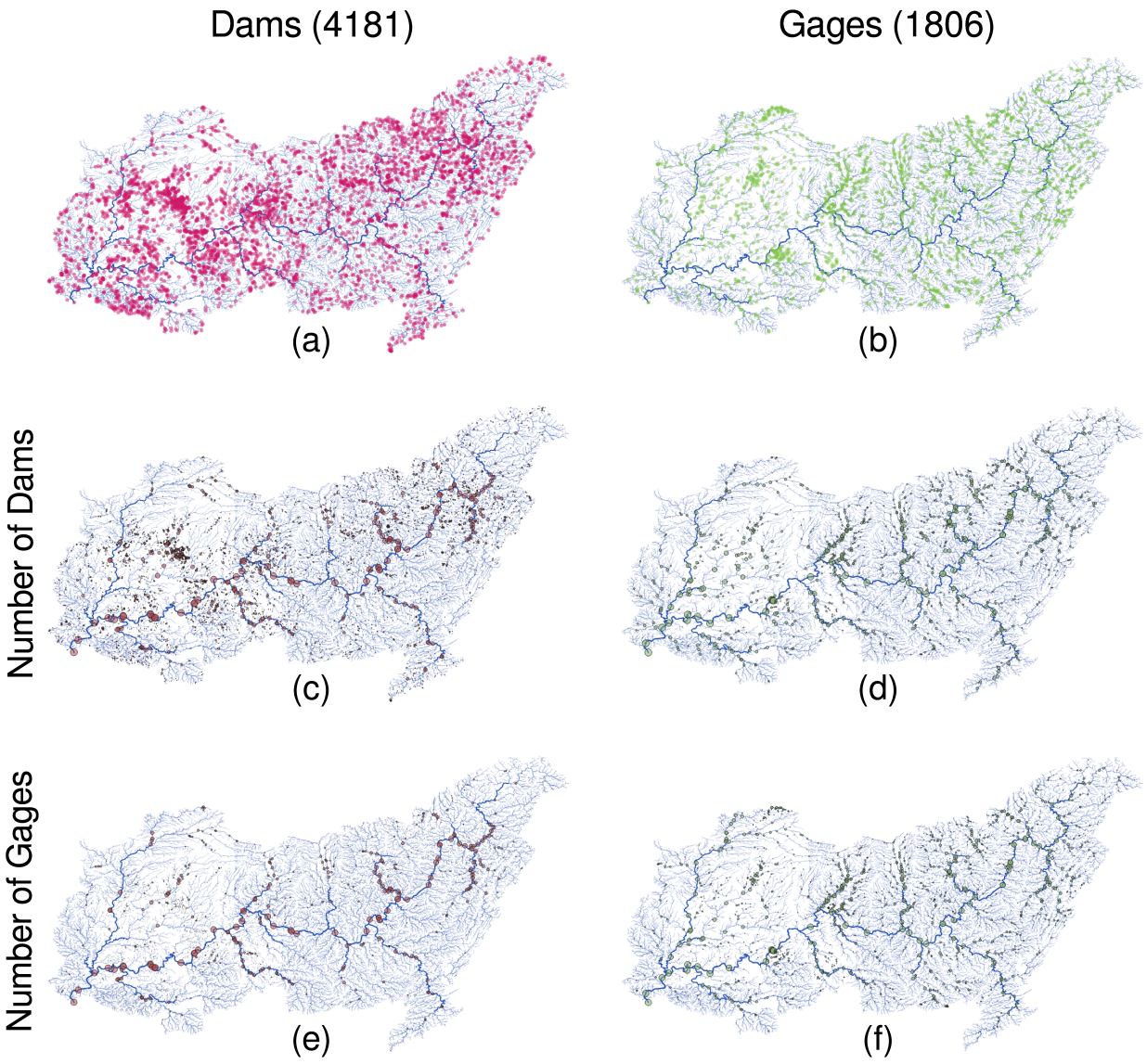


Figure 36: Gage/Dam Count

15.2.5. Extra: Counting Large Dams

The definition of large dam: https://www.icold-cigb.org/GB/dams/definition_of_a_large_dam.asp

- height of more than 15 meters,
- height between 5m and 15m impounding more than 3 million cubic meters.

Converting them to imperial units we get:

Metric	Imperial
15m	49.21ft
5m	16.40ft
3 x 10^6 m^3	810.71 acre feet

Now, we need to load the dam attributes from NID database.

```
network gis_load_attrs("data/ohio-river/nid-uniq.gpkg", "nidId")
node(is_dam).dam_height = float(nidHeight);
node(is_dam).dam_storage = float(nidStorage);

network count(nodes.is_dam)
network count(nodes.dam_height? & nodes.dam_storage?)
node(is_dam & (INDEX < 10)) array(dam_height, dam_storage)
```

Results:

```
4181
4181
{
  IL50499 = [56, 738700],
  IL00070 = [24, 153],
  IL00955 = [36, 366],
  IL40055 = [28, 47],
  IL00039 = [59, 633],
  IL00102 = [38, 1814]
}
```

Lot's of basins do not have basin area.

First, using the previous requirements for large dams:

```
node.large_dam = is_dam & ((dam_height > 49) | ((dam_height > 16) & (dam_storage >
811)));
network count(nodes.large_dam)
network count(nodes.large_dam) / count(nodes.is_dam)
```

Results:

```
1135
0.27146615642190863
```

This will allow us to run the same counting as before:

```
node<inp>.nldam = int(large_dam) + sum(inputs.nldam);
node(INDEX < 10) array(ndam, nldam)
```

Results:

```
{
  03399800 = [4181, 1135],
  IL50499 = [4181, 1135],
  IL00070 = [1, 0],
  IL00955 = [1, 0],
  IL40055 = [1, 0],
  IL00039 = [2, 2],
  03386500 = [1, 1],
  IL00102 = [1, 1],
  03386000 = [0, 0],
  03385500 = [0, 0]
}
```

Although not obivious, the numbers were quite large, so when I inspected the NID data, some dams seem to have very high height values that do not match.

Let's load basin area and flag any locations with more than 50ft of dam height, and less than 10 square miles of basin area.

```
node(is_dam & drainageArea?).basin_area = float(drainageArea);
network count(nodes.basin_area?)
node(! basin_area?).basin_area = nan;

node.flag = large_dam & ((dam_height > 50) & (basin_area < 10));
network count(nodes.flag)
node(flag & (INDEX < 1000)) array(dam_height, dam_storage, basin_area)
```

Results:

```
3766
353
{
  IL50593 = [108, 10790, 0.27],
  IL50678 = [110, 5250, 0.1],
  IL50066 = [55, 1087, 0.7000000000000001],
  IN00446 = [54, 7538, 4.75],
  IL00688 = [52, 3474, 3.3000000000000003],
  IN00439 = [56, 321, 0.52],
  IN03920 = [55, 39, 0.02],
  IN00505 = [56, 66, 0.03],
  IN00036 = [63, 1595, 1.2],
  IN00181 = [55, 1380, 4.7],
  IN00201 = [55, 2861, 4.69],
  IN00069 = [68, 6718, 5.8],
  IN03731 = [57, 128, 0.8200000000000001],
  IN03007 = [55, 29800, 0],
  IN00197 = [62, 3555, 1.82],
  IN00103 = [60, 1158, 0.36],
  IN00133 = [82, 13000, 9.34],
  IN00342 = [63, 198, 0.19],
  IN00565 = [53, 109, 0.1]
}
```

We don't have all basin areas, but from this we flagged around 300 dams. Looking at the values, `IL50678` basically says it has a dam with height of 110, but basin area of 0.1, which seems very unreasonable. To remove these from our identification process, let's add another category.

```
node.large_dam = is_dam & (((dam_height > 49) | ((dam_height > 16) & (dam_storage >
811))) & (basin_area > 10));
network count(nodes.large_dam)
network count(nodes.large_dam) / count(nodes.is_dam)
```

Results:

```
321
0.07677589093518297
```

The number of dams that are now categorized as large dams have been reduced significantly.

15.2.6. Extra: Looking at Main Stem of the River

Let's look at the values along the main stem. `LEVEL == 0` means the mainstem of the network.

```
node(LEVEL==0) array(ngage, ndam, nldam)
```

Results:

```
{
  03399800 = [1806, 4181, 1135],
  IL50499 = [1805, 4181, 1135],
  03384500 = [1801, 4169, 1129],
  IL50443 = [1799, 4164, 1129],
  03381700 = [1785, 4065, 1097],
  KY03060 = [1454, 3087, 905],
  03322420 = [1454, 3086, 904],
  IN03661 = [1451, 3067, 897],
  03322190 = [1451, 3061, 897],
  KY01059 = [1450, 3061, 897],
  03322000 = [1448, 3044, 894],
  IN04061 = [1388, 2822, 826],
  03304300 = [1388, 2821, 826],
  KY03059 = [1387, 2821, 826],
  03303500 = [1386, 2809, 823],
  KY01255 = [1383, 2771, 815],
  KY00842 = [1383, 2769, 814],
  IN03297 = [1383, 2768, 813],
  KY03058 = [1383, 2765, 813],
  03303280 = [1383, 2764, 812],
  IN03192 = [1382, 2764, 812],
  KY01272 = [1373, 2726, 807],
  03294600 = [1322, 2576, 789],
  KY00597 = [1318, 2572, 788],
  KY01022 = [1318, 2571, 788],
  03294500 = [1318, 2570, 788],
  KY03034 = [1316, 2552, 780],
  03293600 = [1316, 2551, 779],
  03293551 = [1315, 2551, 779],
  03293550 = [1314, 2551, 779],
  03293548 = [1313, 2551, 779],
  IN00643 = [1295, 2533, 775],
  KY03033 = [1220, 2287, 689],
  03277200 = [1220, 2286, 688],
  KY01215 = [1217, 2272, 685],
```

```
  OH01690 = [1111, 2112, 657],
  03255000 = [1091, 2087, 648],
  OH01350 = [1013, 1898, 616],
  03238680 = [1011, 1886, 612],
  KY03032 = [1010, 1886, 612],
  KY00938 = [1004, 1869, 609],
  KY01093 = [1003, 1868, 609],
  03238000 = [1003, 1867, 608],
  OH03181 = [1002, 1866, 607],
  03217200 = [883, 1700, 573],
  KY03031 = [875, 1692, 572],
  03216600 = [875, 1691, 571],
  03216000 = [868, 1680, 568],
  03206000 = [788, 1596, 513],
  WV05302 = [760, 1568, 503],
  03201500 = [622, 1335, 415],
  03160000 = [621, 1335, 415],
  OH00971 = [619, 1327, 411],
  WV05312 = [617, 1318, 408],
  WV05313 = [617, 1317, 407],
  WV05301 = [617, 1314, 407],
  03159870 = [617, 1313, 406],
  WV10702 = [609, 1296, 399],
  03159530 = [609, 1295, 398],
  03151000 = [564, 1186, 358],
  OH00943 = [563, 1185, 358],
  OH00939 = [563, 1184, 358],
  03150700 = [563, 1182, 357],
  WV07301 = [396, 842, 288],
  WV10301 = [393, 836, 285],
  03114280 = [393, 835, 284],
  03114275 = [392, 835, 284],
  WV05108 = [389, 827, 280],
  03113600 = [385, 805, 273],
  03112500 = [377, 798, 270],
  03111534 = [373, 769, 251],
  WV06908 = [372, 769, 251],
  03111520 = [372, 768, 250],
  03111515 = [371, 768, 250],
  OH03169 = [368, 742, 246],
  WV02901 = [362, 704, 227],
  03110690 = [362, 703, 226],
  OH03172 = [361, 703, 226],
  03110685 = [361, 702, 226],
  03108500 = [344, 661, 219],
  PA00128 = [343, 661, 219],
```

```
  03108490 = [343, 660, 218],
  PA00127 = [279, 528, 184],
  03086000 = [279, 527, 183],
  PA00126 = [276, 519, 181],
  03085730 = [276, 518, 180],
  PA01994 = [142, 265, 96],
  PA00120 = [141, 264, 96],
  03085000 = [141, 263, 95],
  03075070 = [100, 192, 73],
  03075000 = [98, 172, 64],
  PA00122 = [97, 172, 64],
  PA00123 = [94, 156, 60],
  PA01549 = [89, 135, 50],
  PA01265 = [89, 133, 49],
  PA00124 = [88, 121, 44],
  03072655 = [88, 120, 43],
  03072500 = [86, 118, 43],
  PA00125 = [45, 90, 36],
  03063000 = [45, 89, 35],
  WV06106 = [43, 78, 31],
  03062450 = [43, 77, 30],
  03062445 = [42, 77, 30],
  WV06107 = [40, 75, 29],
  WV06108 = [40, 74, 28],
  03062224 = [40, 73, 27],
  03062000 = [37, 69, 26],
  03061000 = [12, 37, 9],
  03059000 = [9, 20, 8],
  WV03336 = [8, 20, 8],
  03058975 = [8, 18, 8],
  03058500 = [7, 16, 7],
  WV04110 = [6, 6, 1],
  03058020 = [6, 5, 1],
  WV04111 = [5, 2, 1],
  03058006 = [5, 1, 1],
  WV04114 = [4, 1, 1],
  03058000 = [4, 0, 0],
  03057900 = [3, 0, 0],
  03057300 = [1, 0, 0]
}
```

If you want to be more accurate, we can load the `SiteName` from GIS file and match the node with “Ohio River” in its name with the lowest order to find the first Ohio River Node.

15.3. Earliest Dam Year

First load the network, and attributes

```
network load_file("data/ohio-river/ohio.network")
network gis_load_attrs("data/ohio-river/nid-uniq.gpkg", "nidId")

node.is_usgs = NAME match "^[0-9]+";
node.is_dam = !is_usgs;
network count(nodes.is_usgs)
network count(nodes.is_dam)
```

Results:

```
1806
4181
```

Counting the upstream dams again,

```
node<inp>.ndam = int(is_dam) + sum(inputs.ndam);
node.no_dam_us = ndam == 0;
network count(nodes.no_dam_us & nodes.is_usgs)
network count(nodes.no_dam_us & nodes.is_usgs) / count(nodes.is_usgs)
```

Results:

```
605
0.33499446290143964
```

We can see 33% of the USGS gages do not have dams upstream.

And for those that do, let's look at the construction year,

```
env.max_year = 9999; # todo test it with nan
node.dam_year = int(get_attr("yearCompleted", env.max_year));

node<inp>.dam_aff_yr = min_num(inputs.dam_aff_yr, dam_year);
node<inp>.dam_affected = dam_aff_yr < env.max_year;
```

```
network count(nodes.dam_affected & nodes.is_usgs)
network count(!nodes.no_dam_us & nodes.is_usgs)
network count(nodes.dam_affected & nodes.is_usgs) / count(!nodes.no_dam_us &
nodes.is_usgs)
```

Results:

```
1152
1201
0.9592006661115737
```

This shows we have the construction year for 95% of the USGS gages that were constructed after at least one upstream dams.

16. LaTeX Table

This example runs a task to generate latex table with tikz graphics. Similar method can be used to do generative programming (where you write code to write code for another program).

```
network load_str("
tenesse -> ohio
ohio -> mississippi
red -> mississippi
")
node.area = (1 + ORDER) * 100.0

env echo("\\begin{tabular}{llll}")
env echo("Connections & Node & Area & Name \\\\[2mm]")
node(output._?).out = output.INDEX
node echo(render("\\\\Node[{LEVEL}]\\{{INDEX}\\}\\{{INDEX}\\} & $N_{INDEX}$ &
{area:f(2)} & {_NAME:case(title)} \\\\\\\\[2mm]"))
env echo("\\end{tabular}
\\tikz[overlay, remember picture]{")
node(output._?).edge = echo(render("\\\\path[->] ({INDEX}) edge ({out});"))
env echo("}")
```

Results:

```
\begin{tabular}{llll}

Connections & Node & Area & Name \\[2mm]

\Node[0]{0}{0} & $N_0$ & 500.00 & Mississippi \\[2mm]
\Node[1]{1}{1} & $N_1$ & 200.00 & Red \\[2mm]
\Node[0]{2}{2} & $N_2$ & 300.00 & Ohio \\[2mm]
\Node[0]{3}{3} & $N_3$ & 200.00 & Tenesse \\[2mm]

\end{tabular}
\tikz[overlay, remember picture]{

\path[->] (1) edge (0);
\path[->] (2) edge (0);
\path[->] (3) edge (2);

}
```

The latex file contains the `Node` command as follows:

```


\usepackage{tikz}
\usetikzlibrary{tikzmark}

\newcommand{\Node}[3][0]{%
  \tikz[overlay,remember picture]{\draw (#1 + 0.5, 0.1) circle [radius=0.2] node
(#2) {#3};%
  }}
```

The result of the compilation will result in the following table:

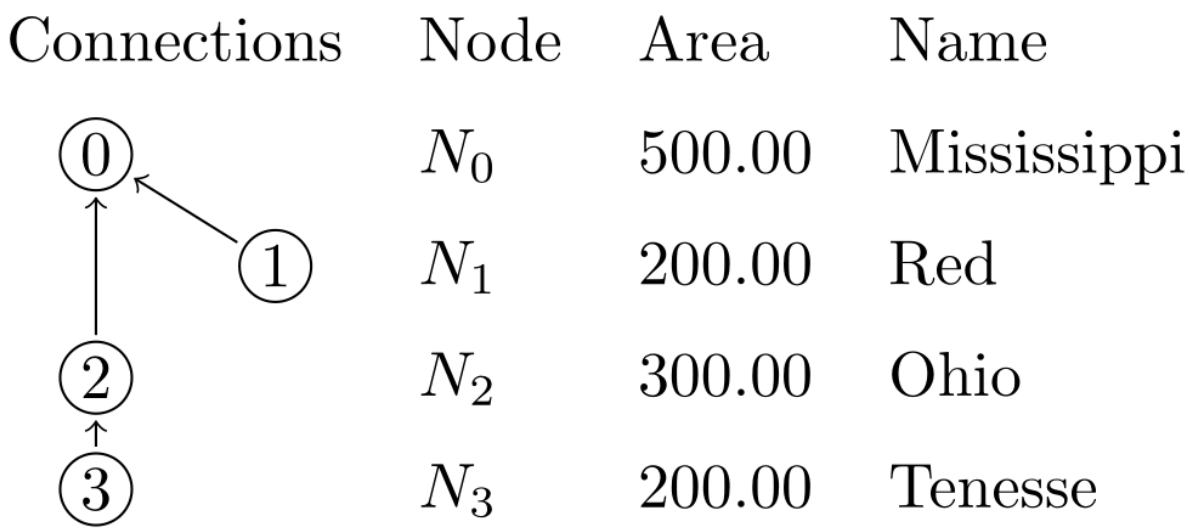


Figure 37: LaTeX Table

This is just an example, you can use this method to generate, markdown, latex, typst, html, and other markups as well. The syntax is a bit hard for LaTeX due to its use of `\` and `{`, both of which are special syntax of template system and need to be escaped. but for others it should be simpler.

