## 1 Documentation structure

1.1 Vis-Tools overview ........................................... 3
   1.1.1 Structure ........................................... 3
   1.1.2 COMPS usage ......................................... 4
   1.1.3 Python preprocessing ................................. 4
   1.1.4 Visset ............................................... 5
   1.1.5 Local web server ..................................... 5
   1.1.6 Clients ............................................... 5

1.2 Prerequisites ................................................. 5
   1.2.1 Computer requirements ............................... 5
   1.2.2 Using Vis-Tools standalone ......................... 6
   1.2.3 Supported web browsers .............................. 7

1.3 Quick starts .................................................. 7
   1.3.1 Vis-Tools in COMPS ................................ 7
   1.3.2 Vis-Tools Standalone ............................... 12

1.4 Preprocessing simulation data ............................. 28
   1.4.1 Preprocessing cycle ................................ 28
   1.4.2 Components of a preprocessing script ........... 32
   1.4.3 Preprocessing classes .............................. 42

1.5 Geospatial visualization .................................... 79
   1.5.1 Running the Geospatial client ..................... 80
   1.5.2 URL and visset ...................................... 80

1.6 Glossary ..................................................... 109

### Python Module Index

111

### Index

113
The Institute for Disease Modeling (IDM) develops complex software for disease modeling. The primary software, Epidemiological MODeling software (EMOD), helps determine the combination of health policies and intervention strategies that can lead to disease eradication. The effective use of EMOD requires analysis of simulation output. Vis-Tools is a collection of software tools that aids in the visualization of geospatial simulation data.
1.1 Vis-Tools overview

Welcome to the IDM Vis-Tools visualization tool set. This page will provide an overview of the structure of Vis-Tools and the inter-operation of its parts. The following pages will provide information on the use of Vis-Tools for advanced simulation data visualization.

1.1.1 Structure

Vis-Tools can be invoked either from the COMPS web client or for more advanced use, run on your local computer. When used standalone, the Vis-Tools files can reside wherever convenient.

Vis-Tools consists of two main parts - a Python package that allows you to preprocess EMOD simulation inputs and outputs into a “visset” file, and a browser-based client to allow visualization of the simulation data. For standalone use, Vis-Tools also contains a simple Python web server that you run to serve up the HTML visualization component.

Fig. 1: Structure of Vis-Tools.
1.1.2 COMPS usage

For spatial simulations run within the COMPS system, Vis-Tools can be invoked from the Spatial tab of COMPS’ explore view.

For more information about running Vis-Tools from COMPS, see the Quick starts.

1.1.3 Python preprocessing

The Vis-Tools visualization client uses a visset file that describes how the simulation data is to be displayed and contains a subset of the simulation’s demographics data (specifically, information about its nodes). The visset file is obtained by running a preprocessing script on your simulation’s input and output files.

You can write this preprocessing script from scratch, but more likely you’ll let Vis-Tools create one for you. The Vis-Tools survey program can look at your input and output files and create a default preprocessing script to get you started.

The preprocessing script will have at least one output, which is the visset JSON file. That file will likely reference files in your simulation’s output folder as well.
1.1.4 Visset

The visset file contains references to all the files necessary to visualize your simulation. It also includes the default settings for the visualization client page. If you so desire, you can have more than one visset for your simulation, perhaps to highlight different aspects of the simulation’s output.

1.1.5 Local web server

If you are running Vis-Tools standalone, the data to be visualized will already be on your local computer. In order to serve that data up to the visualization client, Vis-Tools contains a simple Python web server. This web server is by default on port 8000 of your own computer, “localhost”.

1.1.6 Clients

Vis-Tools is designed to support multiple visualization clients. Currently, however, there is only one client, called “Geospatial,” which is primarily designed for the visualization of spatio-temporal simulation output. In the future other clients will be provided for different data visualization scenarios.

1.2 Prerequisites

Vis-Tools, like many visualization systems, makes relatively high demands on your computer and video graphics processor (GPU).

Note: Vis-Tools is only tested on the Microsoft Windows platform. It may work on Linux and MacOS, but its use on those operating systems is not supported by IDM.

1.2.1 Computer requirements

Minimum configuration

- Hardware-accelerated WebGL
- 2 GB RAM

Recommended configuration

- Discrete nVidia or AMD video graphics accelerator
- Hardware-accelerated WebGL2
- 4 GB RAM

Note: One quick way to see if your computer has hardware-accelerated WebGL is to run the Chrome web browser and navigate to chrome://gpu. In the Graphics Feature Status section, look for WebGL and WebGL2.
### Graphics Feature Status

- **Canvas**: Hardware accelerated
- **Checker Imaging**: Disabled
- **Flash**: Hardware accelerated
- **Flash Stage3D**: Hardware accelerated
- **Flash Stage3D Baseline profile**: Hardware accelerated
- **Compositing**: Hardware accelerated
- **Multiple Raster Threads**: Enabled
- **Native GpuMemoryBuffers**: Software only. Hardware acceleration disabled
- **Rasterization**: Hardware accelerated
- **Video Decode**: Hardware accelerated
- **Video Encode**: Hardware accelerated
- **WebGL**: Hardware accelerated
- **WebGL2**: Hardware accelerated

---

### Additional software

- Google Chrome, Mozilla Firefox, or Opera web browser
- NodeJS

### 1.2.2 Using Vis-Tools standalone

To run Vis-Tools on your local computer, there are a few additional prerequisites for operation of Vis-Tools. If you are already running EMOD, you likely already meet the requirements.

#### Additional software

- Python 3.6.3+ with Python in your PATH environment variable

   - In the **Customize Python** dialog box, verify that **Add python.exe to PATH** is selected to add Python to the PATH environment variable on your computer.

2. Open a Command Prompt window and type the following to verify installation:

```
python --version
```

The Python package manager, pip, is installed as part of Python 3.6 64-bit and is used to install other software packages.

- NodeJS

1. Install NodeJS. See [NodeJS.org](http://NodeJS.org).

2. Open a Command Prompt window and type the following to verify installation:

```
node -v
npm -v
```
1.2.3 Supported web browsers

Vis-Tools supports the Google Chrome, Mozilla Firefox, and Opera web browsers, although it has been most extensively tested on Google Chrome. These browsers can be obtained via the following links.

- To install Google Chrome, see Google Chrome
- To install Mozilla Firefox, see Mozilla Firefox
- To install Opera, see Opera

1.3 Quick starts

This section provides two quick introductions to Vis-Tools, one running Vis-Tools from within the COMPS system, the other running standalone and doing a visualization of an existing set of sample data that is included in the Vis-Tools distribution. The latter quick start also contains an abbreviated treatment of the preprocessing of simulation data.

Be sure to review the Prerequisites before proceeding.

1.3.1 Vis-Tools in COMPS

Running Vis-Tools from COMPS

This page gives a quick overview of how to run Vis-Tools from within the COMPS system.

**Note:** To access and use COMPS you must receive approval and credentials from IDM. Send your request to support@idmod.org.

To use Vis-Tools this way, you’ll need to create or otherwise have access to a spatial simulation, which is to say, a simulation with multiple nodes that have latitude/longitude geospatial locations.

**Locate a target simulation in COMPS**

1. Log into COMPS using your account credentials
2. Navigate to Explore > Simulations
3. Locate a spatial simulation for visualization. When you select such a simulation, you’ll see the following in the Spatial tab of Explore’s Details pane.

**Preprocess**

1. Click the Preprocess button to launch a COMPS work item to perform preprocessing on the simulation. Preprocessing for Vis-Tools will not modify or damage the simulation in any way. After you click the Preprocess button, the Spatial tab will change to this:

2. After the preprocessing work item completes successfully, you’ll see the spatial tab change again:
Chapter 1. Documentation structure

Simulation contains spatial output, but requires preprocessing for visualization.

- 746 nodes
- 365 timesteps
- 11 spatial reports
- Visualization complexity factor: 6.48

Simulation is currently preprocessing.

- 746 nodes
- 365 timesteps
- 11 spatial reports
- Visualization complexity factor: 6.48
**Visualize**

1. Click the **Visualize** button to launch Vis-Tools on the simulation’s data.

The preprocessing script created a visset that lets us see our nodes on the map, with no bindings between our simulation’s outputs and the visual parameters of the visualization at all. It’s a good starting point for us to customize the visualization to our needs.

To learn more about the Geospatial visualization client, see *Geospatial visualization*.

**Re-preprocess**

You may find need to “re-preprocess” a simulation. For example, if you’ve already run Vis-Tools preprocessing for a “Points”-type visualization, you might change your mind and want to preprocess for “Shapes.”

1. In the COMPS Spatial tab, click the **Re-preprocess** button. COMPS will return you to the preprocessing options view, wherein you may re-preprocess or upload your own preprocessing script.

**Using your own preprocessing script**

COMPS can create default “Points” or “Shapes” preprocessing files for you. But more advanced users may choose to modify a default script to their liking and preprocess using that modified script.

1. In the COMPS Spatial tab, run **Preprocess** with either “Points” or “Shapes” to have a default preprocessing script created for you.

2. Navigate to the COMPS Output tab. Therein, find the Vis-Tools folder among your simulation’s output files and find the vt_preprocess.py script that was create by the Preprocess operation.

3. Right-click the **Download** icon next to that file and choose Save Link As… to save it to your local computer.

4. Open the vt_preprocess.py file in an editor and modify it to your liking. For example, you might add default bindings or change the base map. See the comments within the vt_preprocess.py file for examples of changes to make.

5. When you’re done making changes, save the file as “my_preprocess.py” to any convenient location.

6. Back in the COMPS Spatial tab, click **Re-preprocess**.

7. Click the **Custom script…** button. This will open a file open dialog. Locate and open the my_preprocess.py script you saved earlier.

8. Click **Preprocess** to run preprocessing with your custom script.
Simulation contains spatial output, but requires preprocessing for visualization.

- 746 nodes
- 365 timesteps
- 11 spatial reports
- Visualization complexity factor: 6.48

Preprocessing script:
- New Point script
- New Shape script
- Custom script...

Preprocess
1.3. Quick starts
1.3.2 Vis-Tools Standalone

Running Vis-Tools Standalone

This page walks you through the steps to download and install Vis-Tools, start the local web server that serves the visualizations, and trying out a sample visualization included with the installation.

Download

Download the Vis-Tools archive from Vis-Tools Distribution on GitHub.

Unzip

The first part of Vis-Tools installation a matter of unzipping this distribution ZIP file to a convenient location, such as your computer’s desktop.

1. Right-click the Vis-Tools zip file.
2. Select Extract All from the context menu.
3. In the Extract Compressed (Zipped) Folders dialog box, use the Browse button to browse to the desired location on your computer’s local hard drive.
4. Click Extract.
Note: Vis-Tools will not work if installed to a network UNC path (that is, a path starting with “\.”).

Note: It is highly recommended that you install Vis-Tools to a local drive location. Vis-Tools will work when installed on a mapped network drive, but performance will be impacted.

**Install vis_tools Python package**

Vis-Tools preprocessing is taken care of by classes in the included vis_tools Python package.

1. Choose Start > Command Prompt to get a cmd.exe command prompt window
2. Change directory to the location where you unzipped the Vis-Tools distribution zip file.
3. Run the command:

   ```
   python setup.py install
   ```

   That command will install the prerequisite packages and the vis_tools package itself.
4. Verify installation of the package by running the command:

   ```
   survey --help
   ```

   Which will show the options for the built-in survey program, which makes default preprocessing scripts for you.

**Prepare the Geospatial client**

The Vis-Tools client uses several third-party libraries that are not included with the distribution.

1. Choose Start > Command Prompt to get a cmd.exe command prompt window
2. Change directory to the location where you unzipped the Vis-Tools distribution zip file.
3. Run the commands:

   ```
   cd vistools
   npm install
   ```

   That command will install the third-party Javascript libraries into a node_modules folder.

**Start the web server**

Because Vis-Tools visualizations run in a web browser, standalone Vis-Tools is served from a small local web server that runs on your computer. It only serves up the files within the Vis-Tools folder. In this section we will start that web server.

1. Open the folder in which you extracted the Vis-Tools archive in the previous step.
2. Double-click the file `server.bat`.
3. You will see a console window open that looks like this:
Chapter 1. Documentation structure

Vis-Tools

- server.py
- setup.py
- .gitattributes
- .gitignore
- server.bat

Type: Windows Batch File
Size: 1.17 KB
Date modified: 7/9/2018 2:01 PM

Vis-Tools serving on port 8000...
If you get an error starting the web server, check **Troubleshooting**.

Normally you will want to leave this command window open. To get it out of the way, you can click the minimize button its title bar. To stop the web server at any time, simply close the window with the × button in its title bar.

**Try the web client**

In the previous step, you started a local web server that serves the files within the Vis-Tools folder to a web browser. That step, if successful, also opens a web browser window to the URL `http://localhost:8000/vistools/geospatial.html?set=/zambia_data/Vis-Tools/shapes_demo/visset.json`.

If that works, you should see a window that looks like this:

![Geospatial visualization window](image)

Here’s a quick overview of the parts of the Geospatial visualization window:

To introduce you to the client’s capabilities, try the following in the Geospatial visualization client:

- Try dragging the blue time indicator around in the timeline at the bottom of the screen. This simulation has some interesting activity near the end, around timestep 3000.
- Try navigating the view in 3D.

**1.3. Quick starts**
Click-drag in the map will pan along the surface.

Ctrl-drag in the map will change the view angle.

Mouse wheel zooms in or out.

- Try playing animation by clicking the Play button in the Transport area at the top of the window.
- Try turning layers on and off with the layer checkboxes in the left panel.
- Try changing bindings between available simulation data and visual parameters.

For detailed information about how the Geospatial client works, see Geospatial visualization.

Visualize a demo simulation

Note: This section describes the visualization of a demonstration simulation that is part of the standalone Vis-Tools distribution. If you haven’t already installed Vis-Tools standalone, see Running Vis-Tools Standalone

For a simulation to be visualized with Vis-Tools, a preprocessing step needs to be run that creates a visset file that drives the visualization. This section will show you the process of going from bare simulation data to a visualization using the built-in demo data that comes with the Vis-Tools standalone distribution.
Obtain your simulation data

The first step is to obtain the simulation’s data, both input and output. The following are typically required:

- The configuration file (config.json) for simulation duration, demographics file name, etc.
- A demographics file (demographics.json) for node latitude and longitude locations, etc.
- The whole output directory for spatial and other reports

(If these files and directories are not familiar, see Introduction to the software for more information.)

For this example, we’re going to use the demo data that is already present in the Vis-Tools distribution folder. To see what’s inside, double-click the zambia_data directory.

You can see it contains both input files such as config.json and campaign.json in addition to an output directory that contains a number of binary and CSV reports generated during the simulation run. Many of these files will be referenced in the following step.

Note: Some demographics files have “.compiled.json” at the end their names. Such files use a dictionary to reduce the overall size of the demographics file. Vis-Tools can consume both compiled and non-compiled demographics files.

Run Survey

The Vis-Tools package contains a Python program called survey that can create a default preprocessing script for you. In this step we’ll run that program and take a look at the preprocessing script it created.

1. Open a command prompt window within the Vis-Tools directory. The easiest way to do this is to Shift-right-click an empty space in the Vis-Tools directory window and select Open command window here from the context menu.

2. The survey program needs to be run from within a simulation’s directory, so first let’s enter the zambia_data directory with this command:

   ```
   cd zambia_data
   ```

3. Now run survey:

   ```
   survey
   ```

You will see some output like this:

For our example, we can see survey has determined that our sim has 3,650 timesteps and 746 nodes, and has written a preprocessing script for us called preprocess_sim.py.

Note: The preprocessing script has been written into the oddly-named directory Vis-Tools/6bac0b83ebe278e86790150241b834. Vis-Tools concentrates all of its outputs in a Vis-Tools directory to avoid unnecessarily polluting the simulation directory. The long hexadecimal directory is actually the MD5 hash of the preprocess_sim.py script. This helps Vis-Tools keep different runs of survey (with potentially different command-line options) from overwriting each other.

Before we run it, let’s take a look at the default preprocessing script. survey has created a script based on the available data in this simulation. It contains sections that are commented out to suggest possible additions or changes to customize the script to your particular needs.