This should become easier after the introduction of user defined functions.

The full latex source code is shown below:

```
\documentclass{standalone}
\usepackage{tikz}
\usetikzlibrary{tikzmark}

\newcommand{\Node}[3][0]{%
  \tikz[overlay,remember picture]{\draw (#1 + 0.5, 0.1) circle [radius=0.2] node
(#2) {#3};%
}}

\begin{document}
\begin{tabular}{llll}

Connections & Node & Area & Name \\[2mm]

\Node[0]{0}{0} & $N_0$ & 500.00 & Mississippi \\[2mm]
\Node[1]{1}{1} & $N_1$ & 200.00 & Red \\[2mm]
\Node[0]{2}{2} & $N_2$ & 300.00 & Ohio \\[2mm]
\Node[0]{3}{3} & $N_3$ & 200.00 & Tenesse \\[2mm]
```

```
\end{tabular}
\tikz[overlay, remember picture]{

\path[->] (1) edge (0);
\path[->] (2) edge (0);
\path[->] (3) edge (2);

}
\end{document}
```

Python Library

17. NADI Python (nadi-py)

17.1. NADI Python Library

This can be installed from `pypi` with `pip install nadi-py` command.

Then you can simply import and use it:

```
import nadi

net = nadi.Network.from_str("a -> b")
print([n.NAME for n in net.nodes])
```

The functions are available inside the `nadi.functions` submodule.

```
import nadi
import nadi.functions as fn

net = nadi.Network.from_str("a -> b")
fn.network.svg_save(net, "test.svg")
```

17.2. Combining the power of python and Task System

You can combine the power of python with task system using the `command` function from NADI Task System. Basically, you write your logic that cannot be written in nadi in python, you can use `nadi-py` if you need to parse network files, load attributes or call any other nadi functions. And you can pass the results of the python script at the end by simply printing it to the standard output.

Future work is under consideration to have a tight couple between the python and nadi system.

18. Differences with Task System

The difference from Task system is that now we use python syntax and the python functions. The environment from task system is no longer available, and the node functions are not automatically run in a loop.

We lose the advantages brought by the Domain Specific Programming Language, while gaining the flexibility and the well developed libraries of the python language.

Some examples showing how you'd have to write python codes from equivalent examples in the book are shown below.

18.1. Example 1: looping through the nodes

```
network load_str("a -> b\nc -> b")
node(output._?) echo(env.render("{i} -> {o}", i=node.INDEX, o=output.INDEX))
```

Results:

```
1 -> 0
2 -> 0
```

Equivalent Python:

```
import nadi

net = nadi.Network.from_str("a -> b\nc -> b")
for node in net.nodes:
    out = node.output()
  if out is None:
      continue
    print(f"{node.INDEX} -> {out.INDEX}")
```

Here the code for python is longer because it is general purpose and doesn't have the syntax tailored for network analysis like with NADI Task System.

18.2. Example 2: Skip execution when variable is absent

If we had to check for an attribute, then it becomes even more complicated.

```
node(somevar?) somefunc(somevar)
```

```
import nadi
import nadi.functions as fn

net = nadi.Network.from_str("a -> b\nc -> b")
for node in net.nodes():
  try:
      fn.node.somefun(node, node.somevar)
  except AttributeError:
      continue
```

In case of multiple variables being used, the `AttributeError` might catch all of them, further fine tuning in python could make the code far longer than in nadi.

19. Plugins

19.1. Plugins

Not only can you use `nadi-py` to write network based algorithms in python, you can also use it to write executable plugins that you can use to run analysis in python and feed it back to nadi system.

First thing to say about that is, you don't need `nadi-py` for writing python plugins, as they are run as a normal python scripts.

19.1.1. Example without using nadi-py

Here is an example task that calls python using the `command` function:

```
network load_file("scioto.network")

# load average streamflow from the csv file
# containing timeseries using python
node command("python area-and-streamflow.py {_NAME}")
```

```
# this just prints the attributes in csv format.
network print_attr_csv("INDEX", "area", "streamflow")
```

Here the `command` function takes a string template, renders it and runs it as a shell command for each node.

Our python script should have a way to read that node's name that we passed to the python command.

```
import sys
import pandas as pd

try:
    station = sys.argv[1]
except IndexError:
    print("Give station")
    exit(1)

df = pd.read_csv(f"data/streamflow/{station}.csv", header=None)
sf = df.iloc[:, 4]
sf.index = pd.to_datetime(df.iloc[:, 2])
daily = sf.resample('1d').mean()
counts = daily.groupby(daily.index.year).count()
counts.index.name = "datetime"
daily.index.name = "datetime"
annual = daily.groupby(daily.index.year).mean().loc[counts > 300]

print("nadi:var:sf_mean=", float(daily.mean()))

for year, flow in annual.items():
    print(f"nadi:var:sf_year_{year}={flow}")
```

Here the line `sys.argv[1]` reads the argument from command line (node's name in this case). And reads the data for that node. The ouput is printed with prefix `nadi:var:` which tells nadi to load as `key=val` pair for that node.

19.1.2. Example using nadi-py

The same example can be written using `nadi-py` so that the execution is very short (as it is being run as a network function instead of node function; `command` is a slow function as a new shell instance has to be created every time it is invoked).

Here we use the `command` network function and pass the network file as input. If your network has changed you can use `save_file network` function to save the network as a text file and then pass that instead.

```
network load_file("scioto.network")

# load average streamflow from the csv file
# containing timeseries using python
network command("python area-and-streamflow.py scioto.network")

# this just prints the attributes in csv format.
network print_attr_csv("INDEX", "area", "streamflow")
```

The corresponding python script now will look like this:

```
import sys
import pandas as pd
import nadi

try:
    network = sys.argv[1]
except IndexError:
    print("Give station")
    exit(1)

for node in nadi.Network(network).nodes():
  station = node.NAME
  df = pd.read_csv(f"data/streamflow/{station}.csv", header=None)
  sf = df.iloc[:, 4]
  sf.index = pd.to_datetime(df.iloc[:, 2])
  daily = sf.resample('1d').mean()
  counts = daily.groupby(daily.index.year).count()
  counts.index.name = "datetime"
  daily.index.name = "datetime"
  annual = daily.groupby(daily.index.year).mean().loc[counts > 300]

  print(f"nadi:var:{station}:sf_mean=", float(daily.mean()))

  for year, flow in annual.items():
    print(f"nadi:var:{station}:sf_year_{year}={flow}")
```

Here we load the network using `nadi-py`, and then loop through the node, and pass the variables back to nadi through stdout. We have to pass the node names with the `nadi:var:`

as this is being run for the whole network. Without the node name, it'll take the `key=val` pair as network attribute.

This should allow users to have a lot of flexibility in using python to do complex analysis and get the results back into nadi directly. You can also save the results of the python script into a file, and check if the file exists before running the command from nadi to save the redundant computations.

20. Examples

TODO: add examples from papers' case studies.

Plugin Developer Guide

21. Executable Plugins

Executable plugins are programs that can be called from terminal. The node `command` function, network `command` function and their families in the `command` plugin have the capacity to run external programs through the command line.

The inputs to the program is given through the command line arguments, while the output of the programs are read through the standard output of the program. This can be used to call different/same commands for nodes with arguments dependent on their attributes.

And the output from the programs are taken by reading their stdout (standard output). Any lines starting from `nadi:var:` (prefix) is considered a communication attempt with NADI Task System. Currently, you can set attribute values by providing `key=val` pairs after the prefix. The node function will set it for current node, and network function will set it for the network. Furthermore, in network function, you can add one more section after prefix to set node attributes. For example, `nadi:var:node1:value=12` will set the `value` attribute to `12` in the node named `node1` in the current network.

The executable plugin or commands are language agnostic, as long as the command is available to run from the parent shell they will be run.

To learn how to write code in your language to parse command line arguments refer to the Wikipedia page on Command Line Arguments

The following section shows example programs written in python and R that can interact with nadi in this way.

21.1. Python

Here is an example python script that can be called from nadi for each node. This script just reads a CSV file and passes the attributes to nadi, but more complicated programs can be written by the users.

First part is importing libraries and getting the arguments from nadi. The code below reads one string as a commandline argument and saves that into `station` variable.

```
import sys
import pandas as pd

try:
    station = sys.argv[1]
except IndexError:
    print("Give station")
    exit(1)
```

Then we can use any python logic with any libraries to do what we want. Here it reads the CSV and extracts values based on the station name. This is just an example, but you can load different csv files for each station and do a lot of analysis before sending those variables to nadi.

```
import sys
import pandas as pd

try:
    station = sys.argv[1]
except IndexError:
    print("Give station")
    exit(1)

df = pd.read_csv(f"data/streamflow/{station}.csv", header=None)
sf = df.iloc[:, 4]
sf.index = pd.to_datetime(df.iloc[:, 2])
sf = sf.resample('1d').mean()
```

Once we have our variables from analysis, we can simply print them with `nadi:var:` prefix so that nadi knows they are the variables it should read and load into each node.

```
import sys
import pandas as pd

try:
    station = sys.argv[1]
except IndexError:
    print("Give station")
    exit(1)

df = pd.read_csv(f"data/streamflow/{station}.csv", header=None)
sf = df.iloc[:, 4]
```

```
 sf.index = pd.to_datetime(df.iloc[:, 2])
 sf = sf.resample('1d').mean()
print("nadi:var:sf_mean=", float(sf.mean()))

for year, flow in sf.groupby(sf.index.year).mean().items():
    print(f"nadi:var:sf_year_{year}={flow}")
```

Now we can call this script from inside the nadi tasks system like the following, assuming the python file is saved as `streamflow.py`.

```
node command("python streamflow.py {_NAME}")
```

If you want to know what the template will be rendered as, use `render` function, and if you want to check whether it exists or not, you can use `exists` function.

21.2. RScript

Similar to most programming languages R can also read command line arguments when ran with `RScript` command instead of `R`.

For example if you run the following script in a file called `test.r` and ran it with command `Rscript test.r some args 2`, you get the output of `[1] "some" "args" "2"`

```
args <- commandArgs(trailingOnly = TRUE)
print(args)
```

So you can use the same method like in python to pass arguments, do analysis and pass it back using the `cat` function in r as shown below. `cat` function avoids printing the `[1]` type indices to the stdout.

```
cat(sprintf("nadi:var:this_val=%d\n", 1200))
```

22. Compiled Plugins

As it is not possible to forsee all the use cases in advance, the nadi software can be easily extended (easy being an relative term) to account for different use cases.

The program can load compiled shared libraries (`.dll` in windows, `.so` in linux, and `.dylib` on mac). Since they are shared libraries compiled into binaries, any programming languages can be used to generate those. So far, the `nadi_core` library is available for `Rust` only. Using that, plugins can be written and those functions can be made available from the system.

NADI core library automatically loads:

- internal plugins if feature `functions` is used in `nadi_core` to compile it,
- external plugins in the directories inside the `NADI_PLUGIN_DIRS` environmental variables. The plugins must be compiled using the same `nadi_core` version and must have the same internal ABI for data types.

The syntax for functions in plugins are same for internal and external plugins. While the way to register the plugin differ slightly.

The difference between the internal and external plugins are that, internal plugins are compiled with the `nadi_core` and come with the program, while external plugins are separately compiled and loaded through dynamic libraries.

The methods for writing the plugins are the same, except at the top level: to export plugins, you have to use [`nadi_core::nadi_plugin::nadi_plugin`] macro for external plugins while [`nadi_core::nadi_plugin::nadi_internal_plugin`] for internal ones.

In the next sections we will go in detail about how to write plugins and load them in nadi.

22.1. Internal Plugins

Internal plugins come with the nadi system. They are only modified between the different versions of NADI.

The internal plugins provide core functionality of the Task system like data conversion, parsing network/attribute files, logical operations, template rendering, etc.

Future planned internal plugin functions can be found in nadi-futures repository. Which in itself is an external plugin.

22.2. External Plugins

22.2.1. External Plugins

External plugins are plugins that are their own separate programs that compile to a shared library. The shared library has information about the name of the plugin, the functions that are available, as well as the bytecode required to run the functions.

You have to use the `nadi_core` library and the macros available there to make the plugins. Although it might be possible to write it without the macros (an example is provided), it is strongly discouraged. The example only serves as a way to demonstrate the inner working of the external plugins.

Some examples of external plugins are given in the `nadi-plugins-rust` repository.

An example of a complex external plugin can be found in the `gis` plugin from `nadi-gis` repository.

22.2.1.1. Steps to create a Plugin

`nadi` CLI tool has a function that can generate a plugin template. Simply run the `nadi` command with `--new-plugin` flag.

```
nadi --new-plugin <plugin-name>
```

This will create a directory with plugin's name with `Cargo.toml` and `src/lib.rs` with some sample codes for plugin functions. You can then edit them as per your need.

The generated files using `nadi --new-plugin sample` look something like this:

Cargo.toml:

```
[package]
name = "sample"
version = "0.1.0"
edition = "2021"

[lib]
crate-type = ["cdylib"]

# make sure you use the same version of nadi_core, your nadi-system is in
```

```
[dependencies]
abi_stable = "0.11.3"
nadi_core = "0.7.0"
```

src/lib.rs:

```
use nadi_core::nadi_plugin::nadi_plugin;

#[nadi_plugin]
mod sample {
    use nadi_core::prelude::*;

    /// The macros imported from nadi_plugin read the rust function you
    /// write and use that as a base to write more core internally that
    /// will be compiled into the shared libraries. This means it'll
    /// automatically get the argument types, documentation, mutability,
    /// etc. For more details on what they can do, refer to nadi book.
    use nadi_core::nadi_plugin::{env_func, network_func, node_func};

    /// Example Environment function for the plugin
    ///
    /// You can use markdown format to write detailed documentation for the
    /// function you write. This will be availble from nadi-help.
    #[env_func(pre = "Message: ")]
    fn echo(message: String, pre: String) -> String {
        format!("{}{}", pre, message)
    }

    /// Example Node function for the plugin
    #[node_func]
    fn node_name(node: &NodeInner) -> String {
        node.name().to_string()
    }

    /// Example Network function for the plugin
    ///
    /// You can also write docstrings for the arguments, this syntax is not
    /// a valid rust syntax, but our macro will read those docstrings, saves
    /// it and then removes it so that rust does not get confused. This means
    /// You do not have to write separate documentation for functions.
    #[network_func]
    fn node_first_with_attr(
        net: &Network,
        /// Name of the attribute to search
        attrname: String,
```

```
    ) -> Option<String> {
        for node in net.nodes() {
            let node = node.lock();
            if node.attr_dot(&attrname).is_ok() {
                return Some(node.name().to_string());
            }
        }
        None
    }
}
```

The plugin can be compiled with the `cargo build` or `cargo build --release` command, it'll generate the shared library in the `target/debug` or `target/release` folder. You can simply copy it to directory in `NADI_PLUGIN_DIRS` for it to be loaded.

22.3. Functions

Plugin functions are very close to normal rust functions, with extra syntax for the function arguments, and limited function argument and return types.

22.3.1. Function Types

There are 3 function types:

- environment
- node
- network

the macro used for each function type are availabel from `nadi_core::nadi_plugin`. All the macro take optional list of `key = value` pairs that can act like default arguments to the functions while called from the task system.

These macro will read the rust function and generate the necessary plugin code, function signature, documentation, and will even save the original code so that users can browse it through the `nadi-help`.

22.3.2. Function Arguments

There are 5 types of function arugments, that are denoted by the following attributes

macro attr	Type	Supported Types
	Node/Network	`&/& mut` + `NodeInner/Network`
	Normal arguments	`T: FromAttribute`
#[relaxed]	Relaxed arguments	`T: FromAttributeRelaxed`
#[args]	Positional Arguments List	`&[Attribute]`
#[kwargs]	Keyword Arguments AttrMap	`&AttrMap`

Users can not provide the argument `Node/Network` for node/network function as it is automatically provided based on the context.

Furthermore, there are required and optional arguments. And users can optionally omit the arguments that are of type `Option<T>`, or have default value in the macro (e.g. `safe = false` in the codes below).

For now, the function arguments except the `Node` or `Network` cannot be `mut`. But they can be reference of `T` if `T` satisfies the trait constraints, for example, instead of `Vec<String>`, it can be `&[String]`. But because the function context is evaluated for each node/network, there is no optimization by using the references.

22.3.3. Return Types

Function Return can be empty, an attribute value, or an error. When a function returns an error, the execution is halted. When it doesn't return a value and an assignment is performed, it will error as well.

The return type of the function should implement `Into<FunctinRet>`, refer to the documentation for `nadi_core::functions::FunctionRet` to see what types implement it. You can also implement that for your own types.

You can simply use any type that satisfy the trait requirement mentioned above as a function return and the nadi macros will convert them automatically for you.

22.3.4. Verbosity

In future versions the functions will also get a flag that will let them know how verbose the functions can be. This will also come with a way to pass progress and other information while the function is still running.

22.3.5. Examples

Refer to the nadi_core, and other plugin repositories for sample codes for plugin functions as they are always up to date with the current version.

Here is an example containing render function that is available on all function types.

```rust
/// Render the template based on the given attributes
///
/// For more details on the template system. Refer to the String
/// Template section of the NADI book.
#[env_func(safe = false)]
fn render(
    /// String template to render
    template: &Template,
    #[kwargs] keyval: &AttrMap,
    /// if render fails keep it as it is instead of exiting
    safe: bool,
) -> Result<String, String> {
    let text = if safe {
        keyval
            .render(template)
            .unwrap_or_else(|_| template.original().to_string())
    } else {
        keyval.render(template).map_err(|e| e.to_string())?
    };
    Ok(text)
}
```

```rust
/// Render the template based on the node attributes
///
/// For more details on the template system. Refer to the String
/// Template section of the NADI book.
#[node_func(safe = false)]
fn render(
    node: &NodeInner,
    /// String template to render
    template: &Template,
    /// if render fails keep it as it is instead of exiting
    safe: bool,
) -> Result<String, String> {
    let text = if safe {
        node.render(template)
            .unwrap_or_else(|_| template.original().to_string())
```

```
        } else {
            node.render(template).map_err(|e| e.to_string())?
        };
        Ok(text)
    }
```

```
    /// Render from network attributes
    #[network_func(safe = false)]
    fn render(
        network: &Network,
        /// Path to the template file
        template: &Template,
        /// if render fails keep it as it is instead of exiting
        safe: bool,
    ) -> Result<String, String> {
        let text = if safe {
            network
                .render(template)
                .unwrap_or_else(|_| template.original().to_string())
        } else {
            network.render(template).map_err(|e| e.to_string())?
        };
        Ok(text)
    }
```

22.3.6. Environment Functions

Environment functions are like any normal function on programming languages that take arguments and run code. In NADI environment functions can be called from any scope. For example, if a node function and environement function share the same name, then in a node task node function is called, but in network task env function is called.

Environment functions are denoted in the plugins with `#[env_func]` macro. All the arguments this function takes need to be provided by user or through default values.

Here is an example of a environment function `and` in plugin `logic`.

```
    /// Boolean and
    #[env_func]
    fn and(
        /// List of attributes that can be cast to bool
        #[args]
```

```
        conds: &[Attribute],
    ) -> bool {
        let mut ans = true;
        for c in conds {
            ans = ans && bool::from_attr_relaxed(c).unwrap();
        }
        ans
    }
```

This function can be called inside the task system in different context like follows:

```
env and(true, 12)
env.something = false
env and(something, true) == (something & true)

network and(what?, and(true, true))
```

Results:

```
true
true
false
```

22.3.7. Node Functions

Node functions are run for each node in the network (or a selected group of nodes). Hence, it takes the first argument as `& NodeInner` or `& mut NodeInner` depending on the purpose of the function. Immutable functions can be called from any place, while mutable functions can only be called once on the outermost layer on the task.

Other arguments and the return types for node functions are the same as the environement functions.

22.3.8. Network Functions

Network functions, like node functions take `&Network` or `& mut Network` as the first argument. It has the same restrictions as the env/node functions for the arguments and the return types.

List of Plugin Functions

23. Internal Plugins

23.1. Internal Plugins

There are some plugins that are provided with the `nadi_core` library. They are part of the library, so users can directly use them.

For example in the following tasks file, the functions that are highlighted are functions available from the core plugins. Other functions need to be loaded from plugins.

```
# sample .tasks file which is like a script with functions
node<inputsfirst> print_attrs("uniqueID")
node show_node()
network save_graphviz("/tmp/test.gv")
node<inputsfirst>.cum_val = node.val + sum(inputs.cum_val);

node[WV04113,WV04112,WV04112] print_attr_toml("testattr2")
node render("{NAME} {uniqueID} {_Dam_Height_(Ft)?}")
node list_attr("; ")
# some functions can take variable number of inputs
network calc_attr_errors(
    "Dam_Height_(Ft)",
    "Hydraulic_Height_(Ft)",
    "rmse", "nse", "abserr"
)
node sum_safe("Latitude")
node<inputsfirst> render("Hi {SUM_ATTR}")
# multiple line for function arguments
network save_table(
  "test.table",
  "/tmp/test.tex",
  true,
  radius=0.2,
  start = 2012-19-20,
  end = 2012-19-23 12:04
  )
node.testattr = 2
node set_attrs_render(testattr2 = "{testattr:calc(+2)}")
node[WV04112] render("{testattr} {testattr2}")
```

```
# here we use a complicated template that can do basic logic handling
node set_attrs_render(
    testattr2 = "=(if (and (st+has 'Latitude) (> (st+num 'Latitude) 39)) 'true
'false)"
)
# same thing can be done if you need more flexibility in variable names
node load_toml_string(
    "testattr2 = =(if (and (st+has 'Latitude) (> (st+num 'Latitude) 39)) 'true
'false)"
)
# selecting a list of nodes to run a function
node[
  # comment here?
    WV04113,
    WV04112
] print_attr_toml("testattr2")
# selecting a path
node[WV04112 -> WV04113] render("=(> 2 3)")
```

23.2. Attributes

23.2.1. Env Functions

23.2.1.1. strmap

```
env attributes.strmap(
    attr: '& str',
    attrmap: '& AttrMap',
    default: 'Option < Attribute >'
)
```

23.2.1.1.1. Arguments

- `attr: '& str'` => Value to transform the attribute
- `attrmap: '& AttrMap'` => Dictionary of key=value to map the data to
- `default: 'Option < Attribute >'` => Default value if key not found in `attrmap`

map values from the attribute based on the given table

```
env.val = strmap("Joe", {Dave = 2, Joe = 20});
env assert_eq(val, 20)
env.val2 = strmap("Joe", {Dave=2}, default = 12);
env assert_eq(val2, 12)
```

23.2.1.2. parse_attr

```
env attributes.parse_attr(toml: '& str')
```

23.2.1.2.1. Arguments

- `toml: '& str'` => String to parse into attribute

Parse attribute from string

```
env assert_eq(parse_attr("true"), true)
env assert_eq(parse_attr("123"), 123)
env assert_eq(parse_attr("12.34"), 12.34)
env assert_eq(parse_attr("\"my value\""), "my value")
env assert_eq(parse_attr("1234-12-12"), 1234-12-12)
```

23.2.1.3. parse_attrmap

```
env attributes.parse_attrmap(toml: 'String')
```

23.2.1.3.1. Arguments

- `toml: 'String'` => String to parse into attribute

Parse attribute map from string

```
env assert_eq(parse_attrmap("y = true"), {y = true})
env assert_eq(parse_attrmap(
"x = [1234-12-12, true]"),
{x = [1234-12-12, true]}
)
```

23.2.1.4. get

```
env attributes.get(
    parent: 'Attribute',
    index: 'Attribute',
    default: 'Option < Attribute >'
)
```

23.2.1.4.1. Arguments

- `parent: 'Attribute'` => Array or AttrMap Attribute to index
- `index: 'Attribute'` => Index value (Integer for Array, String for AttrMap)
- `default: 'Option < Attribute >'` => Default value if the index is not present

get the choosen attribute from Array or AttrMap

```
env.some_ar = ["this", 12, true];
env.some_am = {x = "this", y = [12, true]};
env assert_eq(get(some_ar, 0), "this")
env assert_eq(get(some_ar, 2), true)
env assert_eq(get(some_am, "x"), "this")
env assert_eq(get(some_am, "y"), [12, true])
```

23.2.1.5. powi

```
env attributes.powi(value: 'f64', power: 'i64')
```

23.2.1.5.1. Arguments

- `value: 'f64'` => base value
- `power: 'i64'` =>

Integer power

```
env assert_eq(powi(10.0, 2), 100.0)
```

23.2.1.6. powf

```
env attributes.powf(value: 'f64', power: 'f64')
```

23.2.1.6.1. Arguments

- `value: 'f64'` => base value
- `power: 'f64'` =>

Float power

```
env assert_eq(powf(100.0, 0.5), 10.0)
```

23.2.1.7. exp

```
env attributes.exp(value: 'f64')
```

23.2.1.7.1. Arguments

- `value: 'f64'` =>

Exponential

```
env assert_eq(log(exp(5.0)), 5.0)
```

23.2.1.8. sqrt

```
env attributes.sqrt(value: 'f64')
```

23.2.1.8.1. Arguments

- `value: 'f64'` =>

Square Root

```
env assert_eq(sqrt(25.0), 5.0)
```

23.2.1.9. log

```
env attributes.log(value: 'f64', base: 'Option < f64 >')
```

23.2.1.9.1. Arguments

- `value: 'f64' =>`
- `base: 'Option < f64 >' =>`

Logarithm of a value, natural if base not given

```
env assert_eq(log(exp(2.0)), 2.0)
env assert_eq(log(2.0, 2.0), 1.0)
```

23.2.1.10. float_div

```
env attributes.float_div(value1: 'f64', value2: 'f64')
```

23.2.1.10.1. Arguments

- `value1: 'f64'` => numerator
- `value2: 'f64'` => denominator

Float Division (same as / operator)

```
env assert_eq(float_div(10.0, 2), 10.0 / 2)
```

23.2.1.11. float_mult

```
env attributes.float_mult(value1: 'f64', value2: 'f64')
```

23.2.1.11.1. Arguments

- `value1: 'f64'` => numerator
- `value2: 'f64'` => denominator

Float Multiplication (same as * operator)

```
env assert_eq(float_mult(5.0, 2), 5.0 * 2)
```

23.2.2. Node Functions

23.2.2.1. load_attrs

```
node attributes.load_attrs(filename: 'PathBuf')
```

23.2.2.1.1. Arguments

- `filename: 'PathBuf'` => Template for the filename to load node attributes from

Loads attrs from file for all nodes based on the given template

23.2.2.1.2. Arguments

- `filename`: Template for the filename to load node attributes from
- `verbose`: print verbose message

The template will be rendered for each node, and that filename from the rendered template will be used to load the attributes.

23.2.2.1.3. Errors

The function will error out in following conditions:

- Template for filename is not given,
- The template couldn't be rendered,
- There was error loading attributes from the file.

23.2.2.2. print_all_attrs

```
node attributes.print_all_attrs()
```

23.2.2.2.1. Arguments

Print all attrs in a node

No arguments and no errors, it'll just print all the attributes in a node with `node::attr=val` format, where,

- node is node name
- attr is attribute name

- val is attribute value (string representation)

23.2.2.3. print_attrs

```
node attributes.print_attrs(*attrs, name: 'bool' = false)
```

23.2.2.3.1. Arguments

- `*attrs` =>
- `name: 'bool' = false` =>

Print the given node attributes if present

23.2.2.3.2. Arguments

- attrs,... : list of attributes to print
- name: Bool for whether to show the node name or not

23.2.2.3.3. Error

The function will error if

- list of arguments are not `String`
- the `name` argument is not Boolean

The attributes will be printed in `key=val` format.

23.2.2.4. set_attrs

```
node attributes.set_attrs(**attrs)
```

23.2.2.4.1. Arguments

- `**attrs` => Key value pairs of the attributes to set

Set node attributes

Use this function to set the node attributes of all nodes, or a select few nodes using the node selection methods (path or list of nodes)

23.2.2.4.2. Error

The function should not error.

23.2.2.4.3. Example

Following will set the attribute `a2d` to `true` for all nodes from `A` to `D`

```
network load_str("A -> B\n B -> D");
node[A -> D] set_attrs(a2d = true)
```

This is equivalent to the following:

```
node[A->D].a2d = true;
```

23.2.2.5. get_attr

```
node attributes.get_attr(attr: '& str', default: 'Option < Attribute >')
```

23.2.2.5.1. Arguments

- `attr: '& str'` => Name of the attribute to get
- `default: 'Option < Attribute >'` => Default value if the attribute is not found

Retrive attribute

```
network load_str("A -> B\n B -> D");
node assert_eq(get_attr("NAME"), NAME);
```

23.2.2.6. has_attr

```
node attributes.has_attr(attr: '& str')
```

23.2.2.6.1. Arguments

- `attr: '& str'` => Name of the attribute to check

Check if the attribute is present

```
network load_str("A -> B\n B -> D");
node.x = 90;
node assert(has_attr("x"))
node assert(!has_attr("y"))
```

23.2.2.7. first_attr

```
node attributes.first_attr(attrs: '& [String]', default: 'Option < Attribute >')
```

23.2.2.7.1. Arguments

- `attrs: '& [String]'` => attribute names
- `default: 'Option < Attribute >'` => Default value if not found

Return the first Attribute that exists

This is useful when you have a bunch of attributes that might be equivalent but are using different names. Normally due to them being combined from different datasets.

```
network load_str("A -> B\n B -> D");
node.x = 90;
node assert_eq(first_attr(["y", "x"]), 90)
node assert_eq(first_attr(["x", "NAME"]), 90)
```

23.2.2.8. set_attrs_ifelse

```
node attributes.set_attrs_ifelse(cond: 'bool', **values)
```

23.2.2.8.1. Arguments

- `cond: 'bool'` => Condition to check
- `**values` => key = [val1, val2] where key is set as first if `cond` is true else second

if else condition with multiple attributes

```
network load_str("a -> b");
env.some_condition = true;
node set_attrs_ifelse(
env.some_condition,
```

```
val1 = [1, 2],
val2 = ["a", "b"]
);
env assert_eq(nodes.val1, [1, 1])
env assert_eq(nodes.val2, ["a", "a"])
```

This is equivalent to using the if-else expression directly,

```
node.val1 = if (env.some_condition) {1} else {2};
env assert_eq(nodes.val1, [1, 1])
```

Furthermore if-else expression will give a lot more flexibility than this function in normal use cases. But this function is useful when you have to do something in a batch.

23.2.2.9. set_attrs_render

```
node attributes.set_attrs_render(**kwargs)
```

23.2.2.9.1. Arguments

- `**kwargs` => key value pair of attribute to set and the Template to render

Set node attributes based on string templates

This renders the template for each node, then it sets the values from the rendered results.

```
network load_str("a -> b");
node set_attrs_render(val1 = "Node: {_NAME}");
node[a] assert_eq(val1, "Node: a")
```

23.2.2.10. load_toml_render

```
node attributes.load_toml_render(toml: '& Template', echo: 'bool' = false)
```

23.2.2.10.1. Arguments

- `toml: '& Template'` => String template to render and load as toml string
- `echo: 'bool' = false` => Print the rendered toml or not

Set node attributes by loading a toml from rendered template

This function will render a string, and loads it as a toml string. This is useful when you need to make attributes based on some other variables that you can combine using the string template system.

In most cases it is better to use the string manipulation functions and other environmental functions to get new attribute values to set.

```
network load_str("a -> b");
node load_toml_render("label = \\\"Node: {_NAME}\\\"")
node assert_eq(label, render("Node: {_NAME}"))
```

23.2.3. Network Functions

23.2.3.1. set_attrs

```
network attributes.set_attrs(**attrs)
```

23.2.3.1.1. Arguments

- `**attrs` => key value pair of attributes to set

Set network attributes

23.2.3.1.2. Arguments

- `key=value` - Kwargs of attr = value

```
network set_attrs(val = 23.4)
network assert_eq(val, 23.4)
```

23.2.3.2. set_attrs_render

```
network attributes.set_attrs_render(**kwargs)
```

23.2.3.2.1. Arguments

- `**kwargs` => Kwargs of attr = String template to render

Set network attributes based on string templates

It will set the attribute as a String

```
network.val = 23.4
network set_attrs_render(val2 = "{val}05")
network assert_eq(val2, "23.405")
```

23.3. Command

23.3.1. Node Functions

23.3.1.1. command

```
node command.command(
    cmd: '& Template',
    verbose: 'bool' = true,
    echo: 'bool' = false
)
```

23.3.1.1.1. Arguments

- `cmd: '& Template'` => String Command template to run
- `verbose: 'bool' = true` => Show the rendered version of command, and other messages
- `echo: 'bool' = false` => Echo the stdout from the command

Run the given template as a shell command.

Run any command in the shell. The standard output of the command will be consumed and if there are lines starting with `nadi:var:` and followed by `key=val` pairs, it'll be read as new attributes to that node.

For example if a command writes `nadi:var:name="Joe"` to stdout, then the for the current node the command is being run for, `name` attribute will be set to `Joe`. This way, you can write your scripts in any language and pass the values back to the NADI system.

It will also print out the new values or changes from old values, if `verbose` is true.

23.3.1.1.2. Errors

The function will error if,

- The command template cannot be rendered,
- The command cannot be executed,
- The attributes from command's stdout cannot be parsed properly

```
network load_str("a -> b");
node command("echo 'nadi:var:sth={NAME}'");
node assert_eq(sth, NAME)
```

23.3.1.2. run

```
node command.run(
    command: '& str',
    inputs: '& str',
    outputs: '& str',
    verbose: 'bool' = true,
    echo: 'bool' = false
)
```

23.3.1.2.1. Arguments

- `command: '& str'` => Node Attribute with the command to run
- `inputs: '& str'` => Node attribute with list of input files
- `outputs: '& str'` => Node attribute with list of output files
- `verbose: 'bool' = true` => Print the command being run
- `echo: 'bool' = false` => Show the output of the command

Run the node as if it's a command if inputs are changed

This function will not run a command node if all outputs are older than all inputs. This is useful to networks where each nodes are tasks with input files and output files.

23.3.2. Network Functions

23.3.2.1. parallel

```
network command.parallel(
    cmd: '& Template',
    workers: 'i64' = 16,
    verbose: 'bool' = true,
    echo: 'bool' = false
)
```

23.3.2.1.1. Arguments

- `cmd: '& Template'` => String Command template to run
- `workers: 'i64' = 16` => Number of workers to run in parallel
- `verbose: 'bool' = true` => Print the command being run
- `echo: 'bool' = false` => Show the output of the command

Run the given template as a shell command for each nodes in the network in parallel.

Other than parallel execution this is same as the `node` function `command`

```
network load_str("a -> b");
network parallel("echo 'nadi:var:sth={NAME}'");
node assert_eq(sth, NAME)
```

23.3.2.2. command

```
network command.command(
    cmd: 'Template',
    verbose: 'bool' = true,
    echo: 'bool' = false
)
```

23.3.2.2.1. Arguments

- `cmd: 'Template'` => String Command template to run
- `verbose: 'bool' = true` => Print the command being run
- `echo: 'bool' = false` => Show the output of the command

Run the given template as a shell command.

Run any command in the shell. The standard output of the command will be consumed and if there are lines starting with `nadi:var:` and followed by `key=val` pairs, it'll be read as new attributes to the network. If you want to pass node attributes add node name with `nadi:var:name:` as the prefix for `key=val`.