1.3. Quick starts
# Generated by Survey.py

# This script is meant to be run from within the target simulation directory.

```python
import time as tm
from os import path, getcwd
from vis_tools import VisSet, SpatialBinary, SpatialReports

# Constants
k_name = path.basename(getcwd())
k_sim_dir = r'.'
k_sim_output_dir = r'.\output'
k_products_dir = path.join(k_sim_dir, path.relpath(path.dirname(path.abspath(__file__)), getcwd()))
k_demographics_path_or_list = r'.\Bbondo_filled_calib_29.842464_prop.json'
k_sim_url_root = '/' + path.basename(getcwd()) + '/'
k_spatial_report_dir = k_sim_output_dir
k_config_path = r'.\config.json'
k_event_recorder_path = r'.\output\ReportEventRecorder.csv'
k_human_migration_path = r'.\output\ReportHumanMigrationTracking.csv'
k_malaria_filtered_path = r'.\output\ReportMalariaFiltered.json'

# Appearance-related constants
k_icon_duration_days = 2
```

# Main program
```
if __name__ == '__main__':
    script_start = tm.time()

    # Create a default VisSet
    vis_set = VisSet(k_name)

    # Set up the VisSet
    vis_set.set_target_client('Geospatial')
    vis_set.set_products_directory(k_products_dir)
    vis_set.set_config(k_config_path)
```

(continues on next page)
vis_set.set_demographics(k_demographics_path_or_list)
vis_set.set_node_vis_type('Points')

# Remove the 'elsewhere' node from the node stats. This doesn't remove the
# node, just leaves it out of min/max stats. Do this before adding the
# spatial binaries, so as to also exclude that node from the spatial stats.
# vis_set.exclude_node_from_stats(1001)

# Inset chart
vis_set.set_inset_chart(k_malaria_filtered_path)
# vis_set.options['insetCharts']['defaultChannelName'] =
# 'Infectious Vectors'

# Add extra NodeAttribute fields
vis_set.add_node_field('InitialPopulation')
# vis_set.add_node_field('BirthRate')

# Create a SpatialReports object that scans the simulation output directory
# looking for SpatialReport*_*.bin files. Later we'll add this to the VisSet
# object.
reports = SpatialReports(k_spatial_report_dir)

# Example of creating an infected vectors count spatial binary by combining
# SpatialReport_Adult_Vectors with SpatialReport_Infectious_Vectors. Once
# created, we manually add the new report to the reports object
# def combine(a, b):
#    return int(a * b)
# print('Synthesizing Infectious Vectors Count spatial binary...')
# infected_vectors_count = SpatialBinary.combine(
#    path.join(k_spatial_report_dir, 'SpatialReport_Adult_Vectors.bin'),
#    path.join(k_spatial_report_dir, 'SpatialReport_Infectious_Vectors.bin'),
#    'Infectious Vectors Count',
#    combine)
# report_path = path.join(k_products_dir,
#    'SpatialReport_Infectious_Vectors_Count.bin')
# infected_vectors_count.write_binary(report_path)
# reports.add(report_path)

# Add spatial reports (also establishes timestep count)
vis_set.add_spatial_reports(reports)

# Include only a subset of available spatial reports. If you have a lot of
# spatial reports, you may run out of memory on the client side. So here you
# can easily decide which channels are important and include just those
# channels. All available channels are shown below, so just delete the lines
# for channels you don't care about.
vis_set.include_spatial_channels(
    ['SpatialReport_Adult_Vectors',
     'SpatialReport_Daily_Bites_Per_Human',
     'SpatialReport_Daily_EIR',
     'SpatialReport_Human_Infectious_Reservoir',
     'SpatialReport_Infectious_Vectors',
     'SpatialReport_Mean_Parasitemia',
     'SpatialReport_New_Clinical_Cases',
     'SpatialReport_New_Diagnostic_Prevalence',
     'SpatialReport_New_Infections',
     'SpatialReport_Population',
)
Run the preprocessing script

Now, in that same command prompt window, run the newly generated preprocessing script. Use the command:

```
python Vis-Tools\6bac0b833ebec278e86790150241b834\preprocess_zambia_data.py
```

Running this command, we get the following output in the command window:

The preprocessing script emits various status information as it processes the simulation’s files, but ultimately emits a new file called a visset, which is located in the same directory as the preprocessing script itself. It also emits the URL we can use to visualize the simulation’s data, which in our example is:

```
1.3. Quick starts
```
Copy that URL for the next step.

**Run the Geospatial visualization client**

If you don’t already have the local web server running, run it now by double-clicking the server.bat file in the Vis-Tools directory.

Once you have a browser window up, paste in the URL from the preprocessing run’s output. In the case of our example, we see the following:

The preprocessing script created a visset that lets us see our nodes on the map, with no bindings between our simulation’s outputs and the visual parameters of the visualization at all. It’s a good starting point for us to customize the visualization to our needs.

To learn more about the Geospatial visualization client, see *Geospatial visualization*.

**Troubleshooting**

If you run into trouble while installing or using standalone Vis-Tools, check to see if your problem is listed here.

**Command window message when double-clicking server.bat**

If server.bat can’t run the python web server, it may show a diagnostic message. This section clarifies those.
1.3. Quick starts
Python not in PATH

Python is not in your PATH environment variable. Please re-run the Python installer and make sure 'Add python.exe to Path' is enabled in the 'Customize Python' page of the installer.

This message indicates that Python is not available to server.bat. The simplest solution is to reinstall Python as described. Alternately, you can manually edit the PATH environment variable to add the path to the Python executable.

Run from UNC location

It is likely that you have run server.bat from an Explorer window that is pointed at a UNC path (that is, one that starts with two \). The windows command prompt does not support UNC paths. If the UNC path corresponds to a mapped drive, open an Explorer window to the mapped drive letter and try again. It is recommended that you unzip Vis-Tools to a local (non-network) location for best performance.

This message indicates that server.bat thinks you’re running from a network location. (It infers that from the current directory, which will be your Windows directory if you try to run server.bat from a network location.) Vis-Tools will not run from a network share unless that share is mapped to a drive letter, like “U:”. If the network location is in fact mapped, open an Explorer window from mapped drive letter, navigate to the Vis-Tools directory, then run server.bat again. If the location is not mapped, reinstall Vis-Tools to a local (non-network) location and try again.

Server Python file not in current directory

The server python file, server.py, is not present in the current directory. Please re-install Vis-Tools to a local (non-network) location and try again.

This can happen if you were to run the server.bat file with the current directory not set to the Vis-Tools directory. It is safest to open an Explorer window to the Vis-Tools directory and double-click server.bat to run the web server.

Windows SmartScreen dialog when double-clicking server.bat

Windows SmartScreen may attempt to prevent you from running server.bat, which is batch file.

If you see this dialog, click More Info, then click the Run Anyway button.

Socket.error starting web server

If you get an exception when you run the local web server that looks like this:

```
C:\Vis-Tools>server.bat
It appears that some other program is currently using port 8000. To use a different port, edit server.py and change the k_port constant to some other port, such as 8080. Or find and kill the program that is currently using port 8000.
Press any key to continue . . .
```

This means that some other program is currently bound to port 8000. You can:
1. Find and terminate the other program currently binding port 8000, or
2. Change the \texttt{k\_port = 8000} line in server.py to use some other, unused port.

**Windows SmartScreen dialog when starting web server**

If, when you run the web server, you get a big blue dialog box from Windows:

This indicates that your user account does not have sufficient permissions to bind port 8000 to the web server. This may occur in highly-managed environments. If you have the username and password for a local administrator account, you can enter it into the dialog box. Otherwise, contact your local IT staff.

**Running with an ad blocker**

Vis-Tools has no advertising and has no telemetry whatsoever. Nevertheless, some of the JavaScript libraries that Vis-Tools uses may be blocked by aggressive ad blocking browser plugins.

Vis-Tools attempts to detect when crucial components are blocked by an ad blocker and may display a page like the following:

Whitelist the Vis-Tools client page in your ad blocker, then click the Reload button and visualization should proceed.

**Geospatial web page never finishes loading**

You can get the web server running, but when it tries to load the demo \textit{Geospatial client}, it gets stuck with a spinning \textbf{Loading} indicator

Generally this means an error occurred while the page was loading. This can be caused by a variety of problems, but the most common one is that the web browser ran out of memory.

Try closing \textit{all} instances of the web browser on your computer, and stop the web server if it is running. Then re-run the web server by double-clicking server.bat. When server.bat runs the server and launches the browser, it uses special
Windows protected your computer

Windows SmartScreen prevented an unrecognized app from starting. Running this app requires administrator approval.

User name

Password

Domain:

Choose another option

Connect a smart card

Run anyway  Don't run
It looks like you're using an ad blocker. Vis-Tools doesn't have ads, but the ad blocker has blocked an essential library. Please turn off the ad blocker and click the Reload button below.

Reload
flags that allow the browser to use more memory, but this only works if there aren’t windows or tabs already open. Thus, letting server.bat open your first browser window may solve the spinning Loading problem.

1.4 Preprocessing simulation data

Before Vis-Tools can provide a visualization for a simulation’s data, a preprocessing step is required. The purpose of this preprocessing step is:

- To establish the data ranges for various simulation outputs.
- To distill overlarge node information into a form that uses less memory for consumption by the HTML visualization client.
- To provide a way to customize what is included in the visualization, bindings between visual parameters and simulation data, and styling.

The output of the preprocessing script is a visset file. The visset file contains a subset of the information about the nodes in your simulation along with links to other related files. The visset file drives the Vis-Tools visualization client HTML.

The section describes the preprocessing cycle involved in using Vis-Tools, the preprocessing classes available to use, and the structure of the preprocessing script.

1.4.1 Preprocessing cycle

It may be useful to think of using Vis-Tools as a cycle.

1. Run survey on your simulation data to generate a preprocessing script.
2. Run the preprocessing script (which will be in the Vis-Tools folder) to generate a visset file.
3. Browse to the URL printed by the preprocessing script (http://localhost:8000/...) and evaluate the visualization.
4. Modify the preprocessing script to suit your liking, then go back to step 2.

This page is mostly concerned with generating, running, and modifying the preprocessing script. The visualization step is covered in detail in Geospatial visualization.

Contents

- Create a default preprocessing script
  - Run survey on your own simulation data
- Run the preprocessing script
- Visualize the simulation
- Points versus shapes

Create a default preprocessing script

While it is possible to write a preprocessing script from scratch using the Python classes provided in Vis-Tools, typically you will let the survey program generate a default preprocessing script for you, and then customize that script to your liking.

survey looks at the available data in your simulation folder and emits a preprocessing script for you, which varies depending on what data your simulation contains. For example, if your simulation contains both files ReportHumanMigrationTracking.csv and ReportEventRecorder.csv, survey will emit code that creates an animation layer for infected human migration.

Here is the command-line usage for the survey program:

usage:
To create a preprocessing script for a simulation:
survey -i <sim_dir> -o <out_py_file> -t <Points|Shapes>

To update an existing visset.json with Asset Manager URLs:
survey -i <visset_path> -a <asset_map_path>

Survey generates Vis-Tools preprocessing scripts or updates existing visset.json files with COMPS Asset Manager URLs.

optional arguments:
-h, --help  show this help message and exit
-a ASSET_MAP_PATH, --assetmap ASSET_MAP_PATH
  Path to a VtAssetMap.json file.
-d DEMOGRAPHICS_PATH, --demographics DEMOGRAPHICS_PATH
  Full path to demographics file.
-f, --force  Write new preprocessing script even if one already exists with the same MD5.
-i SIM_DIR, --input SIM_DIR
  Directory containing simulation input and output files, or to a visset.json for update.
-o OUT_FILE, --output OUT_FILE
  Output script path.
The preprocessing scripts generated by `survey` are well-commented and contain sections of commented-out code that show best practices for various visualization scenarios, such as adding marker layers. Such best practices are described in *Components of a preprocessing script*.

**Note:** For `SIM_DIR` and `OUT_FILE` arguments, use forward slashes (/) in any paths you specify there, e.g.:

```
survey -i my_sim/run_1 -t Points
```

**Note:** The output of `survey` is a new Python program. So if you use the `-o` switch the `OUT_FILE` should have a “.py” extension, e.g.:

```
survey -i my_sim -t Points -o my_sim_prep.py
```

**Important:** The `survey` program is meant to be run from *within a simulation directory*. For example, a typical usage might be:

```
cd my\sim\sir
survey -t Shapes
```

**Run `survey` on your own simulation data**

Follow the steps below to run the `survey` program on your own simulation’s data.

**Note:** The only files strictly required for visualization are a demographics file with latitude and longitude locations for your nodes and either a1) a configuration file or b) at least one spatial report output/SpatialBinary_XXX.bin. Either of those latter two files provide the simulation timestep duration. Climate files are not needed by Vis-Tools.

1. Copy your simulation’s input and output data, including a demographics file and the output directory, into a new directory inside the Vis-Tools directory. This is required because Vis-Tools has its own local web server, which serves up only the Vis-Tools directory, and your simulation’s data needs to be among that.

   For this example, we assume the data is copied to a directory called `my_sim` within the Vis-Tools directory.

2. Open a command prompt window and `cd` to the simulation’s directory within the Vis-Tools directory, then run `survey`:
cd my_sim
survey -t Points

3. `survey` will emit a preprocessing Python script within the simulation directory called (by default) `preprocess_sim.py`. Because the `-t Points` argument was specified on the command line, that preprocessing script will represent your simulation’s nodes as two-dimensional points when run. For more on the difference between `points` and `shapes`, see Points versus shapes.

4. (Optional) Open and modify the preprocessing script to customize it to meet your needs.

Run the preprocessing script

Now that you created a default preprocessing script with `survey`, you can edit it to suit your needs or just run it as-is. Continuing with the “my_sim” example above, to run the preprocessing script, use the command:

```shell
cd my_sim
python Vis-Tools\...\preprocess_sim.py
```

(Where “…” above denotes the MD5 hash directory.) Running the script will generate a `visset` at `my_sim/Vis-Tools/.../visset.json`. The output of the preprocessing script shows the URL you will use to visualize the simulation. In this example that will be:

```
```

(Where again “…” above denotes the MD5 hash directory.) Make note of the URL from your simulation preprocessing output and continue to the next section.

Visualize the simulation

1. Ensure the local http server is running. See Start the web server.
2. In a browser, paste in the URL emitted by your preprocessing script.

Once the Vis-Tools Geospatial client page loads, you will see the yellow dots of your simulation’s nodes. For complete details on customizing the Geospatial client, see Geospatial visualization.

Points versus shapes

The Vis-Tools Geospatial client can represent the nodes of your simulation as either:

- Points, which are two-dimensional circles measured in pixels, or
- Shapes, which are three-dimensional rectangles measured in meters.

Those two representations have different visual parameters that you can bind to the outputs of your simulation. Depending on what you want to show, and how many simulation data channels you need to represent simultaneously, one of those may benefit you more than the other.

When you run `survey`, you can indicate which representation you want. If you don’t specify a `-t` argument to the `survey` program, it will by default create a preprocessing script using a points representation for nodes. However, it is not difficult to change the script to use shapes instead if you change your mind.

For more information on the `points` and `shapes` representation of nodes, see Geospatial visualization.
1.4.2 Components of a preprocessing script

Here we will look at a generated preprocessing script and break it down into parts.

The following script is the default script created by `survey` for the built-in demonstration data (`zambia_data`) that comes with Vis-Tools.

```python
# Generated by Survey.py

# This script is meant to be run from within the target simulation directory.

import time as tm
from os import path, getcwd
from vis_tools import VisSet, SpatialBinary, SpatialReports
```
# Constants
k_name = path.basename(getcwd())
k_sim_dir = r'.'
k_sim_output_dir = r'.\output'
k_products_dir = path.join(k_sim_dir, path.relpath(path.dirname(path.abspath(__file__)), getcwd()))
k_demographics_path_or_list = r'.\Bbondo_filled_calib_29.842464_prop.json'
k_sim_url_root = '/' + path.basename(getcwd()) + '/'
k_spatial_report_dir = k_sim_output_dir
k_config_path = r'.\config.json'
k_event_recorder_path = r'.\output\ReportEventRecorder.csv'
k_human_migration_path = r'.\output\ReportHumanMigrationTracking.csv'
k_malaria_filtered_path = r'.\output\ReportMalariaFiltered.json'

# Appearance-related constants
k_icon_duration_days = 2

# Main program
# This Vis-Tools preprocessing program contains various useful examples in the
# form of comments. Whenever possible the examples are made to be valid for the
# present simulation, but in some cases you may have to modify the examples to
# be valid in your context. Note that in this script you can override any option
# or default value you see in defaultvisset.json.

if __name__ == '__main__':
    script_start = tm.time()

    # Create a default VisSet
    vis_set = VisSet(k_name)

    # Set up the VisSet
    vis_set.set_target_client('Geospatial')
    vis_set.set_products_directory(k_products_dir)
    vis_set.set_config(k_config_path)
    vis_set.set_demographics(k_demographics_path_or_list)
    vis_set.set_node_vis_type('Points')

    # Remove the 'elsewhere' node from the node stats. This doesn't remove the
    # node, just leaves it out of min/max stats. Do this before adding the
    # spatial binaries, so as to also exclude that node from the spatial stats.
    # vis_set.exclude_node_from_stats(1001)

    # Inset chart
    vis_set.set_inset_chart(k_malaria_filtered_path)
    vis_set.options['insetCharts']['defaultChannelName'] =
    # 'Infectious Vectors'