See `node command.command` for more details as they have the same implementation

The examples below run `echo` command to set the variables, you can use any command that are scripting languages (python, R, Julia, etc) or individual programs.

```
network load_str("a -> b");
network command("echo 'nadi:var:sth=123'");
network assert_eq(sth, 123)
network command("echo 'nadi:var:a:sth=123'");
node[a] assert_eq(sth, 123)
```

23.4. Connections

23.4.1. Env Functions

23.4.1.1. root_node

```
env connections.root_node()
```

23.4.1.1.1. Arguments

default name used for ROOT node of the network

23.4.2. Network Functions

23.4.2.1. load_file

```
network connections.load_file(file: 'PathBuf', append: 'bool' = false)
```

23.4.2.1.1. Arguments

- `file: 'PathBuf'` => File to load the network connections from
- `append: 'bool' = false` => Append the connections in the current network

Load the given file into the network

This replaces the current network with the one loaded from the file.

23.4.2.2. load_str

```
network connections.load_str(contents: '& str', append: 'bool' = false)
```

23.4.2.2.1. Arguments

- `contents: '& str'` => String containing Network connections
- `append: 'bool' = false` => Append the connections in the current network

Load network from the given string

This replaces the current network with the one loaded from the string.

```
network load_str("a -> b");
env assert_eq(nodes.NAME, ["b", "a"])
```

23.4.2.3. load_edges

```
network connections.load_edges(edges: '& [(String, String)]', append: 'bool' =
false)
```

23.4.2.3.1. Arguments

- `edges: '& [(String, String)]'` => String containing Network connections
- `append: 'bool' = false` => Append the connections in the current network

Load the given edges as a network

This replaces the current network with the one loaded from the file.

```
network load_edges([["a", "b"], ["b", "c"]]);
env assert_eq(nodes.NAME, ["c", "b", "a"])
```

23.4.2.4. subset

```
network connections.subset(filter: '& [bool]', keep: 'bool' = true)
```

23.4.2.4.1. Arguments

- `filter: '& [bool]'` =>
- `keep: 'bool' = true` => Keep the selected nodes (false = removes the selected)

Take a subset of network by only including the selected nodes

```
network load_str("a -> b\n b->c");
node[a->b].sth = true;
node[c].sth = false;
network subset(nodes.sth);
env assert_eq(nodes.NAME, ["b", "a"])
```

23.4.2.5. save_file

```
network connections.save_file(
    file: 'PathBuf',
    quote_all: 'bool' = true,
    graphviz: 'bool' = false
)
```

23.4.2.5.1. Arguments

- `file: 'PathBuf'` => Path to the output file
- `quote_all: 'bool' = true` => quote all node names; if false, doesn't quote valid identifier names
- `graphviz: 'bool' = false` => wrap the network into a valid graphviz file

Save the network into the given file

For more control on graphviz file writing, use `save_graphviz` from `graphviz` plugin instead.

23.4.2.6. subset_from

```
network connections.subset_from(node: '& str')
```

23.4.2.6.1. Arguments

- `node: '& str' =>`

Take a subset of network by taking the given node as new outlet

```
network load_str("a -> b\n b->c\n x -> y");
network subset_from("b")
env assert_eq(nodes.NAME, ["b", "a"])
```

23.4.2.7. subset_largest

```
network connections.subset_largest(node: '& str' = "*ROOT*")
```

23.4.2.7.1. Arguments

- `node: '& str' = "*ROOT*" =>`

Take a subset of network by only including the largest blob of connected nodes

When you load a network that have disconnected nodes, nadi includes a ROOT note by default and collects all the outlets as inputs to that node. This function allows you to filter out all the nodes except the one belonging to the largest connected network (number of nodes). Alternatively, you can also use ORDER and other logic in the task system to do that.

If your network doesn't have a root node, then it'll just keep the network as it is.

```
network load_str("a -> b\n b->c\n x -> y");
network subset_largest()
env assert_eq(nodes.NAME, ["c", "b", "a"])
```

23.5. Core

23.5.1. Env Functions

23.5.1.1. count

```
env core.count(vars: '& [bool]')
```

23.5.1.1.1. Arguments

- `vars: '& [bool]' =>`

Count the number of true values in the array

```
env assert_eq(count([true, false, true, false]), 2)
```

23.5.1.2. type_name

```
env core.type_name(value: 'Attribute', recursive: 'bool' = false)
```

23.5.1.2.1. Arguments

- `value: 'Attribute'` => Argument to get type
- `recursive: 'bool' = false` => Recursively check types for array and table

Type name of the arguments

```
env assert_eq(type_name(true), "Bool")
env assert_eq(type_name([true, 12]), "Array")
env assert_eq(type_name([true, 12], recursive=true), ["Bool", "Integer"])
env assert_eq(type_name("true"), "String")
```

23.5.1.3. isna

```
env core.isna(val: 'f64')
```

23.5.1.3.1. Arguments

- `val: 'f64' =>`

check if a float is nan

```
env assert(isna(nan + 5))
```

23.5.1.4. isinf

```
env core.isinf(val: 'f64')
```

23.5.1.4.1. Arguments

- `val: 'f64'` =>

check if a float is +/- infinity

```
env assert(isinf(12.0 / 0))
```

23.5.1.5. float

```
env core.float(value: 'Attribute', parse: 'bool' = true)
```

23.5.1.5.1. Arguments

- `value: 'Attribute'` => Argument to convert to float
- `parse: 'bool' = true` => parse string to float

make a float from value

```
env assert_eq(float(5), 5.0)
env assert_eq(float("5.0"), 5.0)
```

23.5.1.6. str

```
env core.str(value: 'Attribute', quote: 'bool' = false)
```

23.5.1.6.1. Arguments

- `value: 'Attribute'` => Argument to convert to float
- `quote: 'bool' = false` => quote it if it's literal string

make a string from value

```
env assert_eq(str(nan + 5), "nan")
env assert_eq(str(2 + 5), "7")
env assert_eq(str(12.34), "12.34")
env assert_eq(str("nan + 5"), "nan + 5")
env assert_eq(str("true", quote=true), "\"true\"")
```

23.5.1.7. int

```
env core.int(
    value: 'Attribute',
    parse: 'bool' = true,
    round: 'bool' = true,
    strfloat: 'bool' = false
)
```

23.5.1.7.1. Arguments

- `value: 'Attribute'` => Argument to convert to int
- `parse: 'bool' = true` => parse string to int
- `round: 'bool' = true` => round float into integer
- `strfloat: 'bool' = false` => parse string first as float before converting to int

make an int from the value

```
env assert_eq(int(5.0), 5)
env assert_eq(int(5.1), 5)
env assert_eq(int("45"), 45)
env assert_eq(int("5.0", strfloat=true), 5)
```

23.5.1.8. array

```
env core.array(*attributes)
```

23.5.1.8.1. Arguments

- `*attributes` => List of attributes

make an array from the arguments

```
env assert_eq(array(5, true), [5, true])
```

23.5.1.9. attrmap

```
env core.attrmap(**attributes)
```

23.5.1.9.1. Arguments

- `**attributes` => name and values of attributes

make an attrmap from the arguments

```
env assert_eq(attrmap(val=5), {val=5})
```

23.5.1.10. json

```
env core.json(value: 'Attribute')
```

23.5.1.10.1. Arguments

- `value: 'Attribute'` => attribute to format

format the attribute as a json string

```
env assert_eq(json(5), "5")
env assert_eq(json([5, true]), "[5, true]")
env assert_eq(json({a=5}), "{\"a\": 5}")
```

23.5.1.11. append

```
env core.append(array: 'Vec < Attribute >', value: 'Attribute')
```

23.5.1.11.1. Arguments

- `array: 'Vec < Attribute >'` => List of attributes
- `value: 'Attribute'` =>

append a value to an array

```
env assert_eq(append([4], 5), [4, 5])
```

23.5.1.12. length

```
env core.length(value: '& Attribute')
```

23.5.1.12.1. Arguments

- `value: '& Attribute'` => Array or a HashMap

length of an array or hashmap

```
env assert_eq(length([4, 5]), 2)
env assert_eq(length({x=4, y=5}), 2)
```

23.5.1.13. year

```
env core.year(value: 'Attribute')
```

23.5.1.13.1. Arguments

- `value: 'Attribute'` => Date or DateTime

year from date/datetime

```
env assert_eq(year(1223-12-12), 1223)
env assert_eq(year(1223-12-12T12:12), 1223)
env assert_eq(year(1223-12-12 12:12:08), 1223)
```

23.5.1.14. month

```
env core.month(value: 'Attribute')
```

23.5.1.14.1. Arguments

- `value: 'Attribute'` => Date or DateTime

month from date/datetime

```
env assert_eq(month(1223-12-14), 12)
env assert_eq(month(1223-12-14T15:19), 12)
```

23.5.1.15. day

```
env core.day(value: 'Attribute')
```

23.5.1.15.1. Arguments

- `value: 'Attribute'` => Date or DateTime

day from date/datetime

```
env assert_eq(day(1223-12-14), 14)
env assert_eq(day(1223-12-14T15:19), 14)
```

23.5.1.16. min_num

```
env core.min_num(vars: 'Vec < Attribute >', start: 'Attribute' = Float(inf))
```

23.5.1.16.1. Arguments

- `vars: 'Vec < Attribute >'` =>
- `start: 'Attribute' = Float(inf)` =>

Minimum of the variables

```
env assert_eq(min_num([1, 2, 3]), 1)
env assert_eq(min_num([1.0, 2, 3]), 1.0)
env assert_eq(min_num([1, 2, 3], start = 0), 0)
```

23.5.1.17. max_num

```
env core.max_num(vars: 'Vec < Attribute >', start: 'Attribute' = Float(-inf))
```

23.5.1.17.1. Arguments

- `vars: 'Vec < Attribute >' =>`
- `start: 'Attribute' = Float(-inf) =>`

Minimum of the variables

```
env assert_eq(max_num([1, 2, 3.0]), 3.0)
env assert_eq(max_num([1.0, 2, 3]), 3)
env assert_eq(max_num([1, inf, 3], 0), inf)
```

23.5.1.18. min

```
env core.min(vars: 'Vec < Attribute >', start: 'Attribute')
```

23.5.1.18.1. Arguments

- `vars: 'Vec < Attribute >' =>`
- `start: 'Attribute' =>`

Minimum of the variables

```
env assert_eq(min([1, 2, 3], 100), 1)
env assert_eq(min([1.0, 2, 3], 100), 1.0)
env assert_eq(min([1, 2, 3], inf), 1)
env assert_eq(min(["b", "a", "d"], "zzz"), "a")
```

23.5.1.19. max

```
env core.max(vars: 'Vec < Attribute >', start: 'Attribute')
```

23.5.1.19.1. Arguments

- `vars: 'Vec < Attribute >' =>`
- `start: 'Attribute' =>`

Maximum of the variables

```
env assert_eq(max([1, 2, 3], -1), 3)
env assert_eq(max([1.0, 2, 3], -1), 3)
env assert_eq(max([1, 2, 3], -inf), 3)
env assert_eq(max(["b", "a", "d"], ""), "d")
```

23.5.1.20. sum

```
env core.sum(vars: 'Vec < Attribute >', start: 'Attribute' = Integer(0))
```

23.5.1.20.1. Arguments

- `vars: 'Vec < Attribute >' =>`
- `start: 'Attribute' = Integer(0) =>`

Sum of the variables

This function is for numeric attributes. You need to give the start attribute so that data type is valid.

```
env assert_eq(sum([2, 3, 4]), 9)
env assert_eq(sum([2, 3, 4], start=0.0), 9.0)
```

23.5.1.21. prod

```
env core.prod(vars: 'Vec < Attribute >', start: 'Attribute' = Integer(1))
```

23.5.1.21.1. Arguments

- `vars: 'Vec < Attribute >' =>`
- `start: 'Attribute' = Integer(1) =>`

Product of the variables

This function is for numerical values/attributes

```
env assert_eq(prod([1, 2, 3]), 6)
env assert_eq(prod([1.0, 2, 3]), 6.0)
```

23.5.1.22. unique_str

```
env core.unique_str(vars: 'Vec < String >')
```

23.5.1.22.1. Arguments

- `vars: 'Vec < String >' =>`

Get a list of unique string values

The order of the strings returned is not guaranteed

```
env.uniq = unique_str(["hi", "me", "hi", "you"]);
env assert_eq(length(uniq), 3)
```

23.5.1.23. count_str

```
env core.count_str(vars: 'Vec < String >')
```

23.5.1.23.1. Arguments

- `vars: 'Vec < String >' =>`

Get a count of unique string values

```
env assert_eq(
count_str(["Hi", "there", "Deliah", "Hi"]),
{Hi = 2, there = 1, Deliah=1}
)
```

23.5.1.24. concat

```
env core.concat(*vars, join: '& str' = "")
```

23.5.1.24.1. Arguments

- `*vars =>`
- `join: '& str' = "" =>`

Concat the strings

```
env assert_eq(concat("Hello", "World", join=" "), "Hello World")
```

23.5.1.25. range

```
env core.range(start: 'i64', end: 'i64')
```

23.5.1.25.1. Arguments

- `start: 'i64' =>`
- `end: 'i64' =>`

Generate integer array, end is not included

```
env assert_eq(range(1, 5), [1, 2, 3, 4])
```

23.5.1.26. assert

```
env core.assert(condition: 'bool', note: 'String' = "Condition False")
```

23.5.1.26.1. Arguments

- `condition: 'bool' =>`
- `note: 'String' = "Condition False" =>`

Assert the condition is true

Use `assert_eq`/`assert_neq` if you are testing equality for better error message.

```
env assert(true)
```

23.5.1.27. assert_eq

```
env core.assert_eq(left: 'Attribute', right: 'Attribute')
```

23.5.1.27.1. Arguments

- `left: 'Attribute' =>`
- `right: 'Attribute' =>`

Assert the two values are equal

This function is for testing the code, as well as for terminating the execution when certain values are not equal

```
env assert_eq(1, 1)
env assert_eq(true, 1 > 0)
env assert_eq("string val", concat("string", " ", "val"))
```

23.5.1.28. assert_neq

```
env core.assert_neq(left: 'Attribute', right: 'Attribute')
```

23.5.1.28.1. Arguments

- `left: 'Attribute' =>`
- `right: 'Attribute' =>`

Assert the two values are not equal

This function is for testing the code, as well as for terminating the execution when certain values are not equal

```
env assert_neq(1, 1.0)
env assert_neq(true, 1 < 0)
env assert_neq("string val", concat("string", "val"))
```

23.5.2. Node Functions

23.5.2.1. inputs_count

```
node core.inputs_count()
```

23.5.2.1.1. Arguments

Count the number of input nodes in the node

```
network load_str("a -> b\n b -> d\n c -> d")
node assert_eq(inputs_count(), length(inputs._))
```

23.5.2.2. inputs_attr

```
node core.inputs_attr(attr: 'String' = "NAME")
```

23.5.2.2.1. Arguments

- `attr: 'String' = "NAME"` => Attribute to get from inputs

Get attributes of the input nodes

This is equivalent to using the `inputs` keyword

```
network load_str("a -> b\n b -> d\n c -> d")
node assert_eq(inputs_attr("NAME"), inputs.NAME)
```

23.5.2.3. has_outlet

```
node core.has_outlet()
```

23.5.2.3.1. Arguments

Node has an outlet or not

This is equivalent to using `output._?`, as _ is a dummy variable that will always be present in all cases, it being absent is because there is no output/outlet of that node.

```
network load_str("a -> b\n b -> d\n c -> d")
node assert_eq(has_outlet(), output._?)
```

23.5.2.4. output_attr

```
node core.output_attr(attr: 'String' = "NAME")
```

23.5.2.4.1. Arguments

- `attr: 'String' = "NAME"` => Attribute to get from inputs

Get attributes of the output node

This is equivalent to using the `output` keyword

```
network load_str("a -> b\n b -> d\n c -> d")
node(output._?) assert_eq(output_attr("NAME"), output.NAME)
```

23.5.3. Network Functions

23.5.3.1. count

```
network core.count(vars: 'Option < Vec < bool > >')
```

23.5.3.1.1. Arguments

- `vars: 'Option < Vec < bool > >' =>`

Count the number of nodes in the network

```
network assert_eq(count(), 0)
network load_str("a -> b")
network assert_eq(count(), 2)
node.sel = INDEX < 1
network assert_eq(count(nodes.sel), 1)
```

23.5.3.2. outlet

```
network core.outlet()
```

23.5.3.2.1. Arguments

Get the name of the outlet node

```
network load_str("a -> b")
network assert_eq(outlet(), "b")
```

23.5.3.3. node_attr

```
network core.node_attr(name: 'String', attribute: 'String' = "_")
```

23.5.3.3.1. Arguments

- `name: 'String'` => name of the node
- `attribute: 'String' = "_"` => attribute to get

Get the attr of the provided node

```
network load_str("a -> b")
network assert_eq(node_attr("a", "NAME"), "a")
```

23.6. Debug

23.6.1. sleep

```
env debug.sleep(time: 'u64' = 1000)
```

23.6.1.1. Arguments

- `time: 'u64' = 1000` =>

sleep for given number of milliseconds

23.6.2. debug

```
env debug.debug(*args, **kwargs)
```

23.6.2.1. Arguments

- `*args` => Function arguments
- `**kwargs` => Function Keyword arguments

Print the args and kwargs on this function

This function will just print out the args and kwargs the function is called with. This is for debugging purposes to see if the args/kwargs are identified properly. And can also be used to see how the nadi system takes the input from the function call.

23.6.3. echo

```
env debug.echo(
    line: 'String',
    error: 'bool' = false,
    newline: 'bool' = true
)
```

23.6.3.1. Arguments

- `line: 'String'` => line to print
- `error: 'bool' = false` => print to stderr instead of stdout
- `newline: 'bool' = true` => print newline at the end

Echo the string to stdout or stderr

This simply echoes anything given to it. This can be used in combination with nadi tasks that create files (image, text, etc). The `echo` function can be called to get the link to those files back to the stdout.

Also useful for nadi preprocessor.

23.6.4. clip

```
env debug.clip(error: 'bool' = false)
```

23.6.4.1. Arguments

- `error: 'bool' = false` => print in stderr instead of in stdout

Echo the `----8<----` line for clipping syntax

This function is a utility function for the generation of nadi book. This prints out the `----8<----` line when called, so that `mdbook` preprocessor for `nadi` knows where to clip the output for displaying it in the book.

This makes it easier to only show the relevant parts of the output in the documentation instead of having the user see output of other unrelated parts which are necessary for generating the results.