    # Add extra NodeAttribute fields
    vis_set.add_node_field('InitialPopulation')
    vis_set.add_node_field('BirthRate')

    # Create a SpatialReports object that scans the simulation output directory
    # looking for SpatialReport_*.bin files. Later we'll add this to the VisSet
    # object.
    reports = SpatialReports(k_spatial_report_dir)
# Example of creating an infected vectors count spatial binary by combining SpatialReport_Adult_Vectors with SpatialReport_Infectious_Vectors. Once created, we manually add the new report to the reports object

```python
def combine(a, b):
    return int(a * b)

print('Synthesizing Infectious Vectors Count spatial binary...')
infected_vectors_count = SpatialBinary.combine(
    path.join(k_spatial_report_dir, 'SpatialReport_Adult_Vectors.bin'),
    path.join(k_spatial_report_dir, 'SpatialReport_Infectious_Vectors.bin'),
    'Infected Vectors Count',
    combine)
report_path = path.join(k_products_dir,
    'SpatialReport_Infectious_Vectors_Count.bin')
infected_vectors_count.write_binary(report_path)
reports.add(report_path)

# Add spatial reports (also establishes timestep count)
vis_set.add_spatial_reports(reports)

# Include only a subset of available spatial reports. If you have a lot of spatial reports, you may run out of memory on the client side. So here you can easily decide which channels are important and include just those channels. All available channels are shown below, so just delete the lines for channels you don't care about.
vis_set.include_spatial_channels([
    'SpatialReport_Adult_Vectors',
    'SpatialReport_Daily_Bites_Per_Human',
    'SpatialReport_Daily_EIR',
    'SpatialReport_Human_Infectious_Reservoir',
    'SpatialReport_Infectious_Vectors',
    'SpatialReport_Infectious_Vectors_Count',
    'SpatialReport_Mean_Parasitemia',
    'SpatialReport_New_Clinical_Cases',
    'SpatialReport_New_Diagnostic_Prevalence',
    'SpatialReport_New_Infections',
    'SpatialReport_Population',
    'SpatialReport_Prevalence'
])

# Infected human migration (do this after adding spatial reports)
vis_set.set_event_recorder(k_event_recorder_path)
vis_set.add_infected_human_migrations(k_human_migration_path)

# Add event marker layers
vis_set.add_marker_layer('Received_Treatment',
    marker=VisSet.k_marker_shape_triangle,
    color='green')
vis_set.add_marker_layer('NewInfectionEvent',
    marker=VisSet.k_marker_shape_triangle,
    color='blue')

# Other option overrides
vis_set.options['defaultBaseLayer'] = 'ESRI National Geographic'
vis_set.hide_layer('heatmapVis')
```

# Example bindings (NOTE: You may or may not have these sources available.)
# Imports and constants

```python
# Generated by Survey.py

# This script is meant to be run from within the target simulation directory.
import time as tm
from os import path, getcwd
from vis_tools import VisSet, SpatialBinary, SpatialReports

# Constants
k_name = path.basename(getcwd())
k_sim_dir = r'.'
k_sim_output_dir = r'.\output'
k_products_dir = path.join(k_sim_dir, path.relpath(path.dirname(path.abspath(__file__)), getcwd()))
k_demographics_path_or_list = r'.\Bbondo_filled_calib_29.842464_prop.json'
k_sim_url_root = '/' + path.basename(getcwd()) + '/'
k_spatial_report_dir = k_sim_output_dir
k_config_path = r'.\config.json'
k_event_recorder_path = r'.\output\ReportEventRecorder.csv'
k_human_migration_path = r'.\output\ReportHumanMigrationTracking.csv'
k_malaria_filtered_path = r'.\output\ReportMalariaFiltered.json'

# Appearance-related constants
k_icon_duration_days = 2
```

This section imports some built-in Python classes and a few of the Vis-Tools classes as well.

- **VisSet** is a wrapper class for the output `visset` file. Most of the preprocessing script revolves around populating and making method calls on a `VisSet` object.

## 1.4. Preprocessing simulation data
SpatialBinary is a wrapper class for an EMOD spatial binary report, which generally ends with the file extension .bin. (The file format for those files is documented in Spatial output report.

SpatialReports is a collection class that surveys and manages the list of spatial reports found (typically) in a simulation’s output directory.

Next the script defines some constants with file paths.

Finally, k_icon_duration_days controls how long marker icons show up in marker layers. This is described in more detail in Marker layers below.

Setup of the VisSet object

```python
if __name__ == '__main__':
    script_start = tm.time()

    # Create a default VisSet
    vis_set = VisSet(k_name)

    # Set up the VisSet
    vis_set.set_target_client('Geospatial')
    vis_set.set_products_directory(k_products_dir)
    vis_set.set_config(k_config_path)
    vis_set.set_demographics(k_demographics_path_or_list)
    vis_set.set_node_vis_type('Points')
```

This section begins the main program. First it creates a VisSet object. This object, at the end of the program, is emitted as a visset.json file.

Next the VisSet object is populated with some basic information.

- **set_target** indicates which visualization client the VisSet is targeting. Currently there is only one visualization target: Geospatial. Subsequent versions of Vis-Tools will have additional visualization clients.

- **set_products_directory** tells the VisSet the directory to which to write its products (e.g. the visset.json file, animation layer files, etc.) The products directory helps keep preprocessing outputs from polluting your simulation directory, and helps isolate them from other users who might also visualize your simulation with Vis-Tools.

- **set_config** tells the VisSet the path to the simulation configuration JSON file, which is typically named config.json.

- **set_demographics** tells the VisSet the file path to the simulation’s demographics file (or an array of filenames, if so specified in the config.json file). The VisSet will open and digest the node data here as well.

- **set_node_vis_type** tells the VisSet what node representation to use: two-dimensional points or three-dimensional shapes, as described in Points versus shapes.
**Drop a node from statistics**

```python
# Remove the 'elsewhere' node from the node stats. This doesn't remove the
# node, just leaves it out of min/max stats. Do this before adding the
# spatial binaries, so as to also exclude that node from the spatial stats.
# vis_set.exclude_node_from_stats(1001)
```

This code, which is commented out by default, shows how to drop a node from the min/max statistics. This is specifically for handling simulations that use an “elsewhere” node in migrations, as mentioned in the comments, to keep such a node from skewing population and other ranges. Multiple nodes can also be excluded with `vis_set.exclude_nodes_from_stats`.

**Inset chart**

```python
# Inset chart
vis_set.set_inset_chart(k_malaria_filtered_path)
# vis_set.options['insetCharts']['defaultChannelName'] =
# 'Infectious Vectors'
```

This section associates an inset chart file with the VisSet. The inset charts file typically contains aggregated information about the simulation. When an inset chart file is attached, the visualization will have an inset chart panel, which looks like this:

![Inset chart example](image)

The inset panel will include all the channels in the associated inset chart file.

It is common for EMOD simulations to emit a file called `InsetChart.json`, but for Vis-Tools, any file with the same format as an inset chart file will work. For example, some malaria simulations emit a `ReportMalariaFiltered.json` file, which has the same format as `InsetChart.json`. In our example preprocessing script, `survey` did not find an `InsetChart.json` file, but did find `ReportMalariaFiltered.json`, so it included that as the inset file.

To learn more about the format of inset chart files, see Inset chart output report.
Extra node attributes

```python
# Add extra NodeAttribute fields
vis_set.add_node_field('InitialPopulation')
vis_set.add_node_field('BirthRate')
```

This bit of code lets us take elements of the demographics file’s `NodeAttributes` and copy them into the `VisSet` abbreviated node data. By default Vis-Tools keeps only the following information about nodes:

- Latitude
- Longitude
- Altitude (if present)

You can call the `VisSet.add_node_field` method to take additional fields from the `NodeAttribute` section of the node’s demographic data. For example, to also retain the `BirthRate` `NodeAttribute`, you would add:

```python
vis_set.add_node_field('BirthRate')
```

Retained `NodeAttribute` fields aren’t limited to numeric fields. You can include text fields too, such as names, catchment areas, or regional identifiers that you may have added to your demographics file.

Creation of a new spatial binary

```python
# Create a SpatialReports object that scans the simulation output directory
# looking for SpatialReport_*\.bin files. Later we'll add this to the VisSet
# object.
reports = SpatialReports(k_spatial_report_dir)

# Example of creating an infected vectors count spatial binary by combining
# SpatialReport_Adult_Vectors with SpatialReport_Infectious_Vectors. Once
# created, we manually add the new report to the reports object
# def combine(a, b):
#     return int(a * b)
# print('Synthesizing Infectious Vectors Count spatial binary...')
# infected_vectors_count = SpatialBinary.combine(
#     path.join(k_spatial_report_dir, 'SpatialReport_Adult_Vectors.bin'),
#     path.join(k_spatial_report_dir, 'SpatialReport_Infectious_Vectors.bin'),
#     'Infectious Vectors Count',
#     combine)
# report_path = path.join(k_products_dir,
#     'SpatialReport_Infectious_Vectors_Count.bin')
# infected_vectors_count.write_binary(report_path)
# reports.add(report_path)
```

First Vis-Tools creates a `reports` variable that contains a collection of spatial reports (SpatialReport_*\.bin) found in the simulation’s output directory. This list is established so that we can, if so desired, create additional reports to add to the collection.

Vis-Tools is not just a visualization technology; it is also a toolset. This next section of code (which is commented out) shows how you can use Vis-Tools to create a new spatial report by combining two existing reports. This example multiplies Adult Vectors by Infections Vectors to obtain Infectious Vector Count, then writes that data to a new spatial report binary file.

**Note:** The simulation’s spatial reports reside in the simulation’s output directory. New spatial reports created using
combine are instead put in the products directory, to avoid polluting the simulation’s original data.

## Inclusion of spatial reports

```python
# Add spatial reports (also establishes timestep count)
vis_set.add_spatial_reports(reports)

# Include only a subset of available spatial reports. If you have a lot of
# spatial reports, you may run out of memory on the client side. So here you
# can easily decide which channels are important and include just those
# channels. All available channels are shown below, so just delete the lines
# for channels you don't care about.
vis_set.include_spatial_channels(
    [
        'SpatialReport_Adult_Vectors',
        'SpatialReport_Daily_Bites_Per_Human',
        'SpatialReport_Daily_EIR',
        'SpatialReport_Human_Infectious_Reservoir',
        'SpatialReport_Infectious_Vectors',
        'SpatialReport_Infectious_Vectors_Count',
        'SpatialReport_Mean_Parasitemia',
        'SpatialReport_New_Clinical_Cases',
        'SpatialReport_New_Diagnostic_Prevalence',
        'SpatialReport_New_Infections',
        'SpatialReport_Population',
        'SpatialReport_Prevalence'
    ])
```

This section starts by calling the `vis_set.add_spatial_reports` method to add the spatial reports (including any you may have synthesized) to the VisSet.

The `vis_set.include_spatial_channels` call has a list of spatial channels that were present when `survey` was run. This call is here to allow you to remove some channels that you don’t need, in case you run out of memory in the Geospatial client. You can simply comment out any reports to which you don’t need access.

## Infected human migration

```python
# Infected human migration (do this after adding spatial reports)
vis_set.set_event_recorder(k_event_recorder_path)
vis_set.add_infected_human_migrations(k_human_migration_path)
```

This section creates an Infected Human Migrations animation layer for the visualization. To create that animation, the `vis_set` needs both the `ReportEventRecorder.csv` and the `ReportHumanMigrationTracking.csv` report. It combines the information in those two sources to find migrations of only infected individuals and adds them to the animation layer.

**Note:** The Infected Human Migrations animation layer is written to a file called `InfectedMigrations.czml` in the products directory.

**Note:** If your simulation lacks either an `ReportEventRecorder.csv` or a `ReportHumanMigrationTracking.csv` file, `survey` won’t emit any code for preprocessing the Infected Human Migrations animation.
Marker layers

```python
# Add event marker layers
# vis_set.add_marker_layer('Received_Treatment',
#    marker=VisSet.k_marker_shape_triangle,
#    color='green')
# vis_set.add_marker_layer('NewInfectionEvent',
#    marker=VisSet.k_marker_shape_triangle,
#    color='blue')
```

A marker layer is a type of animation that creates markers near nodes when particular events occur in the ReportEventRecorder.csv.

To add a marker layer, you call the VisSet add_marker_layer method, passing the name of the event, as well as the marker shape and color. For more information about adding marker layers, see vis_tools.VisSet.

**Note:** Marker layers are generated as new .czml files that are written to the products directory.

Options and bindings

```python
# Other option overrides
# vis_set.options['defaultBaseLayer'] = 'ESRI National Geographic'
vis_set.hide_layer('heatmapVis')

# Example bindings (NOTE: You may or may not have these sources available.
# You will have to modify these examples to utilize your available sources.)
# vis_set.add_binding('nodeVis', 'pointSize',
#    'InitialPopulation', 'scale(3, 16)')
# vis_set.add_binding('nodeVis', 'pointColor',
#    'SpatialReport_Prevalence', 'sampleGradient()')
# vis_set.add_binding('heatmapVis', 'source',
#    'SpatialReport_Infectious_Vectors_Count', 'none()')
```

This section starts by calling the VisSet hide_layer method to make it so that when the visset is loaded by the Vis-Tools Geospatial client, the heatmap layer will be turned off (unchecked) by default.

This is an example of a customization you can make on the preprocessing side that sets the initial state of the visset when it is loaded into the client.
The code below that shows examples of adding bindings between visual parameters and simulation data channels. For more on bindings, see *Geospatial visualization*.

**Note:** There are many options and styles you can override in your preprocessing script. To see the defaults, and what options and styles are available, look at the defaultvisset.json file in the Vis-Tools directory.

For example, here is a small section of defaultvisset.json:

```json
"defaultOptions": {  
  "Geospatial": {  
    "defaultAltitudeM": 1,  
    "defaultBaseLayer": "Bing Maps Aerial",  
    "clockStepDurationSecs": 14400,  
    "clockInitialTimestep": 0,  
    "clockAutoRun": false,  
  }  
}
```

If you wanted to change the base map used in your visualization to a road-style map instead of a satellite map, you could add the following line to your preprocessing script:

```python
visset.options['defaultBaseLayer'] = "OpenStreetMap"
```

## Finish preprocessing

```python
# Consistency check  
vis_set.validate()  

# Emit VisSet  
vis_set.write_as('visset.json', k_sim_url_root)  

# Emit URL  
print('When using the standalone Vis-Tools server, use the following URL:')  
visset_path = path.join(k_products_dir, 'visset.json').replace(r'\\', '/')  
visset_path = visset_path.replace('./', k_sim_url_root)  
print('http://localhost:8000/vistools/geospatial.html?set=%s' % visset_path)  

# Done  
script_end = tm.time()  
print('Elapsed time %.2f seconds' % float(script_end - script_start))
```

The last part of the preprocessing script calls the VisSet `validate` method to check it for consistency, then writes the visset file to the products directory.

**Note:** You can have as many visset files as you want for a given simulation. It may make sense to do that if you have different aspects of your simulation that you wish to demonstrate, each requiring a specific visual presentation.

To do this, make copies of your preprocessing script with the various customizations you want, and modify each scripts’ `write_as` call to use a different file name. After you run all of them, you’ll have multiple visset files in your products directory, which you can point to using the URL parameter `set=` as shown above.
1.4.3 Preprocessing classes

This section contains the reference documentation for the classes that make up the Vis-Tools preprocessing framework. Use this reference documentation in conjunction with Components of a preprocessing script and the default preprocessing scripts generated for you by survey.

vis_tools

vis_tools package

Submodules

vis_tools.CSVReport module

CVSReport.py

This class is a simple Python wrapper for csv report files.

Usage:

```python
report = CSVReport(path.combine(my_dir, "ReportEventRecorder.csv"))
print report
```

```python
class vis_tools.CSVReport.CSVReport (file_path='', verbose=False)
Bases: object

Class to hold DTK CSV report data.

The class is constructed with the path to the report file in question. Thereafter the public data members source_file and rows may be used to directly access the resultant Python representation of the file.

Additionally, since CSVReport implements __len__ and __iter__, the report object can be treated like a list, e.g.:

```python
row = report[10]  # Obtain the 11th row
```

Public members: The following data members are publicly exposed.

- **source_file** (str): A copy of the file_path that was used to construct the CSVReport object.
- **rows** (array): Array of Python objects made from the input file lines.
- **header** (array): Array of field names read from top of CSV file.