23.6.4.2. Example

Given the following tasks file:

```
,ignore
net load_file("...")
net load_attrs("...")
net clip()
net render("{_NAME} {attr1}")
```

The clip function's output will let the preprocessor know that only the parts after that are relevant to the user. Hence, it'll discard outputs before that during documentation generation.

23.7. Files

23.7.1. Env Functions

23.7.1.1. exists

```
env files.exists(path: 'PathBuf', min_lines: 'Option < usize >')
```

23.7.1.1.1. Arguments

- `path: 'PathBuf'` => Path to check
- `min_lines: 'Option < usize >'` => Minimum number of lines the file should have

Checks if the given path exists

23.7.1.2. from_file

```
env files.from_file(path: 'PathBuf', default: 'Option < String >')
```

23.7.1.2.1. Arguments

- `path: 'PathBuf'` => File Path to load the contents from
- `default: 'Option < String >'` => default value

Reads the file contents as string

23.7.1.3. to_file

```
env files.to_file(
    contents: 'String',
    path: 'PathBuf',
    append: 'bool' = false,
    end: 'String' = "\n"
)
```

23.7.1.3.1. Arguments

- `contents: 'String'` => Contents to write
- `path: 'PathBuf'` => Path to write the file
- `append: 'bool' = false` => Append to the file
- `end: 'String' = "\n"` => End the write with this

Writes the string to the file

23.7.2. Node Functions

23.7.2.1. exists

```
node files.exists(path: 'Template', min_lines: 'Option < usize >')
```

23.7.2.1.1. Arguments

- `path: 'Template'` => Path to check
- `min_lines: 'Option < usize >'` => Minimum number of lines the file should have

Checks if the given path exists when rendering the template

23.8. Logic

23.8.1. ifelse

```
env logic.ifelse(
    cond: 'bool',
    iftrue: 'Attribute',
    iffalse: 'Attribute'
)
```

23.8.1.1. Arguments

- `cond: 'bool'` => Attribute that can be cast to bool value
- `iftrue: 'Attribute'` => Output if `cond` is true
- `iffalse: 'Attribute'` => Output if `cond` is false

Simple if else condition

This is similar to using the `if-else` expression, the difference being the condition is relaxed. For example, for `if-else` the condition should be true or false, but for this function, the attribute can be anything that can be cast as true or false. (e.g. 1 => true, 0 => false)

```
env assert_eq(ifelse(true, 1, 2), 1)
env assert_eq(ifelse(false, 1, 2), 2)
env assert_eq(ifelse(100.0, 1, 2), 1)
env assert_eq(ifelse(true, 1, 2), if (true) {1} else {2})
```

23.8.2. gt

```
env logic.gt(a: '& Attribute', b: '& Attribute')
```

23.8.2.1. Arguments

- `a: '& Attribute'` => first attribute
- `b: '& Attribute'` => second attribute

Greater than check

```
env assert_eq(gt(1, 2), 1 > 2)
env assert_eq(gt(1.0, 20), 1.0 > 20)
```

23.8.3. lt

```
env logic.lt(a: '& Attribute', b: '& Attribute')
```

23.8.3.1. Arguments

- `a: '& Attribute'` => first attribute
- `b: '& Attribute'` => second attribute

Less than check

```
env assert_eq(lt(1, 2), 1 < 2)
env assert_eq(lt(1.0, 20), 1.0 < 20)
```

23.8.4. eq

```
env logic.eq(a: '& Attribute', b: '& Attribute')
```

23.8.4.1. Arguments

- `a: '& Attribute'` => first attribute
- `b: '& Attribute'` => second attribute

Equality than check

```
env assert_eq(eq(1, 2), 1 == 2)
env assert_eq(eq(2.0, 2.0), 2.0 == 2.0)
env assert_eq(eq(2.0, 2), 2.0 == 2)
```

23.8.5. and

```
env logic.and(*conds)
```

23.8.5.1. Arguments

- `*conds` => List of attributes that can be cast to bool

Boolean and

Similar to the operator `&` but the values are cast to boolean

```
env assert_eq(and(true, true), true)
env assert_eq(and(true, false), false)
env assert_eq(and(true, false), false & true)
```

23.8.6. or

```
env logic.or(*conds)
```

23.8.6.1. Arguments

- `*conds` => List of attributes that can be cast to bool

boolean or

Similar to the operator `|` but the values are cast to boolean

```
env assert_eq(or(true, false), true)
env assert_eq(or(false, false), false)
env assert_eq(or(true, false), false | true)
```

23.8.7. not

```
env logic.not(cond: 'bool')
```

23.8.7.1. Arguments

- `cond: 'bool'` => attribute that can be cast to bool

boolean not

Similar to the operator `!` but the values are cast to boolean

```
env assert_eq(not(true), false)
env assert_eq(not(false), true)
env assert_eq(not(true), !true)
env assert_eq(not(false), !false)
```

23.8.8. all

```
env logic.all(vars: '& [bool]')
```

23.8.8.1. Arguments

- `vars: '& [bool]' =>`

check if all of the bool are true

```
env assert_eq(all([true]), true)
env assert_eq(all([false, true]), false)
env assert_eq(all([true, true]), true)
env assert_eq(all([false]), false)
```

23.8.9. any

```
env logic.any(vars: '& [bool]')
```

23.8.9.1. Arguments

- `vars: '& [bool]' =>`

check if any of the bool are true

```
env assert_eq(any([true]), true)
env assert_eq(any([false, true]), true)
env assert_eq(any([false, false]), false)
env assert_eq(any([false]), false)
```

23.9. Regex

23.9.1. str_filter

```
env regex.str_filter(attrs: 'Vec < String >', pattern: 'Regex')
```

23.9.1.1. Arguments

- `attrs: 'Vec < String >'` => attribute to check for pattern
- `pattern: 'Regex'` => Regex pattern to match

Filter from the string list with only the values matching pattern

```
env assert_eq(str_filter(["abc", "and", "xyz"], "^a"), ["abc", "and"])
```

23.9.2. str_match

```
env regex.str_match(attr: '& str', pattern: 'Regex')
```

23.9.2.1. Arguments

- `attr: '& str'` => attribute to check for pattern
- `pattern: 'Regex'` => Regex pattern to match

Check if the given pattern matches the value or not

You can also use match operator for this

```
env assert_eq(str_match("abc", "^a"), true)
env assert_eq(str_match("abc", "^a"), "abc" match "^a")
```

23.9.3. str_replace

```
env regex.str_replace(
    attr: '& str',
    pattern: 'Regex',
    rep: '& str'
)
```

23.9.3.1. Arguments

- `attr: '& str'` => original string
- `pattern: 'Regex'` => Regex pattern to match
- `rep: '& str'` => replacement string

Replace the occurances of the given match

```
env assert_eq(str_replace("abc", "^a", 2), "2bc")
env assert_eq(str_replace("abc", "[abc]", 2), "222")
```

23.9.4. str_find

```
env regex.str_find(attr: '& str', pattern: 'Regex')
```

23.9.4.1. Arguments

- `attr: '& str'` => attribute to check for pattern
- `pattern: 'Regex'` => Regex pattern to match

Find the given pattern in the value

```
env assert_eq(str_find("abc", "^[ab]"), "a")
```

23.9.5. str_find_all

```
env regex.str_find_all(attr: '& str', pattern: 'Regex')
```

23.9.5.1. Arguments

- `attr: '& str'` => attribute to check for pattern
- `pattern: 'Regex'` => Regex pattern to match

Find all the matches of the given pattern in the value

```
env assert_eq(str_find_all("abc", "[ab]"), ["a", "b"])
```

23.9.6. str_count

```
env regex.str_count(attr: '& str', pattern: 'Regex')
```

23.9.6.1. Arguments

- `attr: '& str'` => attribute to check for pattern
- `pattern: 'Regex'` => Regex pattern to match

Count the number of matches of given pattern in the string

```
env assert_eq(str_count("abc", "[ab]"), 2)
```

23.9.7. str_split

```
env regex.str_split(
    attr: '& str',
    pattern: 'Regex',
    limit: 'Option < usize >'
)
```

23.9.7.1. Arguments

- `attr: '& str'` => String to split
- `pattern: 'Regex'` => Regex pattern to split with
- `limit: 'Option < usize >'` => Limit the split to maximum number

Split the string with the given pattern

```
env assert_eq(str_split("abc", "^[ab]"), ["", "bc"])
```

23.10. Render

23.10.1. Env Functions

23.10.1.1. render

```
env render.render(
    template: '& Template',
    safe: 'bool' = false,
    **keyval
)
```

23.10.1.1.1. Arguments

- `template: '& Template'` => String template to render
- `safe: 'bool' = false` => if render fails keep it as it is instead of exiting
- `**keyval` =>

Render the template based on the node attributes

For more details on the template system. Refer to the String Template section of the NADI book.

```
env assert_eq(render("abc {_x}", x="ab"), "abc ab")
env assert_eq(render("abc {x}", x=23), "abc 23")
```

If safe parameter is true, then it doesn't error out even if the variable is not present, and will just return the original template. By default it errors out if there are any variables in the template without a value.

```
env assert_eq(render("abc {x}", safe=true), "abc {x}")
```

23.10.2. Node Functions

23.10.2.1. render

```
node render.render(template: '& Template', safe: 'bool' = false)
```

23.10.2.1.1. Arguments

- `template: '& Template'` => String template to render
- `safe: 'bool' = false` => if render fails keep it as it is instead of exiting

Render the template based on the node attributes

For more details on the template system. Refer to the String Template section of the NADI book.

```
network load_str("a -> b")
node.x = 13
node assert_eq(render("abc {x}"), "abc 13")
```

23.10.3. Network Functions

23.10.3.1. render

```
network render.render(template: '& Template', safe: 'bool' = false)
```

23.10.3.1.1. Arguments

- `template: '& Template'` => Path to the template file
- `safe: 'bool' = false` => if render fails keep it as it is instead of exiting

Render from network attributes

```
network.x = 13
network assert_eq(render("abc {x}"), "abc 13")
```

23.10.3.2. render_nodes

```
network render.render_nodes(
    template: '& Template',
    safe: 'bool' = false,
    join: '& str' = "\n"
)
```

23.10.3.2.1. Arguments

- `template: '& Template'` => Path to the template file
- `safe: 'bool' = false` => if render fails keep it as it is instead of exiting
- `join: '& str' = "\n"` => String to join the render results

Render each node of the network and combine to same variable

```
network load_str("a -> b")
node.x = INDEX + 1
network assert_eq(render_nodes("abc {x}"), "abc 1\nabc 2")
```

23.10.3.3. render_template

```
network render.render_template(template: 'PathBuf')
```

23.10.3.3.1. Arguments

- `template: 'PathBuf'` => Path to the template file

Render a File template for the nodes in the whole network

Write the file with templates for input variables in the same way you write string templates. It's useful for markdown files, as the curly braces syntax won't be used for anything else that way. Do be careful about that. And the program will replace those templates with their values when you run it with inputs.

It'll repeat the same template for each node and render them. If you want only a portion of the file repeated for nodes inclose them with lines with `---8<---` on both start and the end. The lines containing the clip syntax will be ignored, ideally you can put them in comments.

You can also use `---include:<filename>[::line_range]` syntax to include a file, the line_range syntax, if present, should be in the form of `start[:increment]:end`, you can exclude start or end to denote the line 1 or last line (e.g. `:5` is 1:5, and `3:` is from line 3 to the end)

23.10.3.3.2. Arguments

- `template`: Path to the template file
- `outfile` [Optional]: Path to save the template file, if none it'll be printed in stdout

23.11. Series

23.11.1. sr_count

```
node series.sr_count()
```

23.11.1.1. Arguments

Number of series in the node

23.11.2. sr_list

```
node series.sr_list()
```

23.11.2.1. Arguments

List all series in the node

23.11.3. sr_dtype

```
node series.sr_dtype(name: '& str', safe: 'bool' = false)
```

23.11.3.1. Arguments

- `name: '& str'` => Name of the series
- `safe: 'bool' = false` => Do not error if series does't exist

Type name of the series

23.11.4. sr_len

```
node series.sr_len(name: '& str', safe: 'bool' = false)
```

23.11.4.1. Arguments

- `name: '& str'` => Name of the series
- `safe: 'bool' = false` => Do not error if series does't exist

Length of the series

23.11.5. sr_mean

```
node series.sr_mean(name: '& str')
```

23.11.5.1. Arguments

- `name: '& str'` => Name of the series

Type name of the series

23.11.6. sr_sum

```
node series.sr_sum(name: '& str')
```

23.11.6.1. Arguments

- `name: '& str'` => Name of the series

Sum of the series

23.11.7. set_series

```
node series.set_series(
    name: '& str',
    value: 'Attribute',
    dtype: '& str'
)
```

23.11.7.1. Arguments

- `name: '& str'` => Name of the series to save as
- `value: 'Attribute'` => Argument to convert to series
- `dtype: '& str'` => type

set the following series to the node

23.11.8. sr_to_array

```
node series.sr_to_array(name: '& str', safe: 'bool' = false)
```

23.11.8.1. Arguments

- `name: '& str'` => Name of the series
- `safe: 'bool' = false` => Do not error if series does't exist

Make an array from the series

23.12. Table

23.12.1. save_csv

```
network table.save_csv(
    path: '& Path',
    fields: '& [String]',
    filter: 'Option < Vec < bool > >'
)
```

23.12.1.1. Arguments

- `path: '& Path'` =>
- `fields: '& [String]'` =>
- `filter: 'Option < Vec < bool > >'` =>

Save CSV

23.12.2. table_to_markdown

```
network table.table_to_markdown(
    table: 'Option < PathBuf >',
    template: 'Option < String >',
    outfile: 'Option < PathBuf >',
    connections: 'Option < String >'
)
```

23.12.2.1. Arguments

- `table: 'Option < PathBuf >'` => Path to the table file
- `template: 'Option < String >'` => String template for table
- `outfile: 'Option < PathBuf >'` => Path to the output file
- `connections: 'Option < String >'` => Show connections column or not

Render the Table as a rendered markdown

23.12.2.2. Error

The function will error out if,

- error reading the table file,
- error parsing table template,
- neither one of table file or table template is provided,
- error while rendering markdown

(caused by error on rendering cell values from templates)
- error while writing to the output file

23.13. Timeseries

23.13.1. Node Functions

23.13.1.1. ts_count

```
node timeseries.ts_count()
```

23.13.1.1.1. Arguments

Number of timeseries in the node

23.13.1.2. ts_list

```
node timeseries.ts_list()
```

23.13.1.2.1. Arguments

List all timeseries in the node

23.13.1.3. ts_dtype

```
node timeseries.ts_dtype(name: '& str', safe: 'bool' = false)
```

23.13.1.3.1. Arguments

- `name: '& str'` => Name of the timeseries
- `safe: 'bool' = false` => Do not error if timeseries does't exist

Type name of the timeseries

23.13.1.4. ts_len

```
node timeseries.ts_len(name: '& str', safe: 'bool' = false)
```

23.13.1.4.1. Arguments

- `name: '& str'` => Name of the timeseries
- `safe: 'bool' = false` => Do not error if timeseries does't exist

Length of the timeseries

23.13.1.5. ts_print

```
node timeseries.ts_print(
    name: '& String',
    header: 'bool' = true,
    head: 'Option < i64 >'
)
```

23.13.1.5.1. Arguments

- `name: '& String'` => name of the timeseries
- `header: 'bool' = true` => show header
- `head: 'Option < i64 >'` => number of head rows to show (all by default)

Print the given timeseries values in csv format

23.13.1.5.2. TODO

- save to file instead of showing with `outfile: Option<PathBuf>`

23.13.2. Network Functions

23.13.2.1. ts_print_csv

```
network timeseries.ts_print_csv(
    name: 'String',
    head: 'Option < usize >',
    nodes: 'Option < HashSet < String > >'
)
```

23.13.2.1.1. Arguments

- `name: 'String'` => Name of the timeseries to save
- `head: 'Option < usize >'` => number of head rows to show (all by default)
- `nodes: 'Option < HashSet < String > >'` => Include only these nodes (all by default)

Save timeseries from all nodes into a single csv file

TODO: error/not on unqual length TODO: error/not on no timeseries, etc... TODO: output to `file: PathBuf`

23.13.2.2. series_csv

```
network timeseries.series_csv(
    filter: 'Vec < bool >',
    outfile: 'PathBuf',
    attrs: 'Vec < String >',
    series: 'Vec < String >'
)
```

23.13.2.2.1. Arguments

- `filter: 'Vec < bool >'` =>
- `outfile: 'PathBuf'` => Path to the output csv
- `attrs: 'Vec < String >'` => list of attributes to write
- `series: 'Vec < String >'` => list of series to write

Write the given nodes to csv with given attributes and series

23.14. Visuals

23.14.1. set_nodesize_attrs

```
network visuals.set_nodesize_attrs(
    attrs: '& [f64]',
    minsize: 'f64' = 4.0,
    maxsize: 'f64' = 12.0
)
```

23.14.1.1. Arguments

- `attrs: '& [f64]'` => Attribute values to use for size scaling
- `minsize: 'f64' = 4.0` => minimum size of the node
- `maxsize: 'f64' = 12.0` => maximum size of the node

Set the node size of the nodes based on the attribute value

23.14.2. svg_save

```
network visuals.svg_save(
    outfile: '& Path',
    label: 'Template' = Template { original: "{_NAME}", parts: [Var("_NAME", "")] },
    x_spacing: 'u64' = 25,
    y_spacing: 'u64' = 25,
    offset: 'u64' = 10,
    twidth: 'f64' = 9.0,
    width: 'u64' = 500,
    height: 'u64' = 240,
    bgcolor: 'Option < String >',
    page_width: 'Option < u64 >',
    page_height: 'Option < u64 >'
)
```

23.14.2.1. Arguments

- `outfile: '& Path'` =>
- `label: 'Template' = Template { original: "{_NAME}", parts: [Var("_NAME", "")] }` =>
- `x_spacing: 'u64' = 25` =>
- `y_spacing: 'u64' = 25` =>
- `offset: 'u64' = 10` =>

- `twidth: 'f64' = 9.0` => in average how many units each text character takes

For auto calculating width of the page since we don't have Cairo

- `width: 'u64' = 500` =>
- `height: 'u64' = 240` =>
- `bgcolor: 'Option < String >'` =>
- `page_width: 'Option < u64 >'` =>
- `page_height: 'Option < u64 >'` =>

Exports the network as a svg

24. External Plugins

This section showcases the functions from external plugins developed along side the NADI project due to various reasons.

The plugins listed here can be installed with following steps:

- clone the repository of external plugins,
- compile it locally with cargo,
- move all generated dynamic libraries to the nadi plugin directory.

24.1. Dams

24.1.1. count_node_if

```
node dams.count_node_if(count_attr: '& str', cond: 'bool')
```

24.1.1.1. Arguments

- `count_attr: '& str'` =>
- `cond: 'bool'` =>

Count the number of nodes upstream at each point that satisfies a certain condition

24.1.2. min_year

```
node dams.min_year(yearattr: '& str', write_var: '& str' = "MIN_YEAR")
```

24.1.2.1. Arguments

- `yearattr: '& str'` =>
- `write_var: '& str' = "MIN_YEAR"` =>

Propagate the minimum year downstream

24.2. DataFill

24.2.1. Node Functions

24.2.1.1. load_csv_fill

```
node datafill.load_csv_fill(
    name: 'String',
    file: 'Template',
    timefmt: 'String',
    columns: '(String, String)',
    method: 'DataFillMethod' = Linear,
    dtype: 'String' = "Floats"
)
```

24.2.1.1.1. Arguments

- `name: 'String'` => Name of the timeseries
- `file: 'Template'` => Template of the CSV file for the nodes
- `timefmt: 'String'` => date time format, if you only have date, but have time on format string, it will panic
- `columns: '(String, String)'` => Names of date column and value column
- `method: 'DataFillMethod' = Linear` => Method to use for data filling: forward/backward/linear
- `dtype: 'String' = "Floats"` => DataType to load into timeseries

24.2.1.2. datafill_experiment

```
node datafill.datafill_experiment(
    name: 'String',
    file: 'Template',
    ratio_var: 'String',
    columns: 'Option < (String, String) >',
    experiments: 'usize' = 10,
    samples: 'usize' = 100
)
```

24.2.1.2.1. Arguments

- `name: 'String'` => Prefix for name of the series to save metrics on
- `file: 'Template'` => Template of the CSV file for the nodes
- `ratio_var: 'String'` => Variable to use for inputratio/outputratio methods
- `columns: 'Option < (String, String) >'` => Names of date column and value column
- `experiments: 'usize' = 10` => Number of experiements to run
- `samples: 'usize' = 100` => Number of samples on each experiment

24.2.2. Network Functions

24.2.2.1. save_experiments_csv

```
network datafill.save_experiments_csv(
    outfile: 'PathBuf',
    attrs: 'Vec < String >',
    prefix: 'String',
    errors: 'Vec < String >',
    filter: 'Option < Vec < bool > >'
)
```

24.2.2.1.1. Arguments

- `outfile: 'PathBuf'` => Path to the output csv
- `attrs: 'Vec < String >'` => list of attributes to write
- `prefix: 'String'` => Prefix
- `errors: 'Vec < String >'` => list of errors to write
- `filter: 'Option < Vec < bool > >'` =>

Write the given nodes to csv with given attributes and experiment results

24.3. DSS

24.3.1. list_catalog

```
env dss.list_catalog(dssfile: 'String', paths: 'String' = "/*/*/*/*/*/*/*/")
```

24.3.1.1. Arguments

- `dssfile: 'String'` =>
- `paths: 'String' = "/*/*/*/*/*/*/*/"` =>

List the catalog of the dss file

env list_catalog("/home/gaurav/work/nadi-project/codes/nadi-dss/dsslib/routed-main-stem.dss")

24.4. Errors

24.4.1. Node Functions

24.4.1.1. calc_ts_error

```
node errors.calc_ts_error(
    ts1: '& str',
    ts2: '& str',
    error: '& str' = "rmse"
)
```

24.4.1.1.1. Arguments

- `ts1: '& str'` => Timeseries value to use as actual value
- `ts2: '& str'` => Timeseries value to be used to calculate the error
- `error: '& str' = "rmse"` => Error type, one of rmse/nrmse/abserr/nse

Calculate Error from two timeseries values in the node

It calculates the error between two timeseries values from the node

24.4.1.2. calc_ts_errors

```
node errors.calc_ts_errors(
    ts1: '& String',
    ts2: '& String',
    errors: '& [String]'
)
```

24.4.1.2.1. Arguments

- `ts1: '& String'` => Timeseries value to use as actual value
- `ts2: '& String'` => Timeseries value to be used to calculate the error
- `errors: '& [String]'` => Error types to calculate, one of rmse/nrmse/abserr/nse

Calculate Error from two timeseries values in the node

It calculates the error between two timeseries values from the node.

24.4.2. Network Functions

24.4.2.1. calc_attr_error

```
network errors.calc_attr_error(
    attr1: 'String',
    attr2: 'String',
    error: 'String' = "rmse"
)
```

24.4.2.1.1. Arguments

- `attr1: 'String'` => Attribute value to use as actual value
- `attr2: 'String'` => Attribute value to be used to calculate the error
- `error: 'String' = "rmse"` => Error type, one of rmse/nrmse/abserr/nse

Calculate Error from two attribute values in the network

It calculates the error using two attribute values from all the nodes.

24.5. Fancy Print

24.5.1. fancy_print

```
network fancy_print.fancy_print()
```

24.5.1.1. Arguments

Fancy print a network

24.6. Gnuplot

24.6.1. plot_timeseries

```
network gnuplot.plot_timeseries(
    csvfile: 'Template',
    datecol: '& str',
    datacol: '& str',
    outfile: '& Path',
    timefmt: '& str' = "%Y-%m-%d",
    config: '& GnuplotConfig' = GnuplotConfig { outfile: None, terminal: None, csv:
false, preamble: "" },
    skip_missing: 'bool' = false
)
```

24.6.1.1. Arguments

- `csvfile: 'Template' =>`
- `datecol: '& str' =>`
- `datacol: '& str' =>`
- `outfile: '& Path' =>`
- `timefmt: '& str' = "%Y-%m-%d" =>`
- `config: '& GnuplotConfig' = GnuplotConfig { outfile: None, terminal: None, csv: false, preamble: "" } =>`
- `skip_missing: 'bool' = false =>`

Generate a gnuplot file that plots the timeseries data in the network

24.7. Graphics

This plugin uses cairo to draw graphics, so far it has only been tested on Linux, but this should also work on Mac. Compiling it in windows might need additional steps that are not documented here.

24.7.1. Node Functions

24.7.1.1. attr_fraction_svg

```
node graphics.attr_fraction_svg(
    attr: '& str',
    outfile: '& Template',
    color: '& AttrColor',
    height: 'f64' = 80.0,
    width: 'f64' = 80.0,
    margin: 'f64' = 10.0
)
```

24.7.1.1.1. Arguments

- `attr: '& str'` =>
- `outfile: '& Template'` =>
- `color: '& AttrColor'` =>
- `height: 'f64' = 80.0` =>
- `width: 'f64' = 80.0` =>
- `margin: 'f64' = 10.0` =>

Create a SVG file with the given network structure

24.7.2. Network Functions

24.7.2.1. csv_load_ts

```
network graphics.csv_load_ts(
    file: 'PathBuf',
    name: 'String',
    date_col: 'String' = "date",
    timefmt: 'String' = "%Y-%m-%d",
```

```
    data_type: 'String' = "Floats"
)
```

24.7.2.1.1. Arguments

- `file: 'PathBuf' =>`
- `name: 'String' =>`
- `date_col: 'String' = "date" =>`
- `timefmt: 'String' = "%Y-%m-%d" =>`
- `data_type: 'String' = "Floats" =>`

Count the number of na values in CSV file for each nodes in a network

24.7.2.1.2. Arguments

- `file`: Input CSV file path to read (should have column with

node names for all nodes)
- `name`: Name of the timeseries
- `date_col`: Date Column name
- `timefmt`: date time format, if you only have date, but have time on format string, it will panic
- `data_type`: Type of the data to cast into

24.7.2.2. csv_count_na

```
network graphics.csv_count_na(
    file: 'PathBuf',
    outattr: 'Option < String >',
    sort: 'bool' = false,
    skip_zero: 'bool' = false,
    head: 'Option < i64 >'
)
```

24.7.2.2.1. Arguments

- `file: 'PathBuf' =>`
- `outattr: 'Option < String >' =>`
- `sort: 'bool' = false =>`
- `skip_zero: 'bool' = false =>`
- `head: 'Option < i64 >' =>`

Count the number of na values in CSV file for each nodes in a network

24.7.2.2.2. Arguments

- `file`: Input CSV file path to read (should have column with node names for all nodes)
- `outattr`: Output attribute to save the count of NA to. If empty print to stdout
- `sort`: show the nodes with larger gaps on top, only applicable while printing
- `head`: at max show only this number of nodes
- `skip_zero`: skip nodes with zero missing numbers

24.7.2.3. csv_data_blocks_svg

```
network graphics.csv_data_blocks_svg(
    csvfile: 'PathBuf',
    outfile: 'PathBuf',
    label: 'Template',
    date_col: 'String' = "date",
    config: 'NetworkPlotConfig' = NetworkPlotConfig { width: 250.0, height: 300.0,
delta_x: 20.0, delta_y: 20.0, offset: 30.0, radius: 3.0, fontsize: 16.0, fontface:
FontFace { inner: Shared { inner: 0x64a7356cd4c0 } } },
    blocks_width: 'f64' = 500.0,
    fit: 'bool' = false
)
```

24.7.2.3.1. Arguments

- `csvfile: 'PathBuf' =>`
- `outfile: 'PathBuf' =>`
- `label: 'Template' =>`
- `date_col: 'String' = "date" =>`
- `config: 'NetworkPlotConfig' = NetworkPlotConfig { width: 250.0, height: 300.0, delta_x: 20.0, delta_y: 20.0, offset: 30.0, radius: 3.0, fontsize: 16.0, fontface: FontFace { inner: Shared { inner: 0x64a7356cd4c0 } } } =>`
- `blocks_width: 'f64' = 500.0 =>`
- `fit: 'bool' = false =>`

Draw the data blocks with arrows in timeline

24.7.2.4. export_svg

```
network graphics.export_svg(
    outfile: 'PathBuf',
    config: 'NetworkPlotConfig' = NetworkPlotConfig { width: 250.0, height: 300.0,
delta_x: 20.0, delta_y: 20.0, offset: 30.0, radius: 3.0, fontsize: 16.0, fontface:
FontFace { inner: Shared { inner: 0x64a7356cd4c0 } } },
    fit: 'bool' = false,
    label: 'Option < Template >',
    highlight: '& [usize]' = []
)
```

24.7.2.4.1. Arguments

- `outfile: 'PathBuf' =>`
- `config: 'NetworkPlotConfig' = NetworkPlotConfig { width: 250.0, height: 300.0, delta_x: 20.0, delta_y: 20.0, offset: 30.0, radius: 3.0, fontsize: 16.0, fontface: FontFace { inner: Shared { inner: 0x64a7356cd4c0 } } } =>`
- `fit: 'bool' = false =>`
- `label: 'Option < Template >' =>`
- `highlight: '& [usize]' = [] =>`

Create a SVG file with the given network structure

24.7.2.5. table_to_svg

```
network graphics.table_to_svg(
    outfile: 'PathBuf',
    table: 'Option < PathBuf >',
    template: 'Option < String >',
    config: 'NetworkPlotConfig' = NetworkPlotConfig { width: 250.0, height: 300.0,
delta_x: 20.0, delta_y: 20.0, offset: 30.0, radius: 3.0, fontsize: 16.0, fontface:
FontFace { inner: Shared { inner: 0x64a7356cd4c0 } } },
    fit: 'bool' = false,
    highlight: '& [String]' = []
)
```

24.7.2.5.1. Arguments

- `outfile: 'PathBuf' =>`
- `table: 'Option < PathBuf >' =>`
- `template: 'Option < String >' =>`

- `config: 'NetworkPlotConfig' = NetworkPlotConfig { width: 250.0, height: 300.0, delta_x: 20.0, delta_y: 20.0, offset: 30.0, radius: 3.0, fontsize: 16.0, fontface: FontFace { inner: Shared { inner: 0x64a7356cd4c0 } } } =>`
- `fit: 'bool' = false =>`
- `highlight: '& [String]' = [] =>`

Create a SVG file with the given network structure

24.8. Graphviz

24.8.1. save_graphviz

```
network graphviz.save_graphviz(
    outfile: '& Path',
    name: '& str' = "network",
    global_attrs: '& str' = "",
    node_attr: 'Option < & Template >',
    edge_attr: 'Option < & Template >'
)
```

24.8.1.1. Arguments

- `outfile: '& Path' =>`
- `name: '& str' = "network" =>`
- `global_attrs: '& str' = "" =>`
- `node_attr: 'Option < & Template >' =>`
- `edge_attr: 'Option < & Template >' =>`

Save the network as a graphviz file

24.8.1.2. Arguments:

- `outfile` - Path to the output file
- `name` - Name of the graph

24.9. HTML

24.9.1. export_map

```
network html.export_map(
    outfile: '& Path',
    template: 'Template',
    pagetitle: '& str' = "NADI Network",
    nodetitle: 'Template' = Template { original: "{_NAME}", parts: [Var("_NAME",
"")] },
    connections: 'bool' = true
)
```

24.9.1.1. Arguments

- `outfile: '& Path' =>`
- `template: 'Template' =>`
- `pagetitle: '& str' = "NADI Network" =>`
- `nodetitle: 'Template' = Template { original: "{_NAME}", parts: [Var("_NAME", "")] } =>`
- `connections: 'bool' = true =>`

Exports the network as a HTML map

24.10. GIS

This plugin uses gdal to read/write GIS files, it can be compiled easily in Linux and Mac by installing gdal as a prerequisite, but on windows that step might be complicated. Please refer to the documentation of gdal to know how to install it in windows. Or use the provided dlls from the plugin repo.

24.10.1. Network Functions

24.10.1.1. gis_load_network

```
network gis.gis_load_network(
    file: 'PathBuf',
    source: 'String',
    destination: 'String',
    layer: 'Option < String >',
```

```
    ignore_null: 'bool' = false
)
```

24.10.1.1.1. Arguments

- `file: 'PathBuf'` => GIS file to load (can be any format GDAL can understand)
- `source: 'String'` => Field in the GIS file corresponding to the input node name
- `destination: 'String'` => layer of the GIS file corresponding to the output node name
- `layer: 'Option < String >'` => layer of the GIS file, first one picked by default
- `ignore_null: 'bool' = false` => Ignore feature if it has fields with null value

Load network from a GIS file

Loads the network from a gis file containing the edges in fields

24.10.1.2. gis_load_attrs

```
network gis.gis_load_attrs(
    file: 'PathBuf',
    node: 'String',
    layer: 'Option < String >',
    geometry: 'String' = "GEOM",
    ignore: 'String' = "",
    sanitize: 'bool' = true,
    err_no_node: 'bool' = false
)
```

24.10.1.2.1. Arguments

- `file: 'PathBuf'` => GIS file to load (can be any format GDAL can understand)
- `node: 'String'` => Field in the GIS file corresponding to node name
- `layer: 'Option < String >'` => layer of the GIS file, first one picked by default
- `geometry: 'String' = "GEOM"` => Attribute to save the GIS geometry in
- `ignore: 'String' = ""` => Field names separated by comma, to ignore
- `sanitize: 'bool' = true` => sanitize the name of the fields
- `err_no_node: 'bool' = false` => Error if all nodes are not found in the GIS file

Load node attributes from a GIS file

The function reads a GIS file in any format (CSV, GPKG, SHP, JSON, etc) and loads their fields as attributes to the nodes.

24.10.1.3. gis_save_connections

```
network gis.gis_save_connections(
    file: 'PathBuf',
    geometry: 'String',
    driver: 'Option < String >',
    layer: 'String' = "network",
    filter: 'Option < Vec < bool > >'
)
```

24.10.1.3.1. Arguments

- `file: 'PathBuf' =>`
- `geometry: 'String' =>`
- `driver: 'Option < String >' =>`
- `layer: 'String' = "network" =>`
- `filter: 'Option < Vec < bool > >' =>`

Save GIS file of the connections

24.10.1.4. gis_save_nodes

```
network gis.gis_save_nodes(
    file: 'PathBuf',
    geometry: 'String',
    attrs: 'HashMap < String, String >' = {},
    driver: 'Option < String >',
    layer: 'String' = "nodes",
    filter: 'Option < Vec < bool > >'
)
```

24.10.1.4.1. Arguments

- `file: 'PathBuf' =>`
- `geometry: 'String' =>`
- `attrs: 'HashMap < String, String >' = {} =>`
- `driver: 'Option < String >' =>`
- `layer: 'String' = "nodes" =>`
- `filter: 'Option < Vec < bool > >' =>`

Save GIS file of the nodes

24.11. Print Node

24.11.1. Node Functions

24.11.1.1. print_node

```
node print_node.print_node()
```

24.11.1.1.1. Arguments

Print the node with its inputs and outputs

24.11.2. Network Functions

24.11.2.1. print_attr_csv

```
network print_node.print_attr_csv(*args)
```

24.11.2.1.1. Arguments

- `*args` =>

Print the given attributes in csv format with first column with node name

24.12. Streamflow

24.12.1. check_negative

```
node streamflow.check_negative(ts_name: '& str')
```

24.12.1.1. Arguments

- `ts_name: '& str'` => Name of the timeseries with streamflow data

Check the given streamflow timeseries for negative values

24.13. NADI PDF

24.13.1. Env Functions

24.13.1.1. typst_compile

```
env nadi_pdf.typst_compile(content: 'String', outfile: 'PathBuf')
```

24.13.1.1.1. Arguments

- `content: 'String' =>`
- `outfile: 'PathBuf' =>`

convert the typst content into pdf/svg/png

24.13.2. Network Functions

24.13.2.1. typst_table

```
network nadi_pdf.typst_table(table: 'Table')
```

24.13.2.1.1. Arguments

- `table: 'Table' =>`

Generate Typst code for given Table

Developer Reference

25. Software Architect

NADI System consists of multiple components. The top-most division is in the GIS component and the DSL component.

25.1. NADI GIS

This is the GIS component that helps with network detection, dealing with GIS files, and other utilities.