```python
make_series (name, time_field, data_field)
```

Make a Highcharts-compatible series object from CSV rows.

Returns Object for use as a Highcharts data series.

Return type obj

Parameters

- **name** (str) – The name that is put into the output series structure.
- **time_field** (str) – The column name for the column representing time.
- **data_field** (str) – The column name for teh data (Y) value.

Raises Data access exceptions.
**read_partial** *(file_path, row_count)*

Read the first row_count rows off a CSV.

To do a partial read of a CSV report, create a CSVReport with the default constructor, then call read_partial
to read as many rows as desired. E.g.:

```python
report = CSVReport()
report.read_partial(my_csv_file_path, 100)
print report.rows[10]
```

**Returns** None.

**Parameters**

- **file_path** *(str)* – File path of CSV report.
- **row_count** – The number of rows to read.

**Raises** I/O, csv exceptions.

**missing_columns** *(column_list)*

Confirms that a given set of columns exists in a CSVReport.

This function can be used to verify the presence of a set of (presumably required) fields in a CSVReport.

Typical usage is:

```python
if (rpt.missing_columns(["Time", "Node_ID"]) is None)
    # All require columns present, so carry on
```

**Returns** List of columns from column_list that are not present, or None if all columns are
present.

**Parameters** column_list *(list)* – Columns to be tested for.

---

**vis_tools.CZMLWriter module**

CZMLWriter.py

This class simplifies the writing of Cesium CZML animation output files. It has methods for writing Vis-Tools-specific animations (migration and marker). Note that there are high-level methods on VisSet that implicitly create and use
CZMLWriter so in many cases you will not need to directly instantiate/call this class yourself.

**Usage:**

```python
writer = CZMLWriter()
writer.set_sim_duration(timestep_count)
writer.add_event_markers(my_event_recorder, my_demographics,
    "New_Infection", "Triangle", "red", "Top", 2)
writer.write(path.join(my_dir, "New_Infections.czml"))
```

**class** vis_tools.CZMLWriter.CZMLWriter *(verbose=False)*

**Bases:** object

Class for creating high-level visualizations in Cesium CZML format.

- **k_default_node_point_size** = 8
- **k_default_clock_multiplier** = 14400
- **k_default_migration_trail_duration** = 86400

---

1.4. Preprocessing simulation data
k_default_migration_duration_days = 2
k_default_migration_duration_seconds = 172800

set_sim_dates (sim_start_date, total_timestep_count)
Set the sim start date and total timesteps.
These are required because all CZML layers are time-synced to the Cesium clock. It is possible to use
set_sim_duration in lieu of this function.

Returns None
Parameters
  • sim_start_date (date) – Start date for timestep 0
  • total_timestep_count (int) – Number of timesteps in simulation data

set_sim_duration (total_timestep_count, timestep_bounds=None, sim_start_date=None, emit_clock=True, anim_step_secs=None)
Set the sim duration and time range.
This function allows you to set the start date and total timesteps, but also allows emitting the CZML over
a subrange of timesteps less than the number of timesteps in total_timestep_count. This is effective for
trimming migration animations that would otherwise be too big down to a workable subset.

Returns None
Parameters
  • total_timestep_count (int) – Total timesteps in sim
  • timestep_bounds – (list (start, end)): Timestep subrange for
  • animation –
  • sim_start_date (date) – Start date for timestep 0
  • emit_clock (bool) – Whether to emit a CZML Clock packet
  • anim_step_secs (int) – Clock multiplier for Clock packet

add_nodes (demographics, population_as_size=True, with_altitude=False, processor=None, namer=None)
Add demographics node data to the CZMLWriter object. DEPRECATED.
This (DEPRECATED) method allows you to add node representations to a CZML output file, where the
node point size is relative to the node’s InitialPopulation. Since the Vis-Tools client does this in a much
more flexible way, this method is deprecated, but may be educational for those looking to extend the
CZMLWriter class.

Returns None
Parameters
  • demographics (Demographics) –
  • population_as_size (bool) – True to scale point size by
  • fixed size otherwise (InitialPopulation, ) –
  • with_altitude – True to emit node coordinates including their
  • false to just emit them with default altitude (altitude, ) –
  • processor – None or a function that processes the node and and the
  • wrapper into a new CZML packet. (czml) –
• **namer** – None or a function that converts a node into a name string

**add_migrations** (*migrations, demographics, with_altitude=False*)

Adds migration animations to a CZML output.

This function, given migrations in a particular format and demographics, adds “comet” animations for migration events.

**Returns** Number of infected human migrations in animation layer

**Parameters**

• **migrations** (*obj*) – An object that describes migrations (see below)

• **demographics** (*Demographics*) – A Demographics object describing nodes

• **with_altitude** – True to respect the altitude in the node coordinates

migrations is a dictionary where the keys are <timestep> and the values are objects with keys <from_node_id>-<to_node_id> and the values are the number of migrations at that timestep from from_node_id to to_node_id. See MigrationHelpers.py for more details.

**add_vector_migrations** (*vector_migrations, demographics, migration_duration_timesteps, dot_color, dot_size_pixels, path_color, path_thickness_pixels, path_trail_time_factor*)

Adds vector cohort migration animations to a CZML output.

This function, given vector migrations in a particular format and a demographics file, adds “comet” animations for migration events.

This function expects the following fields in vector_migrations:

• Time (int): the timestep of the beginning of the migration event

• FromNodeID (int): the node ID from which the migration emanates

• ToNodeID (int): the node ID to which the migration completes

**Returns** Number of vector cohort migrations in animation layer

**Parameters**

• **vector_migrations** (*CSVReport*) – The ReportVectorMigrations.csv

• **report.**

• **demographics** (*Demographics*) – The Demographics object describing the nodes.

• **migration_duration_timesteps** (*int*) – The duration of the migration in timesteps.

• **dot_color** (*string*) – A CSS #rrggb color for the comet dot.

• **dot_size_pixels** (*int*) – Size in pixels of comet dot.

• **path_color** (*string*) – A CSS #rrggb color for the comet tail.

• **path_thickness_pixels** (*float*) – Thickness in pixels of comet tail.

• **path_trail_time_factor** (*float*) – Length of trail as a multiple of migration_duration. E.g. if this is 1.0, the trail length will be...
• the full distance from source node to the destination node.
  \[d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}\]

• the trail length will be half the distance between the
  nodes. \((0.5, d)\) –

\[\text{add_simplified_vector_migrations}(\text{vector_migrations}, \text{demographics}, \text{migration_duration_timesteps}, \text{arrow_color}, \text{arrow_thickness_pixels})\]

Adds vector cohort migration animations to a CZML output.

This function, given vector migrations in a particular format and a demographics file, adds “comet” animations for migration events.

This function expects the following fields in vector_migrations:

• Time (int): the timestep of the beginning of the migration event
• FromNodeID (int): the node ID from which the migration emanates
• ToNodeID (int): the node ID to which the migration completes

Returns: Number of vector cohort migrations in animation layer

Parameters

• \text{vector_migrations}(\text{CSVReport}) – The ReportVectorMigrations.csv
• \text{report.} –
• \text{demographics}(\text{Demographics}) – The Demographics object describing the
  nodes. –
• \text{migration_duration_timesteps}(\text{int}) – The duration of the migration
• \text{in_timesteps.}(\text{animations}) –
• \text{arrow_color}(\text{string}) – A CSS #rrggb color for the migration arrow.
• \text{arrow_thickness_pixels}(\text{float}) – Thickness in pixels of comet tail.

\[\text{add_event_markers}(\text{event_recorder}, \text{demographics}, \text{event_name}, \text{marker}, \text{color}, \text{placement}, \text{duration_days}, \text{with_altitude=False})\]

Adds event marker animation to a CZML output.

This function looks for a particular event in an event recorder report and emits an animation that puts a marker somewhere around the node and leaves it there for a specified period after the event.

Returns: Number of event markers in animation layer

Parameters

• \text{event_recorder}(\text{CSVReport}) – Event recorder report object
• \text{demographics}(\text{Demographics}) – Demographics object with node data
• \text{event_name}(\text{str}) – Name of event to mark in the animation
• \text{marker}(\text{str}) – Name of a marker icon (“Triangle” or “Cross”)
• \text{color}(\text{str}) – Any HTML or SVG named color (e.g. “red”) or a CSS
• \text{string}(\text{color}) –
• \text{placement}(\text{str}) – Placement about the node point (“Top”, “Left”).
• or "Right"("Bottom") –
• duration_days(int) – Number of days for which the symbol should
  about the node after it fires.(remain) –
• with_altitude(bool) – True to set the symbol elevations to match
  node altitude, or default altitude otherwise.(the) –

add_weighted_network(demographics, network, gradient_spec, opacity_func)
Adds a weighted network visualization layer to a CZML output.
This method emits a CZML animation that provides a visual representation of a weighted network between nodes.

   Returns Number of network segments added

   Parameters
   • demographics(Demographics) – Demographics object for nodes.
   • network(array) – array of objects:

   ```
   {
     from: <from-node-id>,
     to: <to-node-id>,
     weight: <float-weight>
   }
   ```

   • gradient_spec(str) – gradient spec for a gradient with which to
     the network lines.(color) –
   • opacity_func(function) – function(weight, norm_weight) that returns
     desired opacity in range [0,1](the) –

write_czml(file_path)
Write the CZML animation file.

   Returns None

   Parameters file_path(str) – The file path to which to write the CZML animation.

   Raises I/O exceptions. –

vis_tools.Color module

Color.py
This file contains a simple RGB color class that uses normalized channel values in the range of 0-1.

Classes:
• Color - a simple RGB color object.

Usage:

```python
color = Color(1.0, 0.5, 0.0)  # Equivalent to #ff7f00 in CSS
```

```python
class vis_tools.Color.Color(r=0, g=0, b=0)
   Bases: object

   Class that encapsulates an RGB color.
```
Public members:  
- r (int): Red channel value in range(0, 255)  
- g (int): Green channel value in range(0, 255)  
- b (int): Blue channel value in range(0, 255)

`to_rgba_array_str()`  
Returns the color as a string representation of an RGBA array.  
This is used by CZMLWriter and other classes that need to emit colors into Javascript formats.

Returns “[r, g, b, 255]” where r, g, and b are the channel values.  
Return type  
str

`to_rgba_array()`  
Returns the color as an RGBA array.

Returns [r, g, b, 255] where r, g, and b are the channel values.  
Return type  
array

`static from_html_hash(html_color)`  
Returns an RGB from an HTML/CSS #rrggbb-format string.

Returns an RGB initialized with the colors from html_color  
Return type  
obj

Parameters  
- html_color – A string in the format “#rrggbb” where rr, gg, and bb are the channel values in hex.  

`static lerp(a, b, factor0to1)`  
Returns an interpolated color.

This function calculates a color that is between colors a and b using the factor0to1 argument to calculate the mix. If factor0to1 is zero, the resulting color is a. If factor0to1 is 1, the resulting color is b. For values of factor0to1 between 0 and 1, the resulting color is a linearly interpolated mix of colors a and b.

Returns An RGB initialized with the interpolated color.  
Return type  
obj

Parameters  
- a (obj) – An RGB representing the start color.  
- b (obj) – An RGB representing the end color.  
- factor0to1 (float) – A value in range(0,1) specifying the desired mix

  colors a and b. (between) –
**vis_tools.Config module**

Config.py

This class is a simple Python wrapper for the config.json file used to configure the DTK.

**Usage:**
```python
config = Config(path.join(my_dir, "config.json"))
print config
```

```python
class vis_tools.Config.Config(file_path='', verbose=False)

    Bases: object

    Class to hold DTK config.json data.

    The class is constructed with the path to the config.json file. Thereafter the public data members source_file, timestep_count, and parameters may be used to directly access the resultant Python object. There are also accessor methods for the most commonly needed parameters.

**Public members:** The following data members are publicly exposed.

    source_file (str): A copy of the file_path used to construct the Config object.

    timestep_count (int): The Simulation_Duration parameter.

    parameters (obj): The entire config.json as a Python object.

**get_demographics_filenames()**
Return the value of Parameters > Demographics_Filenames.

    **Returns** List of demographics filenames.

    **Return type** list(str)

    **Parameters** None. –

**get_first_demographics_filename()**
Returns the first (or only) demographics file from the config.

This function understands the presence of Demographics_Filename with a single path, Demographics_Filename with multiple paths delimited by ‘;’, and Demographics_Filnames as an array of paths.

    **Returns**

    Demographics file name from Demographics_Filenames or Demographics_Filename, or None.

    **Return type** str

    **Parameters** None. –

**get_demographics_filename_or_list()**
Returns a string or list of demographics filenames, or None.

This function understands the presence of Demographics_Filename with a single path, Demographics_Filname with multiple paths delimited by ‘;’, and Demographics_Filnames as an array of paths.

    **Returns** if a single demographics file is specified in the config, returns it as a string. If an array is specified, returns an array of filenames. Otherwise returns None.

    **Return type** str|list

    **Parameters** None. –

**get_timestep_count()**
Return the value of Parameters > Simulation_Duration.

    **Returns** Timestep count.

1.4. Preprocessing simulation data
Vis-Tools

Return type  int
Parameters  None.

vis_tools.Demographics module

Demographics.py
This class is a Python wrapper for the DTK’s demographics JSON files.

Usage::  demographics = Demographics(path.join(my_dir, “Demographics.json”)) print demographics
class vis_tools.Demographics.Demographics (file_path_or_list=None, verbose=False)
    Bases: object

Class to hold DTK demographics data.
The class is constructed with the path to the demographics JSON file. Thereafter the public data members
described below may be used to directly access (and modify, if desired) the resultant Python object. There are
also accessor methods for commonly needed or calculated values.

Additionally, since Demographics implements __len__, __iter__, __getitem__, and __contains__, the object
can be be treated like a dictionary, e.g.:

    node = demographics[my_node_id]  # Obtain a particular node

Public members:  source_file (str): The path to the demographics JSON file. If the object is initialized from a
list of overlays, source_file is the path of the base demographics file.
    latitude_min (float): Minimum latitude for all nodes
    latitude_max (float): Maximum latitude for all nodes
    longitude_min (float): Minimum longitude for all nodes
    longitude_max (float): Maximum longitude for all nodes
    latitude_delta_min (float): smallest non-zero difference between node latitude coordinates. Used to calcu-
late the size of a bitmap that has sufficient resolution to discern between nodes in the vertical axis.
    longitude_delta_min (float): smallest non-zero difference between node longitude coordinates. Used to
calculate the size of a bitmap that has sufficient resolution to discern between nodes in the horizontal axis.
    population_min (float): Minimum InitialPopulation for all nodes
    population_max (float): Maximum InitialPopulation for all nodes
    population_max (float): Maximum InitialPopulation for all nodes
    nodes_by_id (obj): dictionary keyed on nodeId containing node objects
    adjusted_pop_max (float): population_max but with the highest value removed. (DEPRECATED)
    Defaults (obj): The raw Defaults portion of the demographics file
    MetaData (obj): The raw Metadata portion of the demographics file
    NodeProperties (array): The raw NodeProperties portion of the demographics file
    Nodes (array): The raw Nodes portion of the demographics file

    k_default_node_altitude = 1
    bounding_box (for_json=True)
    Returns a geospatial bounding box for the nodes.
Returns bounding box in either Python or Javascript style.

**Return type** obj

**Parameters**

- **for_json** *(bool)* – If true, emits Javascript naming conventions, or
- **conventions otherwise. (Python)* –

**population_range** *(for_json=True)*

Returns the range of InitialPopulation for all nodes.

**Returns** range of population in either Python or Javascript style.

**Return type** obj

**Parameters**

- **for_json** *(bool)* – If true, emits Javascript naming conventions, or
- **conventions otherwise. (Python)* –

**minimum_deltas** *(for_json=True)*

Returns the minimum non-zero deltas for latitude and longitude.

**Returns** minimum non-zero lat/long deltas in either Python or Javascript style.

**Return type** obj

**Parameters**

- **for_json** *(bool)* – If true, emits Javascript naming conventions, or
- **conventions otherwise. (Python)* –

**calc_adjusted_pop_max** ()

Removes the largest Initial Population value. (DEPRECATED)

This function can be used to obtain the next-to-highest InitialPopulation value. This would be used for removing the “elsewhere” node in some simulations. A better way is to use the method Vis-Set.exclude_nodes_from_stats(), which allows multiple nodes and ensures the updated min/max are in the VisSet where they are needed.

**Returns** The adjusted maximum InitialPopulation.

**Return type** int

**Parameters** None. –

**incorporate_altitudes** *(alt_csv)*

Retroactively apply altitudes to the nodes.

This function updates the nodes’ altitude attribute using data from a CSV file. The file may have other fields, but it needs one column called “NodeID” and one column called “Altitude”.

**Returns** number of nodes updated.

**Return type** int

**Parameters** alt_csv *(str)* – The file path of the CSV file with altitudes.

**Raises** I/O and CSV exceptions. –

**emit_nodes** ()

Returns an array of node objects suitable for use in the VisSet.
You can use this function to get a trimmed-down nodes data structure from the Demographics object. Normally the VisSet will do this encapsulation for you, but this method is present if needed.

Note that InitialPopulation, if present, is initial-capped. This is because users can choose to pull in extra data from NodeAttributes into the nodes that are cached in the VisSet, and use those for static visual mappings on the client side. Any fields outside of [nodeId|latitude|longitude|altitude] are exposed by the client as sources. This way the user can use the initial-capped fieldnames they're used to.

**Returns** An array of objects representing the nodes.

**Return type** array

**Parameters** None

```python
make_index(node_attribute_field)
```

Returns an index from node_attribute_field to node id(s).

This function creates an index on any field within NodeAttributes that maps that allows you to look up the nodes that have that node attribute. For example, if you nodes have a FacilityName field in NodeAttributes, you could do:

```python
index = demo.make_index("FacilityName")
```

then later, to look up the node ids for a given FacilityName:

```python
node_ids = index["3628"]
for id in node_ids
    print id
```

**Returns** index

**Return type** dict<value, array<node_id>>

**Parameters** node_attribute_field (**str** – field name within NodeAttributes.

---

**vis_tools.Gradient module**

Gradient.py

This file contains a class for doing Python-side color gradients. The Gradient class is a gradient with an arbitrary number of color stops. Since Gradient is derived from the Vis-Tools gradient.js, the same style text representations of gradients can be used.

**Classes:**

- Gradient - a sampleable color gradient.

**Usage:**

```python
gradient = Gradient("green@0,orange@0.33,yellow@0.66,red@1")
color = gradient.sample(0.5)
```

```python
class vis_tools.Gradient.Gradient (spec="")
    Bases: object
    
    Class for sampleable color gradients.
    
    This class lets you create a color gradient with an arbitrary number of color stops on a normalized range from 0 to 1. You then sample the gradient with a normalized value from 0 to 1 to get a color out of the gradient.
```
This class is based on the Vis-Tools gradient.js. However it does not support the “.r” or “.q<steps>” suffixes supported by gradient.js.

**The spec format is:** `<color>@0,[<color>@<loc>,. . . ]<color>@1

Where:

• color - Color objects representing the stop colors
• loc - Ordered values in range(0, 1) representing the normalized locations of the gradient stops.

Raises `ValueError` – if spec is invalid.

**sample** *(loc0to1)*

Sample the gradient to get a color at a particular location.

**Returns** A Color object for the color at the sample point.

**Return type** obj

**Parameters**

• loc0to1 *(float)* – A normalized value in the range(0, 1) at which to sample the gradient color.

**vis_tools.MigrationHelpers module**

*MigrationHelpers.py*

This class contains static methods that are useful in the composition of infected migration animations. Namely *collate_infected_human_migrations()* , which trawls through a ReportHumanMigrationTracking report and combines that with a ReportEventRecorder report to find and create a data structure of the migrations of only infected individuals.

While there is currently only one method in this class, as other forms of migration culling are developed more methods will be added here as needed.

**class vis_tools.MigrationHelpers.MigrationHelpers**

Bases: object

Helper functions for processing migrations.

**static collate_infected_human_migrations**(rpt_human_migs, rpt_evt_rec)

Glean infected human migrations from two source files.

This function combines a ReportHumanMigrationTracking.csv and a ReportEventRecorder.csv to generate a set of infected human migrations.

**Returns**

obj:

```python
{
    infected_migrations: { ... },
    aggregate_migrations_max: n
}
```

where:

• infected_migrations in the result is an dictionary where the keys are <timestep> and the values are objects with keys <from_node_id>-<to_node_id> and the values are the number of migrations at that timestep from from_node_id to to_node_id.

1.4. Preprocessing simulation data 53
aggregate_migrations_max in the result is the most simultaneous migrations between any two nodes. This could be used for visualization, for example using a thicker line for more migrations.

Parameters


`vis_tools.NamedColors` module

NamedColors.py

This file contains a class that provides all the SVG color names as Color objects.

Classes:

- `NamedColors` - the SVG named colors as Color objects.

Usage:

```python
color = NamedColors.coral
```

```python
class vis_tools.NamedColors.NamedColors
    Bases: object

    Named SVG colors as Color objects.

    This class is a reference data class that contains Color objects for all the SVG named colors. These can be used in Gradient stops for better readability.

    alicebule = <vis_tools.Color.Color object>
    antiquewhite = <vis_tools.Color.Color object>
    aqua = <vis_tools.Color.Color object>
    aquamarine = <vis_tools.Color.Color object>
    azure = <vis_tools.Color.Color object>
    beige = <vis_tools.Color.Color object>
    bisque = <vis_tools.Color.Color object>
    black = <vis_tools.Color.Color object>
    blanchedalmond = <vis_tools.Color.Color object>
    blue = <vis_tools.Color.Color object>
    blueviolet = <vis_tools.Color.Color object>
    brown = <vis_tools.Color.Color object>
    burlywood = <vis_tools.Color.Color object>
    cadetblue = <vis_tools.Color.Color object>
    chartreuse = <vis_tools.Color.Color object>
    chocolate = <vis_tools.Color.Color object>
    coral = <vis_tools.Color.Color object>
    cornflowerblue = <vis_tools.Color.Color object>
```
cornsilk = <vis_tools.Color.Color object>
crimson = <vis_tools.Color.Color object>
cyan = <vis_tools.Color.Color object>
darkblue = <vis_tools.Color.Color object>
darkcyan = <vis_tools.Color.Color object>
darkgoldenrod = <vis_tools.Color.Color object>
darkgray = <vis_tools.Color.Color object>
darkgrey = <vis_tools.Color.Color object>
darkgreen = <vis_tools.Color.Color object>
darkkhaki = <vis_tools.Color.Color object>
darkmagenta = <vis_tools.Color.Color object>
darkolivegreen = <vis_tools.Color.Color object>
darkorange = <vis_tools.Color.Color object>
darkorchid = <vis_tools.Color.Color object>
darkred = <vis_tools.Color.Color object>
darksalmon = <vis_tools.Color.Color object>
darkseagreen = <vis_tools.Color.Color object>
darkslateblue = <vis_tools.Color.Color object>
darkslategrey = <vis_tools.Color.Color object>
darkturquoise = <vis_tools.Color.Color object>
darkviolet = <vis_tools.Color.Color object>
deeppink = <vis_tools.Color.Color object>
deepskyblue = <vis_tools.Color.Color object>
dimgray = <vis_tools.Color.Color object>
dimgrey = <vis_tools.Color.Color object>
dodgerblue = <vis_tools.Color.Color object>
darkred = <vis_tools.Color.Color object>
floralwhite = <vis_tools.Color.Color object>
forestgreen = <vis_tools.Color.Color object>
fuchsia = <vis_tools.Color.Color object>
gainsboro = <vis_tools.Color.Color object>
ghostwhite = <vis_tools.Color.Color object>
gold = <vis_tools.Color.Color object>
goldenrod = <vis_tools.Color.Color object>
gray = <vis_tools.Color.Color object>
grey = <vis_tools.Color.Color object>
green = <vis_tools.Color.Color object>
greenyellow = <vis_tools.Color.Color object>
honeydew = <vis_tools.Color.Color object>
hotpink = <vis_tools.Color.Color object>
indianred = <vis_tools.Color.Color object>
indigo = <vis_tools.Color.Color object>
ivory = <vis_tools.Color.Color object>
khaki = <vis_tools.Color.Color object>
lavender = <vis_tools.Color.Color object>
lavenderblush = <vis_tools.Color.Color object>
lawngreen = <vis_tools.Color.Color object>
lemonchiffon = <vis_tools.Color.Color object>
lightblue = <vis_tools.Color.Color object>
lightcoral = <vis_tools.Color.Color object>
lightcyan = <vis_tools.Color.Color object>
lightgoldenrodyellow = <vis_tools.Color.Color object>
lightgray = <vis_tools.Color.Color object>
lightgrey = <vis_tools.Color.Color object>
lightgreen = <vis_tools.Color.Color object>
lightpink = <vis_tools.Color.Color object>
lightsalmon = <vis_tools.Color.Color object>
lightsagegreen = <vis_tools.Color.Color object>
lightskyblue = <vis_tools.Color.Color object>
lightslategray = <vis_tools.Color.Color object>
lightslategrey = <vis_tools.Color.Color object>
lightsteelblue = <vis_tools.Color.Color object>
lightyellow = <vis_tools.Color.Color object>
lime = <vis_tools.Color.Color object>
limegreen = <vis_tools.Color.Color object>
linen = <vis_tools.Color.Color object>
magenta = <vis_tools.Color.Color object>
maroon = <vis_tools.Color.Color object>
mediumaquamarine = <vis_tools.Color.Color object>
mediumblue = <vis_tools.Color.Color object>
mediumorchid = <vis_tools.Color.Color object>
mediumpurple = <vis_tools.Color.Color object>
mediumseagreen = <vis_tools.Color.Color object>
mediumslateblue = <vis_tools.Color.Color object>
mediumspringgreen = <vis_tools.Color.Color object>
mediumturquoise = <vis_tools.Color.Color object>
mediumvioletred = <vis_tools.Color.Color object>
midnightblue = <vis_tools.Color.Color object>
mintcream = <vis_tools.Color.Color object>
mistyrose = <vis_tools.Color.Color object>
moccasin = <vis_tools.Color.Color object>
navajowhite = <vis_tools.Color.Color object>
navy = <vis_tools.Color.Color object>
oldlace = <vis_tools.Color.Color object>
olive = <vis_tools.Color.Color object>
olivedrab = <vis_tools.Color.Color object>
orange = <vis_tools.Color.Color object>
orangered = <vis_tools.Color.Color object>
orchid = <vis_tools.Color.Color object>
palegoldenrod = <vis_tools.Color.Color object>
palegreen = <vis_tools.Color.Color object>
paleturquoise = <vis_tools.Color.Color object>
palevioletred = <vis_tools.Color.Color object>
papayawhip = <vis_tools.Color.Color object>
peachpuff = <vis_tools.Color.Color object>
peru = <vis_tools.Color.Color object>
pink = <vis_tools.Color.Color object>
plum = <vis_tools.Color.Color object>
powderblue = <vis_tools.Color.Color object>
purple = <vis_tools.Color.Color object>
red = <vis_tools.Color.Color object>
rosybrown = <vis_tools.Color.Color object>
royalblue = <vis_tools.Color.Color object>
saddlebrown = <vis_tools.Color.Color object>
_salmon = <vis_tools.Color.Color object>
seagreen = <vis_tools.Color.Color object>

1.4. Preprocessing simulation data
seashell = <vis_tools.Color.Color object>
sienna = <vis_tools.Color.Color object>
silver = <vis_tools.Color.Color object>
skyblue = <vis_tools.Color.Color object>
slateblue = <vis_tools.Color.Color object>
slategray = <vis_tools.Color.Color object>
slategrey = <vis_tools.Color.Color object>
snow = <vis_tools.Color.Color object>
springgreen = <vis_tools.Color.Color object>
steelblue = <vis_tools.Color.Color object>
tan = <vis_tools.Color.Color object>
teal = <vis_tools.Color.Color object>
thistle = <vis_tools.Color.Color object>
tomato = <vis_tools.Color.Color object>
turquoise = <vis_tools.Color.Color object>
violet = <vis_tools.Color.Color object>
wheat = <vis_tools.Color.Color object>
white = <vis_tools.Color.Color object>
whitesmoke = <vis_tools.Color.Color object>
yellow = <vis_tools.Color.Color object>
yellowgreen = <vis_tools.Color.Color object>

vis_tools.SpatialBinary module

SpatialBinary.py

This file contains:

- SpatialBinary - a wrapper for SpatialReport DTK output files.

SpatialBinary is a Python wrapper for DTK SpatialReport_* files. It can both read and write them, and can combine them using combiner functions to make new SpatialReports.

Usage:

```python
spatial_binary = SpatialBinary(path.combine(my_dir,
                                       "SpatialReport_Prevalence.bin"))
print spatial_binary
```

class vis_tools.SpatialBinary.SpatialBinary (file_path="", drop_zeros=False, excluded_node_ids=None, verbose=False)

Bases: object

Class to hold DTK spatial binary report data.

The class is constructed with the path to the report file in question. Thereafter the public data members described below may be used to directly access (or change) the data.
Additionally, since SpatialBinary implements __len__, __iter__, and __getitem__, the object can be treated like an array on timestep, e.g.:

```python
timestep_rec = spatial_binary[timestep]  # Obtain one timestep
```

The resulting timestep_rec is a dictionary<node_id, channel_value>.

**Public members:**
- `drop_zeros (bool)`: True: drop zero values from in-memory representation.
- `source_file (str)`: A copy of the file_path that was used to construct the SpatialBinary object.
- `channel_name (str)`: The channel name, pulled from source_file.
- `node_count (int)`: The number of nodes in the SpatialBinary’s node table.
- `value_min (float)`: The minimum value for all nodes * timesteps.
- `value_max (float)`: The maximum value for all nodes * timesteps.
- `timesteps (array)`: Array of dictionaries containing the spatial report’s data.

**value_range** *(for_json=False)*

Returns an object with the value range of the data.

---

**Returns**
An object with the min/max values of the data, with either Python or Javascript naming conventions.

**Return type** `obj`

**Parameters**

- `for_json (bool)` – If true, emit an object using Javascript naming conventions.
- `otherwise use Python naming conventions (conventions,)`

**write_binary** *(bin_file_path)*

Write the SpatialBinary to a given file path.

This function write out the spatial data in the object to a SpatialReport-format binary file. Typically this is used when the caller has modified the data in a SpatialBinary object or used combine() to create a new one. Note that if zeros were dropped, write_binary will throw an exception.

---

**Returns** None.

**Parameters**

- `bin_file_path (str)` – The file path to which to write.

**Raises** `ValueError` – if the SpatialBinary used drop_zeros on construction.

---

**To do:**

- Make it work even for sparse spatial binaries. The zero values are implied in by missing keys in the timestep records, so no actual data is missing.

**clone**

Returns a copy of this SpatialBinary in a new SpatialBinary object.

---

**Returns** A new SpatialBinary object populated from self.

**Return type** `obj`

**Parameters** None. –

**print**

Prints the entire contents of the spatial binary. Can be lengthy.

---

**Returns** None.
Parameters None.

**static combine** *(bin_file_path_1, bin_file_path_2, channel_name, combine_func)*

Combine two SpatialBinary objects into a new SpatialBinary object.

This function takes two SpatialBinary objects (of the same exact dimensions in both timesteps and nodes) and combines them through a "combine function" to make an entirely new in-memory SpatialBinary. That resulting SpatialBinary would then typically be written out using write_binary().

There are four simple arithmetic static combine functions built into SpatialBinary, but the user may pass in any valid combine function that has a compatible signature. (See Usage below.) For operations that are not commutative such as division, let it be known that argument value1 in the combine function comes from bin_file_path1, and value2 comes from bin_file_path2.

Beware: temporarily has all three SpatialBinaries in memory.

Usage:

```python
def rounded_multiply_combiner(value1, value2):
    return round(value1 * value2)
inf_vec_count = SpatialBinary.combine(
    "output/SpatialReport_Adult_Vectors",
    "output/SpatialReport_Infected_Vectors",
    "Infected Vector Count", rounded_multiply_combiner)
inf_vec_count.write_binary("SpatialReport_Infected_Vector_Count")
```

**Returns** A new SpatialBinary object combining sources 1 and 2

**Return type** obj

**Parameters**

- **bin_file_path_1** *(str)* – File path of first spatial binary file.
- **bin_file_path_2** *(str)* – File path of second spatial binary file.
- **channel_name** *(str)* – Channel name to assign to the result binary.
- **combine_func** *(function)* – A function that combines the values from the two spatial binary inputs, one at a time. The signature of the combine_func is:

  ```python
  combine_func(value1, value2)
  return value1 + value2  # for example
  ```

**Raises**

- **ValueError** – if SpatialBinary objects don’t have same dimensions or
- **nodes** –

**static multiply_combiner** *(value1, value2)*

Combiner function that multiplies channel values.

**Returns** new value.

**Return type** float

**Parameters**

- **value1** *(float)* – Value from input file 1.
- **value2** *(float)* – Value from input file 2.
**static add_combiner**(value1, value2)
Combiner function that adds channel values.

- **Returns**: new value.
- **Return type**: float
- **Parameters**
  - value1 (float) – Value from input file 1.
  - value2 (float) – Value from input file 2.

**static subtract_combiner**(value1, value2)
Combiner function that subtracts channel values.

- **Returns**: new value.
- **Return type**: float
- **Parameters**
  - value1 (float) – Value from input file 1.
  - value2 (float) – Value from input file 2.

**static divide_combiner**(value1, value2)
Combiner function that divides channel values.

- **Returns**: new value.
- **Return type**: float
- **Parameters**
  - value1 (float) – Value from input file 1.
  - value2 (float) – Value from input file 2.

---

### vis_tools.SpatialBinaryHeader module

SpatialBinaryHeader.py

This file contains:
- SpatialBinaryHeader - a class for extracting just the header from SpatialReport files.

SpatialBinaryHeader is a class that allows access to the dimensions of a spatial report without actually reading the entire file.

```python
class vis_tools.SpatialBinaryHeader.SpatialBinaryHeader (file_path='', verbose=False)
Bases: object

Class to read header off a spatial binary without reading the contents.
```

Usage:

```python
bin_header = SpatialBinaryHeader("output/SpatialReport_Prevalence")
print bin_header
```
**vis_tools.SpatialReports module**

SpatialReports.py

This class is a collection class that keeps track of all the SpatialReport_* files in a given directory. It is used by VisSet.