NADI GIS repository contains 3 important components:

25.1.1. `cli_tool`

This contains the CLI for the NADI GIS tool (this compiles to generate `nadi-gis` binary). The `main.rs` file contains the code for the CLI, with files inside it providing data types, or code for the subcommands.

Currently it has the following subcommands:

Command	File	Description
`nid`	`nid.rs`	Download the National Inventory of Dams dataset
`usgs`	`usgs.rs`	Download data from USGS National Hydrography Dataset (NHD+)
`layers`	`layers.rs`	Show list of layers in a GIS file
`check`	`check.rs`	Check the stream network to see outlet, branches, etc
`order`	`order.rs`	Order the streams, adds order attribute to each segment
`network`	`network.rs`	Find the network information from streams file between points

25.1.2. `qgis`

This directory contains the NADI QGIS plugin. The plugin is written in Python. This is an experiemental plugin that calls the `nadi-gis` command line utility with appropriate flags.

25.1.3. `nadi_plugin`

This is the plugin for NADI DSL that has GIS functions. This compiles into a shared library that can be loaded with NADI System, refer to the nadi plugins documentation on how to load plugins.

The repo contains `lib.rs` that generates the plugin codes and function available for loading/saving GIS files from the NADI DSL.

25.2. NADI DSL

The NADI DSL also called the Task System consists of components that are related to the programming language that helps with network analysis.

The figure below shows the internal data structure and components that support the NADI DSL execution.

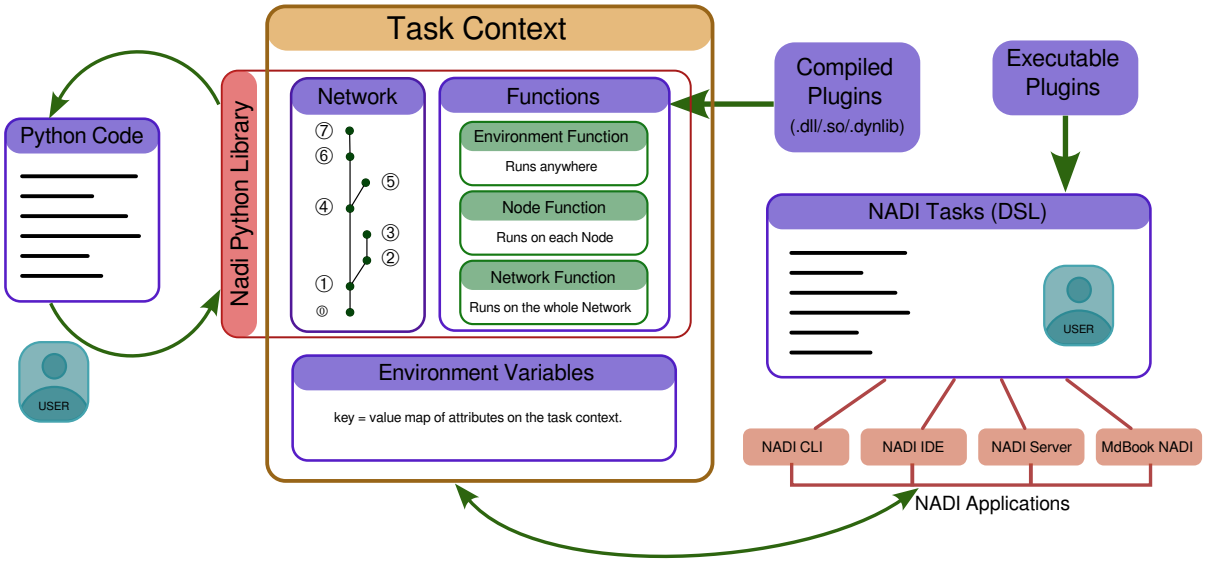


Figure 38: Tasks Architecture

The task context is the main runtime environment for the DSL. The Context contains a Network (starts as empty network), functions loaded from the plugins, and the environmental variables. As the DSL is executed Task by Task, the context is modified with mutable functions or assignment operator. For example running a `network load_file(...)` will load a network from file and save that in the context. When a user runs a `node` task, the expression/function is run on each node of the network in the current context.

If a user is using NADI from Python library instead of the DSL, then they have access to the Functions, and the data types to generate the Network from files/strings/edges. But they don't have access to environment variables (they can use python variables), and other syntax advantages of DSL. They also have to run the node functions in loop yourself. Because of this `nadi-py` should provide data types from NADI, as well as the functions from the plugins as callable objects in python along side their documentation. We use `maturin` to generate the python bindings from the rust code.

NADI DSL can also load `nadi_gis` plugin that provides the functions to read/write GIS files.

The NADI DSL repository contains the following components:

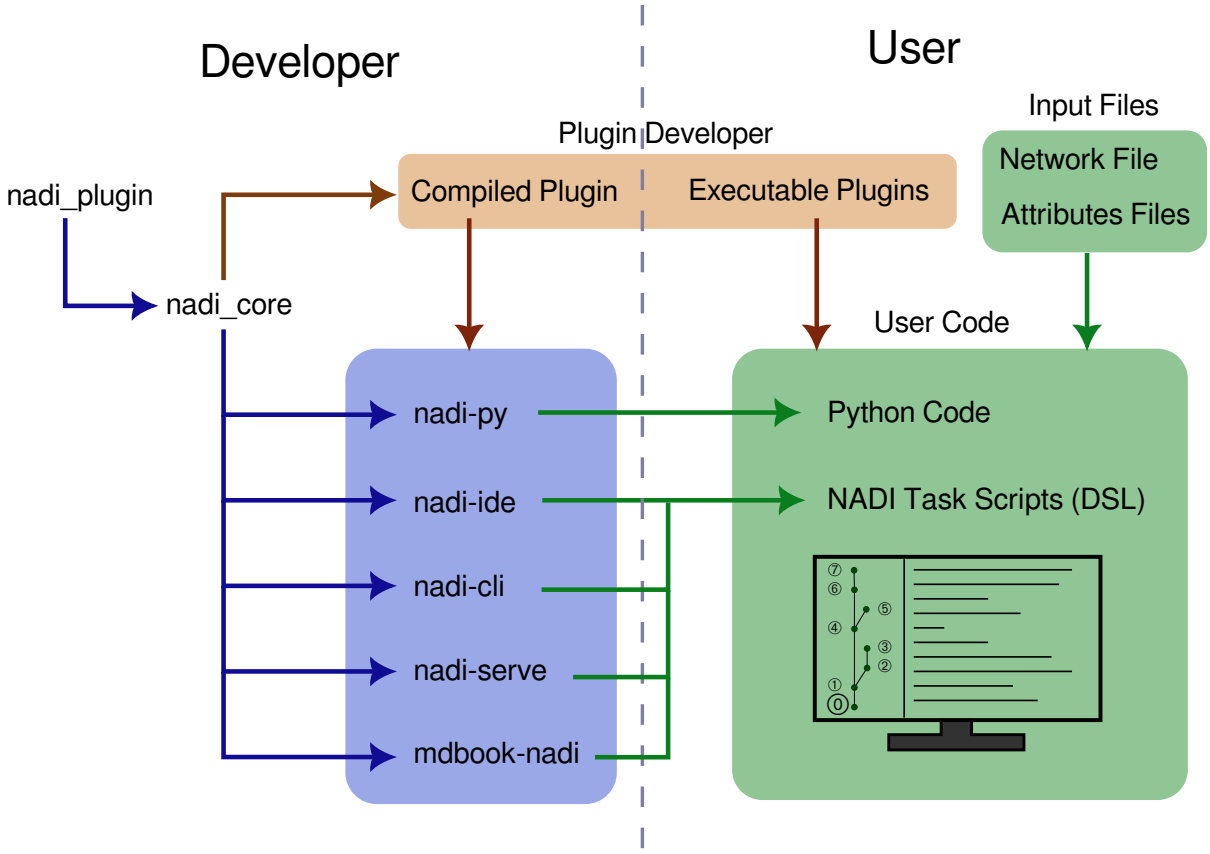


Figure 39: Architecture

25.2.1. `nadi_plugin`

This is crate for proc-macros that `nadi_core` uses, and can be used to make plugin development easier. This contains the codes for the macros `nadi_plugin`, `env_func`, `node_func`, and `network_func`. This is re-exported by `nadi_core` so that it is not used directly by the users.

25.2.2. `nadi_core`

This is the main/core library for NADI System. This contains the data types, APIs and mechanisms for the Plugin system to work.

25.2.3. `nadi-cli`

This contains the code for generating the `nadi` binary. It uses the `nadi_core` library and just runs the tasks. The binary has command line options (using `clap`) to run in interactive mode, run a file, show help/code for functions, list functions, etc.

25.2.4. `nadi-ide`

This is the main GUI component of NADI system. This provides a graphical interface (written in `iced`), that has tiling window style panes that each have different components that can display information, or let user interact with it.

This also contains codes to generate custom highlight for the DSL, generate custom widget with network diagram, and such.

The `nadi-ide/src/` directory contains multiple rust source code files, many of them are for corresponding component of the GUI window.

25.2.5. `nadi-serve`

This contains experimental server for running NADI Tasks from a browser. It uses `rocket` to open a local server, then listens for requests. The requests to run tasks are run independently of one another. The security of letting users run tasks in the server is not well explored, and might need more experimentation before it can be recommended to be used.

25.2.6. `extra`

This directory contains some extra scripts. Currently it contains codes and examples for syntax highlighing task scripts in sublime, typst, latex, HTML (using javascript), etc.

26. Data Structure

This section will describe the data structures associated with NADI system in brief.

For more accurate and upto date details on the data structures and their available methods. Look at the API reference of `nadi_core` on docs.rs.

26.1. Node

Points with attributes and timeseries. These can be any point as long as they'll be on the network and connection to each other.

The attributes can be any format. There is a special type of attribute timeseries to deal with timeseries data that has been provided by the system. But users are free to make their own attributes and plugins + functions that can work with those attributes.

Since attributes are loaded using TOML file, simple attributes can be stored and parsed from strings, moderately complex ones can be saved as a combination of array and tables, and more complex ones can be saved in different files and their path can be stored as node attributes.

Here is an example node attribute file. Here we have string, float, int and boolean values, as well as a example csv timeseries

```
stn="smithland"
nat_7q10=12335.94850131619
orsanco_7q10=16900
lock=true

[ts.csv]
streamflow = {path="data/smithland.csv", datetime="date", data="flow"}
```

26.2. Network

26.2.1. Network

Collection of Nodes, with Connection information. The connection information is saved in the nodes itself (=inputs= and =output= variables), but they are assigned from the network.

The nadi system (lit, river system), is designed for the connections between points along a river. Out of different types of river networks possible, it can only handle non-branching tributaries system, where each point can have zero to multiple inputs, but can only have one output. Overall the system should have a single output point. There can be branches in the river itself in the physical sense as long as they converse before the next point of interests.

There cannot be node points that have more than one path to reach another node in the representative system.

Network file are simple text files with each edge on one line. Node names can be words with alphanumeric characters with the additional character _, similar to how rust identifiers work. The Node names can also be quoted strings, in those cases any characters are supported inside the quotes.

Here is an example network file,

```
cannelton -> newburgh
newburgh -> evansville
evansville -> "jt-myers"
# comments are supported
"jt-myers" -> "old-shawneetown"
"old-shawneetown" -> golconda
markland -> mcalpine
golconda -> smithland
```

Drawing it out:

```
network load_file("./data/mississippi.net")
network svg_save(
   "./output/mississippi.svg",
  label="[{INDEX}] {_NAME:repl(-, ):case(title)}"
)
network clip()
# the link path needs to be relative to this file
network echo("../output/mississippi.svg")
```

Results:

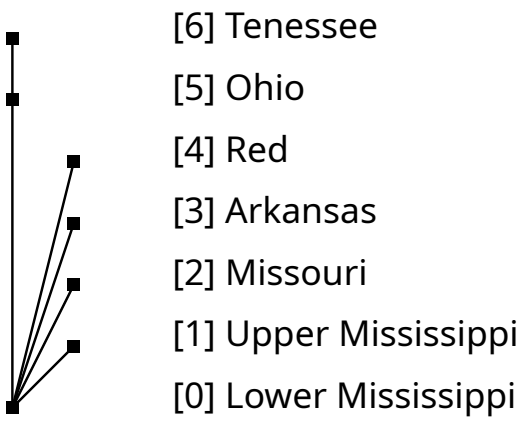


Figure 40:

The program also plans to support the connection import from the DOT format (graphviz package).

Network file without any connection format can be written as a node per line, but those network can only call sequential functions, and not input dependent ones.

Depending on the use cases, it can probably be applied to other systems that are similar to a river system. Or even without the connection information, the functions that are independent to each other can be run in sequential order.

26.3. Timeseries

Timeseries of values, at regular interval. Can support integers, floats, booleans, strings, Arrays and Tables.

For timeseries that are not in a format that NADI can understand. The path to the timeseries can be provided as a node attribute and plugin functions can be written to use that path to load the timeseries for the node.

26.4. String Templates

26.4.1. String Templates

The templating system will be used by an external library developed by me. The library can be modified if there are specific needs for this project.

The template system is feature rich, allowing for formatting, simple string transformations, and airthmatic calculations based on the variables (node attributes in this case). This can be used to generate file paths, and similar strings based on node attributes, as well as to format the cell values for exported table, figures, etc.

The template library is also available for `Rust`, `C` and `C++`, but all the interactions with the templates will be done through the `nadi` interface, so that is not required.

Documentations on the template system, can be redirected to the `string_template_plus` library page.

Brief explanation on the template system is given below.

26.4.1.1. Template Parts

Templates have variables, time formats, expressions, and commands (disabled by default);

```
Hi, my name is {name}, my address is {address?"N/A"}.
Current time is {%H} hour  {%M} minutes.
```

Results (with: name=John; address=123 Road, USA):

```
Hi, my name is John, my address is 123 Road, USA.
Current time is 18 hour  49 minutes.
```

26.4.1.2. Optional Variables

Variables can be chained in an optional way, so the first one that's found will be used (e.g. `{nickname?name}` will render nickname if it's present, else name);

```
Hi, I am {nickname?name}, my address is {address?"N/A"}.
```

Results (with: name=John; nickname=J; address=123 Road, USA):

```
Hi, I am J, my address is 123 Road, USA.
```

26.4.1.3. String Literal

Variables when replaced with literal strings (quoted strings), they will be used directly `{address?"N/A"}` will render `N/A` is `address` is not present;

```
Hi, I am {nickname?name}, my address is {address?"N/A"}.
```

Results (with: name=John):

```
Hi, I am John, my address is N/A.
```

26.4.1.4. Transformers

Variables can have optional transformers which transform the string based on their rules, (e.g. float transformer will truncate the float, upcase will make the string UPPERCASE, etc.);

```
Hi, I am {nickname?name:case(up)}, my address is {address?"N/A"}.
```

Results (with: name=Joe):

```
Hi, I am JOE, my address is N/A.
```

26.4.1.5. Time formats

time formats are formatted current time (e.g. `{%Y}` will become `2024` as of now);

```
Today is {%B %d} of the year {%Y}.
```

Results (with: name=John):

```
Today is August 06 of the year 2025.
```

26.4.1.6. Lisp Expressions

expressions are lisp expressions that will be evaluated and the results will be used. The lisp expression can also access any variables and do any supported programming. (e.g. `(+ 1 1)` in lisp will become `2`);

```
guess my age(x) if: (x + 21) * 4 = =(* (+ (st+num 'age) 21) 4).
```

Results (with: age=20):

```
guess my age(x) if: (x + 21) * 4 = 164.
```

26.4.1.7. NADI Specific options

Besides the above points, specific to nadi system, any node template will have all the variables from node attributes available as strings for template. For string variables, their name can be used to access quoted string format, while their name with underscore prefix will be unquoted raw string. (e.g. if we have attribute `name="smithland"`, then `{name}` will render to `"smithland"`, while `{_name}` will render to `smithland`).

NADI system uses templates in a variety of place, and plugin functions also sometimes take templates for file path, or strings, and such things. Look at the help string of the function to see if it takes `String` or `Template` type.

For example `render` is a function that takes a template and prints it after rendering it for each node.

```
network load_file("./data/mississippi.net")
node[ohio] set_attrs(river="the Ohio River", streamflow=45334.12424343)
node[ohio,red] render(
  "(=(+ 1 (st+num 'INDEX))th node) {_NAME:case(title)}
  River Flow = {streamflow:calc(/10000):f(3)?\"NA\"} x 10^4"
)
```

Results:

```
{
  red = "(5th node) Red\n\tRiver Flow = NA x 10^4",
  ohio = "(6th node) Ohio\n\tRiver Flow = 4.533 x 10^4"
}
```

As seen in above example, you can render variables, transform them, use basic calculations.

Or you can use `lisp` syntax to do more complex calculations. Refer to NADI Extension Capabilities section for more info on how to use `lisp` on string template.

```
network load_file("./data/mississippi.net")
node[ohio] set_attrs(river="the Ohio River", streamflow=45334.12424343)
node[ohio] render(
  "{_river:case(title)} Streamflow
  from lisp = {=(/ (st+num 'streamflow) 1000):f(2)} x 10^3 cfs"
)
```

Results:

```
{
  ohio = "The Ohio River Streamflow\n\tfrom lisp = 45.33 x 10^3 cfs"
}
```

26.4.1.8. Some Complex Examples

Optional variables and a command; note that commands can have variables inside them:

```
hi there, this {is?a?"test"} for $(echo a simple case {that?} {might} be "possible")
```

Results (with: might=may):

```
hi there, this test for $(echo a simple case  may be possible)
```

Optional variables with transformers inside command.

```
Hi {like?a?"test"} for $(this does {work:case(up)} now) (yay)
```

Results (with: work=Fantastic Job):

```
Hi test for $(this does FANTASTIC JOB now) (yay)
```

If you need to use `{` and `}` in a template, you can escape them. Following template shows how LaTeX commands can be generated from templates.

```
more {formatting?} options on {%F} and
\\latex\{command\}\{with {variable}\}, should work.
```

Results (with: command=Error;variable=Var):

```
more  options on 2025-08-06 and
\latex{command}{with Var}, should work.
```

This just combined a lot of different things from above:

```
let's try {every:f(2)?and?"everything"}
for $(a complex case {that?%F?} {might?be?not?found} be "possible")

see $(some-command --flag "and the value" {problem})
=(+ 1 2 (st+num 'hithere) (st+num "otherhi"))
{otherhi?=(1+ pi):f(4)}
```

Error:

```
None of the variables ["might", "be", "not", "found"] found
```

This shows the error for the first template part that errors out, even if `{problem}` will also error later, so while solving for problems in string templates, you might have to give it multiple tries.