Usage:

```python
reports = SpatialReports("output/")
print reports
```

```python
class vis_tools.SpatialReports.SpatialReports (spatial_dir='', verbose=False)
Bases: object
Collection class for a set of spatial reports.
The class is constructed with the directory in which SpatialReport_* files reside. Thereafter the public data members described below, and the various accessor functions, can be used to get access to the list.

Additionally, since SpatialReports implements __len__, __iter__, and __getitem__, the SpatialReports object can be treated as a list, e.g.:

```python
spatial_reports = SpatialReports("/output")
path = spatial_reports[0]
```

Public members:

- **source_dir (str):** The source directory used to construct the SpatialReports object.
- **paths (array):** Array of strings with paths to the SpatialReport_* files.
- **properties (dict):** Dictionary<friendlyName, dict<key,value>> of additional properties to associate with each SpatialBinary file path. Currently these are unused, but will be used in an upcoming version.

```python
k_spatial_filename_glob = 'SpatialReport*_*.bin'
k_spatial_leaf_re = '^SpatialReport[^_]*_(.*)'
```

```python
get(index)
Returns detailed information on spatial binary by index.

Returns
An object containing the name, friendly name, and path (url) to the spatial binary at the provided index.

Return type
obj

Parameters
index (int) – Index, 0-based, into path list.
```

```python
add(report_path)
Manually add a spatial binary to the list.

This method can be used to manually add a spatial binary to the list even if it is not in the source_dir where spatial_binaries normally live. This is useful for adding a programmatically-generated spatial binary to the list. Such generated spatial binaries are generally located in the preprocessing script’s products directory.

Example:

```python
reports = SpatialReports("output/")
reports.add(path.join(k_products_dir, 'SpatialReport_Infectious_Vector_Count'))
```

Returns None.
Parameters

- `report_path (str)` – The path to the spatial binary. Note that this
- `need not be within source_dir`.

`emit_object()`
Emits an object used by VisSet to represent spatial binary channels.

**Result**: object: A Dictionary<friendly_name, file_path> for the spatial binaries found in the target directory.

**Parameters** None.

`emit_object_json()`
Emits the JSON for the object from emit_object.

**Result**: str: JSON representation of the emit_object() result.

**Parameters** None.

`remove (friendly_name)`
Removes a spatial binary from the list by friendly name.

**Returns** None.

**Parameters**

- `friendly_name (str)` – The friendly name of the spatial binary path
- `remove`. E.g., for SpatialReport_Adult_Vectors, the friendly
  to “Adult Vectors”.

**vis_tools.Survey module**

Survey.py

Survey is a Python program that surveys a sim folder and generates a default Vis-Tools preprocessing program for that sim. Run the output .py file and it will generate a working (if rather plain) visset.json for the sim as a starting point for customizations.

Note that both Survey and its generated preprocessing script is meant to be run from the simulation directory.

Typical Usage:

```
cd <my_sim_directory>
survey -t [Points|Shapes]
```

Where:

- This usage defaults the simulation directory (-i) to . (the current directory). It defaults the products
directory (-p) to ./Vis-Tools/<md5_of_script>/

- Points|Shapes (str): Optional. The type of nodes visualization desired. Defaults to Points if not specified.

- The resulting preprocessing script will be written to <my_sim_directory>/VisTools/<md5_of_script>/preprocess_sim.py

**1.4. Preprocessing simulation data**
To run the preprocessing script you’d do cd <my_sim_directory> # if not already there 
Vis-Tools/<md5_of_script>/preprocess_sim.py

class vis_tools.Survey.Surveyor(dir_path='', specific_demo_path=None, sim_id=None)
Bases: object

Class to survey a sim directory for inputs and outputs.

This class looks at a given directory that includes simulation inputs and outputs and collects file paths and in some cases reads the files there to determine key parameters such as number of timesteps and demographics.

Public members: sim_dir (str): dir_path if len(dir_path) > 0 else None

    sim_id (str): simulation id if given in constructor else None
    out_dir (str): simulation output directory if found or None
    assets_dir (str): Assets directory if found or None
    asset_map_path (str): Path to VtAssetMap.json file if found or None
    config_path (str): Config path if found or None
    config (obj): Config object read from config_path if found or None
    demographics_path_or_list (str|list): If a string, the single demographics file path. If a list, the array of demographics files referenced in the config.json.
    demographics (obj): Demographics object read from demographics_path_or_list if found or None. If there are overlays, they are already applied here.
    inset_path (str): InsetChart.json path if found or None
    malaria_filtered_path (str): ReportMalariaFiltered.json path if found or None
    human_mig_tracking_path (str): ReportHumanMigrationTracking.csv path if found or None
    event_recorder_path (str): ReportEventRecorder.csv path if found or None
    spatial_reports (obj): SpatialReports object if out_dir is found or None
    timestep_count (int): Timestep count read from config.json if found or None

k_default cesium_view_margin_fraction = 0.28

k_min_shape_px = 2.0

survey (dir_path)

Survey the given simulation directory for input and output files.

    If a Surveyor object is passed a dir_path at construction, this method is called automatically. If you construct a Surveyor with no dir_path, you can use this method to later associate a dir_path with the Surveyor and cause that directory to be surveyed.

    Returns None.

    Parameters dir_path (str) – Directory path to survey.

    Raises I/O, csv, JSON, and custom exceptions.

summary ()

Print a text summary of the Surveyor object.

This method prints a summary of the Surveyor object that includes the source directory, number of timesteps (if found), and number of nodes (if found).

    Returns None. Prints to stdout.
get_event_names()

Gets a sample set of event names from the ReportEventRecorder.csv.

This method reads the first 100 lines off the ReportEventRecorder.csv and returns a list of the unique event names therein. This is not a comprehensive list of all the event types in the file.

Returns a set of event names, or None.

Return type set

Parameters None. –

Raises I/O and csv exceptions. –

get_node_attributes_simple_fields()

Gets the set of non-object field names from NodeAttributes.

This method looks at the first node in the demographics file and collects all the field names that are simple key-value pairs. So it gets fields like InitialPopulation and BirthRate, but doesn’t get InitialVectorsPerSpecies, since its value is an object. Only looks at the first node, so it is not technically comprehensive. It also removes the “known” fields of Latitude, Longitude, and Altitude.

Returns a set of simple field names from NodeAttributes or None.

Return type set

Parameters None. –

calc_default_shape_size()

Returns a default size for shape-based visualizations.

For shape-based geospatial visualizations, this method calculates a default size in meters that will hopefully make the nodes visible at the default zoom in the geospatial client.

Returns size in meters

Return type float

Parameters None. –

class vis_tools.Survey.PrepMaker

Bases: object

Class to emit a preprocessor script given a Surveyor object.

This class generates a default text python preprocessing script that includes sections various sections based on the surveyed contents of a simulation directory.
**static emit** *(survey, nodes_type, out_file_path)*

Emits a default text python preprocessing script.

This method generates a default text python preprocessing script that includes sections various sections based on the surveyed contents of a simulation directory.

**Returns** None.

**Parameters**

- **survey** *(obj)* – A Surveyor object for sim directory.
- **nodes_type** *(str)* – “Points” or “Shapes” - nodes type to emit.
- **out_file_path** *(str)* – Output python script file path.

**Raises I/O exceptions.** –

`vis_tools.Survey.update_visset(visset_path, asset_map_path)`

Updates a visset with COMPS Asset Manager URLs from an asset map.

**Returns** None.

**Parameters**

- **visset_path** *(str)* – Path to visset.json file
- **asset_map_path** *(str)* – Path to VtAssetMap.json file

`vis_tools.Survey.main()`

Main program

**Overview:**

- Parse arguments
- Fixup sim_dir if needed
- Make a Surveyor to survey the sim directory
- Ensure we got node information
- Use PrepMaker to emit the preprocessing Python program.

---

**vis_tools.VisSet module**

**VisSet.py**

This class generates the visset.json file that is the input for the Vis-Tools HTML client component. It reads defaults from a file called “defaultvisset.json” and extends itself with that. The caller fills in some vital information and then can overrides defaults the object with various method calls, then finally emits the visset.json file.

**Note:** This file will eternally appear to have PEP errors in it because it makes reference to members that are not initially created in the constructor. Those members are acquired when then defaultvisset.json is read in and used to extend self. So these PEP errors are not actually errors.

**Minimal usage:**

```python
vis_set = VisSet("MySim")
vis_set.set_target_client("Geospatial")
vis_set.set_node_vis_type("Points")
vis_set.set_products_directory("my_sim/output/")
```
**class vis_tools.VisSet.VisSet**(name='Untitled', verbose=True)

Bases: object

Class that generates visset.json files.

This class provides a friendly way to generate a customized visset.json for Vis-Tools various HTML clients.

Rather than incorporating its defaults directly in code, it instead reads its defaults from defaultvisset.json. This important, because that same file is used by the HTML clients, so that the defaults for both remain in sync.

VisSet extends itself from defaultvisset.json, which is to say, all of the fields from defaultvisset.json become part of the VisSet Python object itself.

There are methods to allow easy access to some aspects of the VisSet, such as adding bindings, showing/hiding sections, and adding various types of visual layers. But the visset data structure is a public part of this object and it is allowable for a preprocessing script to directly poke into it whatever customizations are required. E.g.:

```python
vis_set.options["defaultBaseLayer"] = "ESRI National Geographic"
vis_set.options["insetCharts"]["defaultChannelName"] = "Prevalence"
```

**Public members:**

- **name (str):** The name (given in the constructor).
- **options (obj):** The options section based on the targetClient all keys found in defaultvisset.json

```python
k_default_visset = 'defaultvisset.json'
k_visset_file_name = 'visset.json'
k_infected_migrations_file_name = 'InfectedMigrations.czml'
k_vector_migrations_file_name = 'VectorMigrations.czml'
k_network_layer_file_name = 'Network.czml'
k_geospatial = 'Geospatial'
k_errno_permission_denied = 13
k_nodes_as_points = 'Points'
k_nodes_as_shapes = 'Shapes'
k_node_shape_square = 'Square'
k_marker_shape_triangle = 'Triangle'
k_marker_shape_cross = 'Cross'
k_marker_placement_top = 'Top'
k_marker_placement_left = 'Left'
k_marker_placement_bottom = 'Bottom'
k_marker_placement_right = 'Right'
k_default_marker_duration_days = 2
```

1.4. Preprocessing simulation data
**set_sim_id** *(sim_id)*
Set a sim_id to be associated with this VisSet.

Returns None.

Parameters **sim_id** *(str)* – Simulation ID as a GUID.

**set_asset_map_path** *(asset_map_path)*
Set the path to the optional asset map.

Returns None.

Parameters

• **asset_map_path** *(str)* – Path to a VtAssetMap.json file containing the
  • between local asset paths and COMPS asset manager URLs. If
    (mapping) –
  • map is present, then the output visset.json will contain
    both(this) –
  • paths and asset manager URLs.(local) –

**setVerbose** *(verbose)*
Set whether the class emits verbose messages.

Returns None.

Parameters **verbose** *(bool)* – True for verbose output, False for relative silence.

**get_output_directory** ()
Getter for the output directory, where visset.json will be written. THIS METHOD IS DEPRECATED. Use
get_products_dir instead.

Returns Currently set output directory or None.

Return type **str**

Parameters **None** . –

**get_products_directory** ()
Getter for the products directory, where the visset.json and other artifact files are written.

This function replaces get_output_directory, which had a name that could lead to confusion with the sim-
ulation’s output/directory.

Returns Currently set products directory or None.

Return type **str**

Parameters **None** . –

**get_config** ()
Getter for the config object.

Returns Currently set Config object or None.

Return type **obj**

Parameters **None** . –

**get_demographics** ()
Getter for the demographics object.

Returns Currently set Demographics object or None.

Return type **obj**
Parameters None. –

get_event_recorder()
Getter for the event recorder object.

Returns Currently set CSVReport object for event recorder, or None.

Return type obj

Parameters None. –

set_target_client(client_name)
Setter for the selected target client.

This method sets the target client for the output visset. This causes the public member options to get overwritten with the defaults for the new client name. The allowable client names are provided as class constants.

Returns None.

Parameters client_name (str) – “Geospatial” is the only target right now.

set_node_vis_type(node_vis_type)
Setter for the node visualization type.

For the Geospatial visualization client, there are two fundamental ways nodes may be represented: Points and Shapes. The former are 2D dots, measured in pixels and always perpendicular to the eye vector, which can be sized and styled variously. The latter uses 3D boxes, measured in meters, which can be styled as well but differently from Points. This setter is where the caller chooses which representation they want for their nodes. Each node visualization type has its own suite of settings.

The allowable node_vis_type values are provided as class constants.

Returns None.

Parameters node_vis_type (str) – “Points” or “Shapes”. See constants section.

set_output_directory(output_directory)
Setter for the output directory. THIS METHOD IS DEPRECATED. Use set_products_directory instead.

This setter must be called before write().

Returns None.

Parameters

• output_directory (str) – the directory in which to write the output

• file. (visset.json) –

set_products_directory(products_directory)
Setter for products dir, where preprocessing products are written.

This setter must be called before write() or other methods that write preprocessing artifacts. This method replaces the now-deprecated set_output_directory method, which was replaced because of possible confusion with the simulation output/ directory.

Returns None.

Parameters

• products_directory (str) – the directory into which to write the

• visset.json and other preprocessing artifact files. (output) –
set_demographics(demographics_file_path)
Setter for the demographics file path.

This setter must be called before write(). The Geospatial client requires node locations for visualization. An internal Demographics object is created using the given path. If it is useful to access that object directly it can be obtained using the corresponding getter method. Note that if you have multiple demographics files (i.e. overlays) you can call set_demographics_files() instead.

For: Geospatial visualization.

Returns None.

Parameters demographics_file_path (str) – Path to demographics file.

Raises I/O and JSON exceptions.

set_demographics_files(demographics_file_list)
Setter for demographics from multiple overlay files.

This setter must be called before write(). The Geospatial client requires node locations for visualization. An internal Demographics object is created using the given path. If it is useful to access that object directly it can be obtained using the corresponding getter method. Note that if you have just a single demographics file you can call set_demographics() instead.

For: Geospatial visualization.

Returns None.

Parameters demographics_file_list (list) – Paths to demographics files.

Raises I/O and JSON exceptions.

set_config(config_file_path)
Setter for config file path.

The VisSet needs to know how many timesteps are in the simulation. This can be obtained from any spatial binary, or from the config. This method allows you to specify the path to the config.json, and the VisSet will read the Simulation_Duration from there. Use this method if your simulation does not have any spatial binaries, and you just want to look at the nodes.

Note: If you call add_spatial_reports() and there is at least one spatial binary, the VisSet will obtain the timestep count from that, and you do not need to call set_config at all.

Returns None.

Parameters config_file_path (str) – Path to config.json file.

Raises I/O and JSON exceptions.

set_event_recorder(event_recorder_file_path)
Setter for event recorder file path for marker layers.

This setter must be called before any calls to add_marker_layer(). An internal CSVReport object will be created with the data at the given path.

For: Geospatial visualization.

Returns None.
Parameters `event_recorder_file_path (str) – Path to an ReportEventRecorder.csv.

Raises I/O and csv exceptions.

set_inset_chart (inset_chart_file_path)
Setter for inset chart data source file.

The Geospatial Vis-Tools client has an Inset Panel which shows inset chart data. Typically that data is sourced from an InsetChart.json file, but any file with the same format is allowable, such as ReportMalariaFiltered.json. This method allows the user to specify the source file for Insets.

Note: If this method is not called with some json file path, the Inset Panel will not appear at all in the Geospatial client.

Returns None.

Parameters

• `inset_chart_file_path (str) – File path to InsetChart.json or other file.

set_custom_base_layer (base_layer_path, friendlyName=None)
Setter for custom base layer source file.

The Geospatial Vis-Tools client allows the use of a single-tile globe image as its base layer. Such an image needs to be a high-resolution JPEG or PNG image that is an equirectangular projection.

Returns None.

Parameters

• `base_layer_path (str) – File path to a JPEG or PNG base layer image.

set_start_date (yyyy_mm_dd)
Setter for simulation start date.

If this is not given, the start date (that shows in the timeline in Vis-Tools visualizations) will be back-calculated from the number of timesteps such that the simulation ends on the date on which the preprocessing script was run. This setter allows you to set an arbitrary start date, and the timeline will start there instead.

Returns None.

Parameters

• `yyyy_mm_dd (str) – a date in the form “yyyy-dd-mm”, e.g. “1970-01-01”

add_node_field (field_name)
Allows the caller to pull in additional fields from NodeAttributes.

The output visset.json contains a summarized version of the nodes that are read from the demographics file. By default that summary includes latitude, longitude, and altitude if present. This method allows the caller to pull in any additional desired fields from NodeAttributes into the node summaries in the output visset.json. E.g.:

```
vis_set.add_node_field("BirthRate")
```

Note: Automatically calls update_nodes(). Caller does not need to do that.

Returns None.
**Parameters**

field_name (*str*) – The NodeAttributes field name to pull in.

---

**add_spatial_reports** *(spatial_reports)*

Adds all the spatial reports in the given SpatialReports object.

This method adds as sources all the spatial report files referenced in the given SpatialReports object. Note that the current Geospatial client HTML loads all the spatial binaries it knows about into memory, so it may make sense to edit down the set of reports that are included. That can be done (subsequent to calling this method) using exclude_spatial_channels().

**Note:** This method also, as a side effect, determines the number of timesteps in the simulation by inspecting the first spatial binary. As a result, if spatial binaries are provided, a config.json need not be added to the VisSet object.

**Returns** None.

**Parameters**

spatial_reports (*obj*) – the SpatialReports object to add.

---

**add_marker_layer***(event_name, marker=None, color=None, placement=None, duration=None, show=True)*

Adds a marker animation layer to the output visset.

This method generates a CZML animation layer that provides visual indication of event recorder events.

**Note:** Call set_products_directory, set_demographics, set_event_recorder, and (add_spatial_channels or set_config) before calling this method.

**For:** Geospatial visualization.

**Returns** None. Creates a CZML file in the VisSet’s products directory.

**Parameters**

- **event_name** (*str*) – The name of the event for which a marker layer is
  - It must exactly match the event name in the event (desired.)—
  - E.g. "Received_Treatment". (recorder.)—
  - marker (*str*) – The name of the marker image to use. “Triangle” or
  - are allowable ("Cross")—
  - color (*str*) – An HTML/CSS or SVG named color for the marker. E.g.
  - or "#ccff00"("red")—
  - placement (*str*) – The location, with respect to the node lat/log
  - for the marker. "Top", "Left", "Bottom", or "Right" are (center)—
  - allowable (see constants at top of class)—
  - duration (*int*) – The duration, in timesteps for which the marker will
  - visible after the event occurs. E.g. 2 will cause the (remain)—
to appear above the node on the timestep of the event,

• remain there for two timesteps before disappearing. (and)

• show (bool) – Whether this layer will be visible by default when the

• client loads this visset. If show is False, the layer

• is still generated and loaded by the client, but the layer

• have its checkbox unchecked (will)

• the client when the page is first loaded. (in)

add_infected_human_migrations (human_migration_file_path)
Add an infected human migrations layer to the output visset.

Note: Call set_products_directory, set_demographics, set_event_recorder, and (add_spatial_channels or set_config) before calling this method.

For: Geospatial visualization.

Returns None. Creates a CZML file in the VisSet’s products directory.

Parameters

• human_migration_file_path (str) – The file path to the simulation’s

• file. (ReportHumanMigrationTracking.csv)

To do:

• Add arg for comet head start color

• Add arg for comet head end color

• Add arg for comet tail start color

• Add arg for comet tail end color

• Add arg for tail duration in timesteps

add_vector_migrations (vector_migration_path, migration_duration_timesteps=2,
dot_color='#ba7fb7', dot_size_pixels=10, path_color='#ffffff',
path_thickness_pixels=2.5, path_trail_time_factor=0.75)
Add a vector cohort migration layer to the output visset.

Note: Call set_products_directory, set_demographics, and either set_config or add_spatial_reports before calling this method.

For: Geospatial visualization.

Returns None. Creates a CZML file in the VisSet’s products directory.

Parameters
- `vector_migration_path (str)` – The path to the file.
- `file. (ReportVectorMigration.csv)` –
- `migration_duration_timesteps (int)` – Number of timesteps that the animation spans.
- `dot_color (string)` – CSS #rrggbb color for migration comet dot.
- `dot_size_pixels (int)` – Size in pixels of comet dot.
- `path_color (string)` – CSS #rrggbb color for migration comet tail.
- `path_thickness_pixels (float)` – Thickness in pixels of comet tail.
- `path_trail_time_factor (float)` – Length of trail as a multiple of migration duration. E.g. if this is 1.0, the trail length will be the full distance from source node to the destination node. If (be) the trail length will be half the distance between the nodes. (0.5,)

```python
add_simplified_vector_migrations (vector_migration_path, migration_duration_timesteps=2, arrow_color='#ffffff', arrow_thickness_pixels=2)
```

Add a simplified vector cohort migration layer to the output visset.

**Note:** Call `set_products_directory`, `set_demographics`, and either `set_config` or `add_spatial_reports` before calling this method.

**For:** Geospatial visualization.

**Returns** None. Creates a CZML file in the VisSet’s products directory.

**Parameters**

- `vector_migration_path (str)` – The path to the file.
- `file. (ReportVectorMigration.csv)` –
- `migration_duration_timesteps (int)` – Number of timesteps that the animation spans.
- `dot_color (string)` – CSS #rrggbb color for migration comet dot.
- `dot_size_pixels (int)` – Size in pixels of comet dot.
- `path_color (string)` – CSS #rrggbb color for migration comet tail.
- `path_thickness_pixels (float)` – Thickness in pixels of comet tail.
- `path_trail_time_factor (float)` – Length of trail as a multiple of migration duration. E.g. if this is 1.0, the trail length will be the full distance from source node to the destination node. If (be) the trail length will be half the distance between the nodes. (0.5,)

```python
add_binding (vis_section, sink_name, source_name, func)
```

Add a binding between a data source and a visualization sink.

This method lets you add a binding between a visualization sink (e.g. point node color) and a data source (e.g. a spatial report) through a binding function (e.g. “scale(3, 20)”). Such bindings will be preset in the client page when it is first presented from the visset.
Note: The available sinks vary depending on target client, and within a target client, on other factors. For example, there is one set of sinks for “Points” node representation, and a different set of sinks for “Shapes” node representation. You can see the list of sinks by inspecting the defaultvisset.json file.

Note: The correctness of the functions provided in the func argument is not validated here. If the caller makes an error in the syntax of the function string, the errant string will still end up in the output visset.json. When the client loads the visset, the binding will throw an exception and the client will show an error message.

Returns True if the binding was set, or False if the sink could not be found.

Return type bool

Parameters

• **vis_section**(str) – The visualization section of the visset.json file. For the Geospatial client, valid values for vis_section are “nodeVis” or “heatmapVis”. Other clients will have other visualization sections. See defaultvisset.json.

• **sink_name**(str) – The name of the sink to which to bind. E.g. “shapeExtrusion”.

• **source_name**(str) – The name of the data source to which to bind. E.g. “SpatialReport_Prevalence” or “BirthRate”. Can be None.

• **func**(str) – The function to use on the binding or None. The details of function syntax are documented elsewhere, but in general this may be “none()”, some built-in function e.g. “scale(3, 20)”, or a custom Javascript function body, e.g.:

```
"{ return 1.0 - binding.value; }"
```

**static opacity_one**(weight, norm_weight)

**add_weighted_network_layer**(network, gradient_spec, layer_name='Network', layer_file_name='Network', opacity_func=None)

Adds a weighted network visualization layer to the output visset.

This method generates a CZML layer that provides a visual representation of a weighted network between nodes.

Note: Call set_products_directory and set_demographics before calling this method.

For: Geospatial visualization.

Returns None. Creates a CZML file in the VisSet’s products directory.

Parameters

• **network**(array) – array of objects:

```
{
    from: <from-node-id>,
    to: <to-node-id>,
    weight: <float-weight>
}
```
- \textbf{gradient\_spec} (\textit{str}) – gradient spec for a gradient with which to
- \textbf{the network lines} (\textit{color}) –
- \textbf{layer\_name} (\textit{str}) – Layer name (to be displayed in the client UI).
- \textbf{layer\_file\_name} (\textit{str}) – The layer base name for the output file. For
- "Network" results in an output file called "Network.czml".
- \textbf{opacity\_func} (\textit{function}) – function(weight, norm_weight) that returns
- \textbf{desired opacity in range} $[0,1]$ (the –

\textbf{show\_layer} (\textit{vis\_section})
Sets a layer to be shown by default in the output visset.json.
This method lets the caller set that the particular visualization section is shown by default when the visset
is loaded by the client.

\textbf{Returns} None.

\textbf{Parameters}
- \textbf{vis\_section} (\textit{str}) – The visualization section of the visset.json
- \textbf{For the Geospatial client, valid values for vis\_section} (\textit{file.}) –
- "nodeVis" or "heatmapVis". Other clients will have other
- (\textit{are} –
- \textbf{sections}. See defaultvisset.json. (\textit{visualization}) –

\textbf{hide\_layer} (\textit{vis\_section})
Sets a layer to be hidden by default in the output visset.json.
This method lets the caller set that the particular visualization section is hidden by default when the visset
is loaded by the client. Note that all the data and bindings for that visualization section are still created and
emitted to the visset...only their default visibility is affected.

\textbf{Returns} None.

\textbf{Parameters}
- \textbf{vis\_section} (\textit{str}) – The visualization section of the visset.json
- \textbf{For the Geospatial client, valid values for vis\_section} (\textit{file.}) –
- "nodeVis" or "heatmapVis". Other clients will have other
- (\textit{are} –
- \textbf{sections}. See defaultvisset.json. (\textit{visualization}) –

\textbf{exclude\_spatial\_channels} (\textit{spatial\_channel\_names})
Exclude named channels from the list of binding sources.
Allows caller to exclude a set of spatial channels from showing up as sources for visualization bindings.
This would primarily be done to reduce the memory pressure (or load time) of the client page, since all
the source spatial binaries have to be loaded into memory on the client side. All this really does is set
show=False in the link, which excludes the source on the client side.
Note: Call add_spatial_reports() before calling this method.

**Returns** None.

**Parameters** `spatial_channel_names (list)` – a list or array of spatial channel names.
If only one channel is to be excluded, enclose it in parens or [] so that it is iterable. E.g.:

```python
eclude_spatial_channels(["SpatialReport_Adult_Vectors"])
```

**include_spatial_channels** (`spatial_channel_names`)
Include (only) named channels in the list of binding sources.

Allows caller to include a subset of the available spatial channels as the list of possible sources for visualization bindings. This would primarily be done to reduce the memory pressure (or load time) of the client page, since all the source spatial binaries have to be loaded into memory on the client side. All this really does is set the specified channels to show=True and all the rest to show=False.

Note: Call add_spatial_reports() before calling this method.

**Returns** None.

**Parameters** `spatial_channel_names (list)` – a list or array of spatial channel names.
If only one channel is to be excluded, enclose it in parens or [] so that it is iterable. E.g.:

```python
include_spatial_channels(["SpatialReport_Adult_Vectors"])
```

**exclude_nodes_from_stats** (`node_id_array`)
Excludes a set of nodes from node min/max statistics.

This method lets the caller specify a list of node ids that should be excluded from the min/max calculations data channels. This includes both static sources (e.g. `NodeAttributes.InitialPopulation`) and spatial reports (e.g. `SpatialReport_Population`). This is typically used to drop an outlier from the data set to avoid visually skewing the output visualization. For example, one might exclude the “elsewhere” node, which typically has extraordinary population, prevalence, etc. in a simulation that incorporates external migration.

Note: The node itself is not eliminated - it will still be visually represented, but it will have a value that may lie outside the min/ max range calculated for a given source. The client will clamp the value at visualization time.

Note: This method calls update_nodes... the caller need not do that.

**Returns** None.

**Parameters** `node_id_array (list)` – a list or array of node id’s to be excluded. If only a single node is to be excluded, enclose it in parens or [] so that it is iterable, e.g.:

```python
exclude_nodes_from_stats([10001])
```

or just call `exclude_node_from_stats()`
**exclude_node_from_stats** *(node_id)*

Exclude a node from node min/max statistics.

This method is exactly `exclude_nodes_from_stats` but for a single node id. See that method for details.

Returns None.

Parameters `node_id` (`int`) – Node id to exclude from stats.

**update_nodes** ()

Updates the local summarized nodes from the Demographics source.

This method forces the VisSet object to refresh its summarized copy of the node data that it will later include in the written visset.json. Use this method if you change the data in the Demographics object and want those changes reflected in the output visset.json.

Returns None.

Parameters None. –

**update_ranges** ()

Updates the local min/max values for extra node fields.

This method updates the local min/max values that are calculated and later emitted into the visset.json file. Use this method if you change the values in the Demographics object.

Returns None.

Parameters None. –

**validate** ()

Validate the settings in the visset to ensure their validity.

This function validates the visset to increase the chances of it working correctly when brought into the client.

Returns None.

Parameters None. –

To do:

- Validate targetClient
- Validate nodeVis.visType
- Validate that self.nodes is populated
- Validate that self.timestepCount and startDate are populated
- Try to guess whether the list of spatial binaries is too big to be loaded on the client side
- Validate binding sources have show=true
- Validate binding functions to the extent possible
- Validate that the inset chart file is a JSON and contains the expected keys

**write** *(sim_url_root=None)*

Write the customized VisSet object to a visset.json file in the products directory.

This method is called after the caller has made all their additions and modifications to the default visset to write a visset.json file to the products directory.

Returns None. Writes a “visset.json” file to the VisSet’s products directory.

Parameters
• **sim_url_root** *(str)* – Optional. Url root substitution if desired. If provided, URLs in the output visset will begin with "./". 

**Raises I/O and JSON exceptions.**

**write_as** *(file_name, sim_url_root=None)*

Write the customized VisSet object to a specified JSON file.

This method is called after the caller has made all their additions and modifications to the default visset to write a JSON file with the given name to the products directory.

**Returns** None. Writes a JSON file to the VisSet’s products directory.

**Parameters**

• **file_name** *(str)* – The file name to use for the output visset JSON

• **file** –

• **sim_url_root** *(str)* – Optional. Url root substitution if desired. If provided, URLs in the output visset will begin with "./". 

**Raises I/O and JSON exceptions.**

**static update_urls** *(visset, asset_map_or_url_root)*

Integrate COMPS Asset Manager URLs or update url roots.

This function does two different things, but they are encapsulated into this one function because this function is the only spot in VisSet that knows where all the URLs are in visset[links].

If passed an asset map, it adds “url_asset” entries next to “url” entries to allow the visset to be used in a hosted environment.

If passed a string, the “url” entries are updated to replace “./” with the given root. This makes the local paths suitable for use with a locally-hosted Vis-Tools.

**Returns** None.

**Parameters**

• **visset** *(obj)* – A visset read with json.load or a VisSet __dict__.

• **asset_map** *(obj|str)* – Either an asset map read with json.load, or

• **string to be the new root for urls** *(a)* –

---

**Module contents**

**1.5 Geospatial visualization**

This section describes in detail the Vis-Tools Geospatial visualization client. It allows you to visualize simulation inputs and outputs on a 3D globe of earth. The client affords lots of flexibility to give you control over just how your data is displayed.

**Note:** Vis-Tools is built to support multiple visualization clients, based on whatever technology is best for the task at hand. The first visualization client is oriented towards geospatial data, but subsequent versions of Vis-Tools will feature other clients for other visualization scenarios.
1.5.1 Running the Geospatial client

Running from COMPS

To use the Geospatial client from COMPS, locate a spatial simulation in COMPS Explore view and click on the Spatial tab of the Details pane. To run the Vis-Tools Geospatial client, click the Visualize button.

For more details on using Vis-Tools from COMPS, see Running Vis-Tools from COMPS.

Running standalone

To use the Geospatial client, run the Vis-Tools Python web server as described in Start the web server. The web server will automatically launch the Geospatial client to display demo data. The next section describes how to point the client at other data.

1.5.2 URL and visset

The data displayed by the Geospatial client is controlled by a URL parameter. Here is the default URL dispatched by the Vis-Tools Python web server:

```
```

Within that URL, note the set=/zambia_data/Vis-Tools/shapes_demo/visset.json. That is URL parameter that tells the client which data to load. If you had a set of preprocessed data (see Preprocessing simulation data) in a directory called my_sim within the Vis-Tools directory, you would typically use an URL like:

```
```

(Where “…” represents the hexadecimal hash directory created by Vis-Tools’s survey program.)

Note: You can bookmark your customized URL for faster access. Bookmarks to simulations you create with standalone Vis-Tools will work as long as the Vis-Tools Python web server is running. Bookmarks to simulations you’ve preprocessed in COMPS will work any time.

Tip: The Visualize button in COMPS is actually a link. You can Ctrl-click it to open the Geospatial visualization in a new tab, or right-click it to copy the URL or bookmark it.
You can in fact have as many vissets in your simulation directory as you want. You might do that in order to have customized vissets for demonstrating various aspects of your simulation. See the section *Finish preprocessing* in *Components of a preprocessing script* to see how to write out vissets with different names.

**Interface overview**

This topic describes the user interface elements in the Vis-Tools Geospatial client. The Geospatial client provides both data visualization and a user interface for manipulating various aspects of the visual presentation. It is built for both experimentation and presentation. Here’s an overview of the window’s parts:

![Interface overview diagram](image)

**Note:** The figure above shows a visset that uses the *shapes* node representation. Some aspects of the window will differ for vissets that use “points” representation, which are noted later. For more information about the two node representations, see *Points versus shapes*.

**Note:** You can make the browser window any size you like, including full screen for presentation purposes. Note that the bigger the window, the harder that your computer’s GPU will have to work to render animation frames, so you may see a degradation in animation performance for large windows. You can also make the window smaller to make room for other windows on your computer, but below 1024 x 768, some user interface elements may start to overlap.

**Contents**

- *Manipulating the globe*
- *Top-row controls and message area*
Manipulating the globe

The Geospatial client is built atop the excellent open-source Cesium global mapping package, and follows the conventions set forth by that package for manipulation of the globe itself.