26.4.1.9. Advanced String Template with LISP

NADI String Template is useful when you want to represent node specific string, or file path in a network. This is not as advanced as the formatted strings in python. But it can be used for complex situations based on the current functionality.

The most important extension capability of the string template is the embedded lisp system.

As we know, templates can render variables, and have some capacity of transforming them:

```
{name:case(title):repl(-, )} River Streamflow = {streamflow} cfs
```

Results (with: name=Ohio; streamflow=12000):

```
Ohio River Streamflow = 12000 cfs
```

But for numerical operation, the transformers capabilities are limited as they are made for strings.

With lisp, we can add more logic to our templates.

```
{name:case(title):repl(-, )} River Streamflow is =(
  if (> (st+num 'streamflow) 10000)
  'Higher 'Lower
) than the threshold of 10^5 cfs.
```

Results (with: name=Ohio; streamflow=12000):

```
Ohio River Streamflow is Higher than the threshold of 10^5 cfs.
```

The available lisp functions are also limited, but the syntax itself gives us better airthmetic and logical calculations.

26.4.2. Note

As the template string can get complicated, and the parsing is done through Regex, it is not perfect. If you come across any parsing problems, please raise an issue at string template plus github repo.

26.4.3. Commands

Note that running commands within the templates is disabled for now.

```
echo today=$(date +%Y-%m-%d) {%Y-%m-%d}
```

Results (with:):

```
echo today=$(date +%Y-%m-%d) 2025-08-06
```

But if you are writing a command template to run in bash, then it'll be executed as the syntax is similar.

```
network command("echo today=$(date +%Y-%m-%d) {%Y-%m-%d}")
```

Results:

```
$ echo today=$(date +%Y-%m-%d) 2025-08-06
```

Here although the `$(date +%Y-%m-%d)` portion was not rendered on template rendering process, the command was still valid, and was executed.

26.5. Tables

26.5.1. Tables

Tables are data types with headers and the value template. Tables can be rendered/exported into CSV, JSON, and LaTeX format. Other formats can be added later. Although tables are not exposed to the plugin system, functions to export different table formats can be written as a network function.

A sample Table file showing two columns, left aligned name for station in title case, and right aligned columns for latitude and longitude with float value of 4 digits after decimal:

```
network load_file("./data/mississippi.net")
<Name => {_NAME:repl(-, ):case(title)}
^Ind => =(+ (st+num 'INDEX) 1)
>Order => {ORDER}
^Level => {LEVEL}
# something is wrong with the set_level algorithm
# Ohio - tenessee should be level 1, and missouri/yellowstone should be 0
```

Results:

Name	Ind	Order	Level
Lower Mississippi	1	7	0
Upper Mississippi	2	1	1
Missouri	3	1	1
Arkansas	4	1	1
Red	5	1	1
Ohio	6	2	0
Tenessee	7	1	0

Here the part before => is the column header and the part after is the template. Presence of < or > in the beginning of the line makes the column left or right aligned, with center aligned (^) by default.

Exporting the table in svg instead of markdown allows us better network diagram.

```
network load_file("./data/mississippi.net")
network echo("../output/example-table2.svg")
<Name => {_NAME:repl(-, ):case(title)}
^Ind => =(+ (st+num 'INDEX) 1)
>Order => {ORDER}
^Level => {LEVEL}
```

Error:

```
network function: "table_to_svg" not found
```

A SVG Table can also be generated using the table file, using the task system like this:

```
network load_file("./data/mississippi.net")
network table_to_svg(
  table = "./data/sample.table",
  # either table = "path/to/table", or template = "table template"
  outfile = "./output/example-table.svg",
  config = {fontsize = 16, delta_y = 20, fontface="Noto Serif"}
)
network clip()
# the link path needs to be relative to this file
network echo("../output/example-table.svg")
```

Error:

```
network function: "table_to_svg" not found
```

26.6. File Templates

File templates are templates that use string templates, but they are a whole file that can be used to generate rendered text files.

File templates also have sections which can be repeated for different nodes, with corresponding syntax.

Following template will render a markdown table with headers and all the name and index of the nodes.

```
| Node | Index |
|------|-------|
<!-- ---8<--- -->
| {_NAME} | {INDEX} |
<!-- ---8<--- -->
```

26.7. Tasks

Task is a function call that the system performs. The function call can be a node function or a network function. The function can have arguments and keyword arguments that can determine its functionality. Node functions will be called on a node at a time, while the network function will be called with the whole network at once.

Currently tasks are performed one after another. The functions that any task can use can be internal functions provided by the library or the external functions provided by the plugins.

A sample tasks file is shown below:

```
node print_attrs()
network save_graphviz("/tmp/test.gv", offset=1.3, url="{_NAME}")
node savedss(
  "natural",
  "test.dss",
  "/OHIO-RIVER/{_NAME}/01Jan1994/01Jan2012/1Day/NATURAL/"
  )
node check_sf("sf")
node.inputsfirst route_sf("observed")
node render("Node {NAME} at index {INDEX}")
```

Here each line corresponds to one task. And if it's a node task, then it'll be called for each node (in sequential order by default). The last line `node.inputsfirst` will call that function in input node before the current node. Those functions can only be called for network with an output node.

Please note that although the string in the examples are highlighted as if they are string templates for readability. Those are just normal strings that functions take as inputs. Whether they are used as template or not depends on the individual function, refer to their help to see if they take `Template` type or `String` type.

26.8. Node Functions

Node functions are functions that take a node, and the function context to do some operations on it. They take mutable reference to the node, hence can read all node attributes, inputs, outputs, their attributes and timeseries.

Node functions can be run from the system for all the nodes in the network in different orders.

Currently the task system only supports running node functions for all nodes in the following 6 ways,

- Sequential order,
- Reverse order,
- Run input nodes before the current node (recursively),
- Run output node before the current node (recursively),
- Run a list of nodes, and
- Run on a path between two nodes (inclusive).

Depending on the way the function works, it might be required to be run in a particular order. For example, a function that counts the number of dams upstream of each point, might have to be run inputs first, so that you can cumulate the number as you move downstream.

26.9. Network Functions

Network functions are functions that take the network as a mutable reference and run on it.

Some examples of network functions:

- List all the networks with their inputs/outputs,
- Checks if any nodes have some attribute larger than their output,
- Export the node attributes as a single CSV file,
- Export the nodes in LaTeX file using Tikz to draw the network,
- Calculate rmse,mse,etc errors between two attribute values for all nodes,

- Generate an interactive HTML/PDF with network information and some other template, etc.

Developer Notes and Future Direction

27. Developer Notes

This section contains my notes as I develop the NADI system. Kind of like a dev blog.

The software package will consists of multiple components. It is planned to be designed in such a way that users can add their functionality and extend it with ease.

Along with the Free and Open Source Software (FOSS) principles, the plugin system will make extension of the software functionality and sharing between users. As well as a way to develop in-house functionality for niche use cases.

27.1. Motivation

As Hydrologist, we often deal with the data related to the points in the river. Since most of the analysis requires doing the same things in multiple points, the initial phase of data cleaning process can be automated.

We spend a beginning phases of all projects preparing the data for analysis. And combining the time spent on visualizing the data, it’s a significant chunk of our time.

Data visualization influences the decision making from the stakeholders. And can save time by making any problems obvious from the very beginning. For examples, things like showing the quality of data (continuity for time series), interactive plots to compare data in different locations/formats, etc can help people understand their data better.

Besides plot, the example below shows how simply adding a column with connection visual can immediately make it easier to understand the relationship between the data points in a river. Without it people need to be familiar with the names of the data points and their location, or consult a different image/map to understand the relationship.

Connection	Node	7Q10		1Q10	
		Observed	Naturalized	Observed	Naturalized
40	sutersville	499.4	139.3	427.3	2.7
39	elizabeth2	884.6	223.1	403.1	3.1
38	natrona	2794.0	899.0	1360.8	31.8
37	emsworth	4530.5	1554.2	3928.7	422.9
36	dashields	4534.3	1566.8	3990.4	421.8
35	beaverfalls	603.2	194.2	493.1	31.6
34	montgomery	5340.3	2012.2	4877.8	969.0
33	new-cumberland	5350.3	2026.1	4925.4	1071.2
32	pike-island	5364.0	2048.7	5047.3	1251.9
31	hannibal	5375.5	2073.3	5048.7	1400.1
30	willow-island	5388.0	2092.8	5177.8	1611.7

Figure 41: Table with Connection Information

The inspiration on making this software package comes from many years of struggle with doing the same thing again and again in different projects like these. And the motivation to make something generic that can be used for plethora of projects in the future.

27.2. Why Rust?

Rust is an open source programming language that claims to be fast and memory efficient to power performance critical services. Rust is also able to integrate with other programming languages.

Rust provides a memory safe way to do modern programming. The White House has a recent press release about the need to have memory safe language in future softwares. The report has following sentense about the Rust language.

> At this time, the most widely used languages that meet all three properties are C and C++, which are not memory safe programming languages. Rust, one example of a memory safe programming language, has the three requisite properties above, but has not yet been proven in space systems.

The results of the survey from stackoverflow shows Rust has been a top choice for developers who want to use a new technology for the past 8 years, and the analysis also shows Rust is a language that generates for desire to use it once you get to know.

https://www.rust-lang.org/

https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/press-release-technical-report/

https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

https://survey.stackoverflow.co/2023/#technology-admired-and-desired

27.3. Plugin System Experiments

27.3.1. First Prototype: cdylib

27.3.2. Second Prototype: abi_stable

27.4. Writing this Book

27.4.1. Writing this Book

I'm used to `emacs`'s `org-mode`, where you can evaluate code and show output and all those things. Like `markdown` in steroids.

`mdbook` seems to have some of those functionality in it as well. Though I think `emacs`'s extension through `elisp` is lot more flexible and easier to extend. `mdbook` supporting custom preprocessors and renderer means we can extend it as well.

In the process of writing this book. I made the following things.

27.4.1.1. Syntax Highlight for NADI specific syntax

`mdbook` uses `highlight.js` to syntax highlight the code blocks in it. And since nadi system has a lot of its own syntax for string templates, task system, table system, network system etc. I wanted syntax highlight for those things. Although the attribute files are subset of `TOML` format, so we have syntax highlight for it. Everything else needed a custom code.

Following the comments in this github issue led me to find a workaround for the custom syntax hightlight. I don't know for how long it will work, but this works well for now.

Basically I am using the custom JS feature of `mdbook` like:

```
[output.html]
additional-js = ["theme/syntax-highlight.js"]
```

To insert custom highlight syntax. For example adding the syntax highlight for network text is:

```
// network connections comments and node -> node syntax
hljs.registerLanguage("network", (hljs) => ({
    name: "Network",
    aliases: [ 'net' ],
    contains: [
  hljs.QUOTE_STRING_MODE,
  hljs.HASH_COMMENT_MODE,
  {
      scope: "meta",
      begin: '->',
      className:"built_in",
  },
    ]
}));
```

The syntax for network is really simple, for others (`task`, `table`, `string-template`, etc) refer to the `theme/syntax-highlight.js` file in the repository for this book.

After registering all the languages, you re-initialize the `highlight.js`:

```
hljs.initHighlightingOnLoad();
```

27.4.1.2. `mdbook-nadi` preprocessor

Instead of just showing the syntax of how to use the task system, I wanted to also show the output of the examples for readers. So I started this with writing some `elisp` code to run the text in selection and then copying the output to clipboard that I could paste in output block. It was really easy in emacs.

Following code takes the selection, saves them in temporary tasks file, runs them and then puts the output in the clipboard that I can paste manually.

```
(defun nadi-run-tasks (BEG END)
  (interactive "r")
```

```
  (let ((tasks-file (make-temp-file "tasks-")))
    (write-region BEG END tasks-file)
    (let ((output '(shell-command-to-string (format "nadi %s" tasks-file))))
    (message output)
    (kill-new output)
    (delete-file tasks-file))))
```

But this is manual process with a bit of automation. So I wanted a better solution, and that's where the `mdbook` preprocessor comes in.

With the `mdbook-nadi` preprocessor, I can extract the code blocks, run it, and insert the contents just below the code block as output.

Once I had a working prototype for this, I also started adding support for rendering string templates, and generating tables along with the task system.

27.4.1.2.1. String templates

For string templates, write the templates in `stp` blocks like below that will have the syntax hightlight.

```
Hi my name is {name}.
```

If you add `run` into it, it'll run the template with any `key=val` pairs provided after run.

Basically writing the following in the `mdbook` markdown:

````
```stp run name=John
Hi my name is {name}.
```
````

Will become:

```
Hi my name is {name}.
```

Results (with: name=John):

```
Hi my name is John.
```

27.4.1.2.2. Tasks

For tasks, similary write a block with `task` as language. You can use `!` character at the start of the line to hide it in the view. Use them for essential code that are needed for results but are not the current focus. And when you add `run` it'll run and show the output.

````
```task run
network load_file("data/mississippi.net")
node render("Node {NAME}")
```
````

```
network load_file("data/mississippi.net")
node render("Node {NAME}")
```

Results:

```
{
  lower-mississippi = "Node \"lower-mississippi\"",
  upper-mississippi = "Node \"upper-mississippi\"",
  missouri = "Node \"missouri\"",
  arkansas = "Node \"arkansas\"",
  red = "Node \"red\"",
  ohio = "Node \"ohio\"",
  tenessee = "Node \"tenessee\""
}
```

27.4.1.2.3. Tables

The implementation for tables are little weird right now, but it works. Since we need to be able to load network, and perform actions before showing a table.

So the current implementation takes the hidden lines using `!`and runs them as task system, with additional task of rendering the table at the end.

Example:

````
```table run markdown
network load_file("./data/mississippi.net")
<Name => {_NAME:repl(-, ):case(title)}
^Ind => =(+ (st+num 'INDEX) 1)
```
````

```
>Order => {ORDER}
```

Becomes:

```
network load_file("./data/mississippi.net")
<Name => {_NAME:repl(-, ):case(title)}
^Ind => =(+ (st+num 'INDEX) 1)
>Order => {ORDER}
```

Results:

Name	Ind	Order
Lower Mississippi	1	7
Upper Mississippi	2	1
Missouri	3	1
Arkansas	4	1
Red	5	1
Ohio	6	2
Tenessee	7	1

I'd like to refine this further.

Task can be used to generate markdown in the same way as the tables can:

For example `task run` of this:

```
network load_file("./data/mississippi.net")
network table_to_markdown(template="
<Name => {_NAME:repl(-, ):case(title)}
^Ind => =(+ (st+num 'INDEX) 1)
>Order => {ORDER}
")
```

Results:

```
| Name              | Ind | Order |
|:------------------|:---:|------:|
```

```
| Lower Mississippi |  1  |     7 |
| Upper Mississippi |  2  |     1 |
| Missouri          |  3  |     1 |
| Arkansas          |  4  |     1 |
| Red               |  5  |     1 |
| Ohio              |  6  |     2 |
| Tenessee          |  7  |     1 |
```

If you do `task run markdown` then:

```
network load_file("./data/mississippi.net")
network table_to_markdown(template="
<Name => {_NAME:repl(-, ):case(title)}
^Ind => =(+ (st+num 'INDEX) 1)
>Order => {ORDER}
")
```

Results:

Name	Ind	Order
Lower Mississippi	1	7
Upper Mississippi	2	1
Missouri	3	1
Arkansas	4	1
Red	5	1
Ohio	6	2
Tenessee	7	1

Which means it can be used for other things:

```
network load_file("./data/mississippi.net");
network echo("**Details about the Nodes:**")
network echo(render_nodes("
=(+ (st+num 'INDEX) 1). {_NAME:repl(-, ):case(title)} River
"))
```

Results:

Details about the Nodes:

1. Lower Mississippi River
2. Upper Mississippi River
3. Missouri River
4. Arkansas River
5. Red River
6. Ohio River
7. Tenessee River

You can also use the same method to insert images like this, at the end of your tasks, so that the image generated by the tasks can be inserted here.

```
# do some tasks
network echo("Some other output form your tasks")
network clip()
network echo("../images/ohio-low.svg")
```

Results:

		7Q10		1Q10	
Connection	Node	Observed	Naturalized	Observed	Naturalized
40	sutersville	499.4	139.3	427.3	2.7
39	elizabeth2	884.6	223.1	403.1	3.1
38	natrona	2794.0	899.0	1360.8	31.8
37	emsworth	4530.5	1554.2	3928.7	422.9
36	dashields	4534.3	1566.8	3990.4	421.8
35	beaverfalls	603.2	194.2	493.1	31.6
34	montgomery	5340.3	2012.2	4877.8	969.0
33	new-cumberland	5350.3	2026.1	4925.4	1071.2
32	pike-island	5364.0	2048.7	5047.3	1251.9
31	hannibal	5375.5	2073.3	5048.7	1400.1
30	willow-island	5388.0	2092.8	5177.8	1611.7

Figure 42:

28. Future Ideas to Implement

28.1. Optimization Algorithms

28.1.1. Optimization Algorithms

We can have input variables to change, and output variables to optimize, but how do we take what function to run to calculate the output variable...

One simple idea can be to take a command template to run. So we will change the input variables, run the command for each node or network, and then that command will update the output variable that we can optimize for.

We might require an option to call other functions in this case. Then maybe we can just pass the name of the function.

Complex idea could be to add the support for loop syntax in task system.

28.1.2. While loop

Visiting this page was fun because I ended up implementing a while loop and the if-else statement.

Now, as long as you have functions to calculate error metrics from your variables/network you should be able to run optimization using the brute-force method.

28.2. Interactive Plots

An experiment using the `cairo` graphics library shows that a PDF can be directly produced without using LaTeX as intermediate using the network information. This functionality — although not as complete as the one in the example — has been exposed as an internal network function for now. Further functionality related to this idea can be embedding network information in simple plots, or generate the whole plot along side the network information.

It might be a good idea to make several functions that can export the interactive plots in LaTeX, PDF, PNG, SVG, HTML, etc. separately instead of single format.

LaTeX and HTML will be easier due to text nature, for others I might have to spend time with some more experimentation on cairo.