The Geospatial client will automatically fly to your simulation’s nodes when it is first launched, but you will likely find yourself moving around the map using these mouse gestures.

Top-row controls and message area

The top of the window contains some fundamental sections.

The Message area shows informational and error messages. The Date & timestep area shows the date and timestep number of the current location in the timeline. (The timeline itself is not shown in the figure. It is located at the bottom of the Geospatial client window. See Timeline.)

Note: The timestep-to-date mapping is under the control of your preprocessing script. Preprocessing scripts generated by survey will by default choose a simulation start date such that the simulations ends at the date the preprocessing script itself is run. To override this behavior, you can set the start date in the visset by adding a line to your preprocessing script:
To the right of those areas are some buttons with the following functions:

<table>
<thead>
<tr>
<th>Button</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Help" /></td>
<td><strong>Help.</strong> This button shows the built-in help dialog box with a subset of the information in this document.</td>
</tr>
<tr>
<td><img src="image" alt="COMPS" /></td>
<td><strong>COMPS.</strong> This button, which is only present if the simulation has a COMPS sim ID, returns to the COMPS explore view for the present simulation.</td>
</tr>
<tr>
<td><img src="image" alt="Save" /></td>
<td><strong>Save.</strong> This button saves the current visualization settings for the Geospatial client to a file on your local computer.</td>
</tr>
<tr>
<td><img src="image" alt="Load" /></td>
<td><strong>Load.</strong> This button lets you reload a set of visualization settings you previously saved with the <strong>Save</strong> button.</td>
</tr>
<tr>
<td><img src="image" alt="Aggregate inset charts" /></td>
<td><strong>Aggregate inset charts.</strong> This button, which is only present if an inset chart file is specified in the preprocessing script, toggles the visibility of the aggregate inset chart panel. See <a href="#">Aggregate inset charts</a>.</td>
</tr>
<tr>
<td><img src="image" alt="Recenter" /></td>
<td><strong>Recenter.</strong> This button causes the Geospatial client to recenter the view onto the bounding box of your nodes, looking straight down.</td>
</tr>
<tr>
<td><img src="image" alt="Play/Pause" /></td>
<td><strong>Play/Pause.</strong> This button starts/stops animation playback of the simulation's data. How fast the playback advances depends critically on the number of nodes, timesteps, and layers being displayed.</td>
</tr>
<tr>
<td><img src="image" alt="Pause" /></td>
<td><strong>Pause.</strong> This button pauses playback of the simulation animation.</td>
</tr>
<tr>
<td><img src="image" alt="Base map" /></td>
<td><strong>Base map.</strong> This button, which is part of the underlying Cesium mapping package, allows you to choose a base map layer from a list of available base maps, some of which are satellite-based, and others cartographic.</td>
</tr>
</tbody>
</table>

**Note:** The base map menu is provided by the underlying Cesium mapping library. Along with setting the base map layer, the menu also provides the ability to display 3D terrain, which is turned off by default. While you can enable terrain in the Geospatial client, most of the visualization will become invisible, because it will be at 0 meters altitude, and thus rendered beneath the terrain. Therefore you should generally stick to the “WGS Ellipsoid” terrain map, which is a plain ellipsoid approximation of the Earth.
Vis-Tools

Timeline

At the very bottom of the window is the Timeline.

The timeline is a part of the Cesium mapping package. The blue Current time indicator shows the current date in the simulation, and the range of dates in the data. You can drag the indicator with the mouse to “scrub” time, updating the visualization as you drag.

Side panels

The window has two retractable panels on the left and right. Both panels share some common elements, as shown here.

The Expand/retract button lets you show or hide the panel with a single click. When the panel is expanded, you can also resize it by dragging the inside edge.

Left panel

The left panel contains controls for showing and hiding the various layers that have been created in the preprocessing step, and controlling their appearance. The exact set of elements in this panel varies depending on the simulation and preprocessing script.

The controls for each layer type are described in detail below. See Layer controls.
Right panel

The right panel contains the per-node inset charts, node information, and node search area. The inset charts and node information are only available when a node is selected (by clicking on the node on the map).

To search for a particular node, type the node ID into the search box and click the magnifier search button. The node’s information and charts will be shown, and the view will pan to center the found node.

Aggregate inset charts

If an inset chart file is specified in the preprocessing script (see Inset chart), the inset panel and toggle button will be available. This panel shows any one of the aggregate channels from the simulation’s output. To change the channel, use the channel select control at the top of the window.

Per-node inset charts

The right panel in the Vis-Tools Geospatial client shows information and inset charts for the selected node.

To select a node, click on its node point or shape. To deselect, click anywhere on the map where there is no node.

When a node is selected, a selection indicator will appear on it. The appearance of the selection indicator is different for points and shapes nodes.

The number and content of per-node inset charts depends on the spatial reports in the associated visset. The panel will scroll if necessary.

The green line in the chart indicates the current timestep. To seek the visualization to a different timestep, you can click directly in any graph.

Note: The color of the chart traces and the current-time bar can be customized in the Vis-Tools preprocessing step.
1.5. Geospatial visualization
Keyboard shortcuts

The Vis-Tools Geospatial client has some keyboard shortcuts that can be useful when experimenting with settings or during presentation. For elements with a tooltip (such as buttons), the keyboard shortcut is shown in the tooltip.

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Toggle the built-in help panel.</td>
</tr>
<tr>
<td>E</td>
<td>Toggle the right panel open/closed.</td>
</tr>
<tr>
<td>F</td>
<td>Toggle flat (equirectangular) map projection.</td>
</tr>
<tr>
<td>I</td>
<td>Toggle the aggregate inset chart panel (if present) open/closed.</td>
</tr>
<tr>
<td>L</td>
<td>Equivalent to clicking the Load settings button.</td>
</tr>
<tr>
<td>P</td>
<td>Toggle presentation mode - show/hide all onscreen controls.</td>
</tr>
<tr>
<td>R</td>
<td>Recenter the map onto the nodes bounding box.</td>
</tr>
<tr>
<td>S</td>
<td>Equivalent to clicking the Save settings button.</td>
</tr>
<tr>
<td>W</td>
<td>Toggle the left panel open/closed.</td>
</tr>
<tr>
<td>1 - 9</td>
<td>Toggle the visibility of layers. “1” toggles Nodes, “2” toggles Heatmap, and subsequent numbers toggle whatever animation layers are present in the order they appear.</td>
</tr>
<tr>
<td>Space</td>
<td>Play/pause timeline animation.</td>
</tr>
<tr>
<td>Home</td>
<td>Seek time bar to beginning of simulation.</td>
</tr>
<tr>
<td>End</td>
<td>Seek time bar to end of simulation.</td>
</tr>
<tr>
<td>Left arrow</td>
<td>Step backward in time (4 hours by default).</td>
</tr>
<tr>
<td>Shift-left</td>
<td>Step backward in time by one full timestep.</td>
</tr>
<tr>
<td>Right arrow</td>
<td>Step forward in time (by four hours by default).</td>
</tr>
<tr>
<td>Shift-right</td>
<td>Step forward in time by one full timestep.</td>
</tr>
</tbody>
</table>

**Note:** The amount of time by which the arrow keys advance/retard time can be set during the Vis-Tools preprocessing.
1.5. Geospatial visualization
Layer, gradient, and binding controls

This topic describes the controls available in Vis-Tools to manage separate visualization layers, define the color gradients used to represent data, and control how data is bound to different visual elements in the Geospatial client.

Contents

- Layer controls
  - Nodes layer
    - Points visual parameters
    - Shapes visual parameters
  - Heatmap layer
  - Animation layers
- Gradient controls
- Binding controls
Layer controls

All layer control sections have a title bar.

The title bar has a Layer toggle checkbox that controls the layer’s visibility, a layer title, and an Expand/contract button to reduce the control block down to just the title bar.

Nodes layer

The nodes layer controls affect the visual representation of a simulation’s nodes. The layer provides a set of bindable visual parameters that let you drive the visuals using the inputs and outputs of the simulation. For example, you could bind the visual size of the node to the population in that node, so that more populous nodes have larger dots.

The nodes layer controls differ slightly depending on whether the visset specifies points or shapes node representations. Both are shown here.

Here we’ll examine the parts that make up the nodes layer controls.

The first section are the controls for the color gradient. (See Gradient controls.)

Below that is the Scale slider, which uniformly scales the size of the nodes in the visualization. Increasing the scale can help if nodes are too small or don’t stand out enough against the base map. It is also valuable when using shape nodes (which are sized in meters) while zoomed way out.

The last section are the Binding controls, which link simulation data channels with visual parameters. Usage of the Binding controls is discussed below in Binding controls, but the visual parameters that can be controlled for nodes are as follows (separately for points and shapes).
Vis-Tools

Chapter 1. Documentation structure

Gradient controls

Node scale slider

Bindings

Nodes

Scale

Color: Infectious Vectors Count
Opacity: None
Outline color: Birth Rate
Outline thickness: None
Size: Initial Population

Quant: None

SampleGradient...
Points visual parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>Controls node fill color as a color string, like “red” or “#00ff00”.</td>
</tr>
<tr>
<td>Opacity</td>
<td>Controls the node’s opacity, which has a numeric range from zero to one.</td>
</tr>
<tr>
<td>Outline color</td>
<td>Controls the color of the outline around the filled node, as a color string like “red” or “#00ff00”.</td>
</tr>
<tr>
<td>Outline thickness</td>
<td>Controls the thickness of the outline around the filled node as a numeric non-negative value in pixels.</td>
</tr>
<tr>
<td>Size</td>
<td>Controls the size of the filled node point, as a non-negative value in pixels.</td>
</tr>
</tbody>
</table>

For more information about bindings see Binding controls.

Shapes visual parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td>Controls node fill color as a color string, like “red” or “#00ff00”.</td>
</tr>
<tr>
<td>Extrusion</td>
<td>Controls the extruded height of the node rectangle from the surface of the earth, as a numeric non-negative value in meters.</td>
</tr>
<tr>
<td>Opacity</td>
<td>Controls the node shape’s opacity, which has a numeric range from zero to one.</td>
</tr>
<tr>
<td>Size</td>
<td>Controls the size of the filled node shape, as a non-negative value in pixels.</td>
</tr>
</tbody>
</table>

For more information about bindings see Binding controls.

Heatmap layer

The heatmap layer provides a way to show aggregated data by populating a heatmap image from simulation data then overlaying that onto the map. It makes sense for any data that aggregates well, such as populations of humans or vectors. It also can be useful for just finding which nodes have certain characteristics, such as prevalence or infected humans/vectors.

The first section are the controls for the color gradient. (See Gradient controls.)

The Size slider controls the image resolution of the bitmap overlay image. A larger value here increases the spatial resolution of the heatmap at the expense of memory and animation speed. The default is 1024 pixels in the longest dimension, which is a good trade-off between the two.

The opacity slider allows you to adjust the overall transparency of the heatmap layer. To hide the layer entirely, use the layer checkbox in the title bar rather than setting opacity to zero.

Note: There is a known bug in Vis-Tools that can occur if you run long animations with the heatmap layer turned on, and this problem is exacerbated by large values of the size slider. The underlying Cesium mapping library may run out of memory, causing a “rendering has stopped” error message. Until this problem is resolved, avoid very long runs of animation with the heatmap layer visible.

The last section are the Binding controls, which links a simulation data channel to the heatmap. For more information, see Binding controls.
Fig. 6: Heatmap at 1024 px (left) and 4096 px (right)
Animation layers

Animation layers are generated in the Vis-Tools preprocessing scripts. Examples of animation layers are marker layers and infected human migration. You can have as many animation layers as you want, although showing a large number of them may affect animation performance.

Animation layers themselves don’t have any controls except for their visibility checkbox. In the case of marker layers, the layer title also acts as a legend by showing the layer’s assigned symbol and color.

Gradient controls

The Geospatial client has a standard set of controls for color gradients that are used in both the nodes and heatmap layer controls. This section describes the gradient controls in detail.

The Gradient preview shows the currently selected gradient. To the left of the preview is the Gradient menu button. Clicking this button shows the gradient menu.

The gradient menu shows the available gradients, divided into categories related to their intended use. Notable among these are many gradients sourced from the site Color Brewer 2.0 which is favored in many academic circles. Within the menu, the currently selected gradient’s name is highlighted in dark blue.

Note: You are not limited to using the gradients provided in the Vis-Tools gradients menu. In the preprocessing step, you can install custom gradients into both the node and heatmap controls using a simple text gradient language. The syntax for a text gradient is:
Fig. 7: The gradient menu

\[ \langle \text{color} \rangle @0, [\langle \text{color} \rangle @\langle \text{loc} \rangle, \ldots] \langle \text{color} \rangle @1[\langle r \rangle, [\langle q \rangle \langle \text{quant} \rangle]] \]

Where:

- \( \langle \text{color} \rangle \) is an SVG color like “red” or “lime” or an HTML/CSS-style color in the form “#rrggbbaa” like “#ff0000”. The color may also optionally contain a transparency component (alpha) in the form “rrggbbaa”, e.g. “#ff00007f”, which represents red with 50% transparency.
- \( \langle \text{loc} \rangle \) is the color stop location as a number in the range \([0, 1]\). The locations must increase monotonically from 0 to 1, and color stops at 0 and 1 are required.
- the optional \( \langle r \rangle \) reverses the gradient.
- the optional \( \langle q \rangle \langle \text{quant} \rangle \) quantizes the gradient, where \( \langle \text{quant} \rangle \) is an integer of 2 or more.

Examples:

<table>
<thead>
<tr>
<th>Gradient</th>
<th>Text spec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&quot;#00660000, #00ff0000.25, #ffff0000.5, #ff000001&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;#00660000, #00ff0000.25, #ffff0000.5, #ff000001, r&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;#00660000, #00ff0000.25, #ffff0000.5, #ff000001, r, q5&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;black@0, #183567@0.25, #2e649e@0.5, #17adcb@0.75, #00fafa@1&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;aqua@0, #f2ff0000.5, #fe00fe@1&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;#00000000@0,#00ff007f@0.25,#ffff00ff@.5,#ff0000ff@1&quot; (Example uses alpha.)</td>
</tr>
</tbody>
</table>

To install a text gradient spec for use by nodes, you’d add a line to your preprocessing script like:

```python
vis_set.options["nodeVis"]["gradient"] = "black@0,red@1"
```

or for the heatmap, like:

```python
vis_set.options["heatmapVis"]["gradient"] = "black@0,red@1"
```
To the right of the gradient preview is the **Gradient reverse** button. It reverses the order and position of all the color stops in the current gradient.

On the far left of the top row is the **Quantization menu** that lets you quantize the gradient to a discrete subset of its colors. This can be useful in identifying contours in heatmaps, for example.

At the bottom of the gradient controls is the **Gradient range** slider. This slider allows you to compress the range over which the heatmap’s gradient will show full range. For example, for data ranging from 0 to 1, setting the left slider thumb to 0.2 means that any values less than or equal to 0.2 will show the leftmost gradient color. Likewise, setting the right slider thumb to 0.8 means that any value greater than or equal to 0.8 will show rightmost color.

**Binding controls**

The Geospatial client has a standard set of controls for binding simulation data sources to visual parameters within the client. This section examines how such bindings are made and modified.

The following figure shows the binding section for the *points* representation of nodes.
An example will be instructive. You can follow along with this example by running the Python web server (see Start the web server) and loading this URL:

```
```

Observe that the binding for the visual parameter Size looks like this:

To bind simulation data to a visual parameter, first choose one of your simulation data sources from the source menu.

- On the Size binding row, click the source menu and choose “Population”

Note that the nodes are now sized according to their population. In fact, the node size in pixels is exactly the population, which seemingly works well in this example, but if a node had a population of 1,000, it would obliterate all the nodes around it. Also, some of the nodes have populations of zero or one, so they are too small or disappear entirely. We’ll fix these problems next.
Also note that if you scrub around on the timeline, you will see the node sizes changing, because we have bound a time-varying simulation output. Thus you are visualizing the node populations changing due to births, deaths, and migrations.

For the next step, click the function button (currently labeled None) next to the source menu.

Clicking that button will bring up the Edit function dialog box.

The Edit function dialog box has a top section of information, and a function text box below where we can enter some text. The information line entitled Source shows that we’ve bound Size to the Population simulation output, and also shows us that the range of that data is [0, 46], meaning from 0 to 46 inclusive. (That data range is calculated over all nodes and all timesteps.)

Enter the following (exactly as shown) into the Function text box:
The function \( \text{scale}(3, 20) \) means “normalize the input value to the range \([0, 1]\), then scale it into the range \([3, 20]\)”. Now click \textbf{OK} to install the new function.

Note that now the nodes range from 3 to 20 pixels in size based on their population.

Vis-Tools has a useful set of built-in functions for your use, as well as the ability to use custom user-defined functions. See \textit{Functions reference}. 
Much of the power of the Vis-Tools Geospatial client is afforded by the ability to perform real-time functional manipulation of simulation values when applying those values to visual parameters of the visualization.

Contents

- Built-in functions
  - add()
  - fixed()
  - if()
  - inRange()
  - multiply()
  - none()
  - normalize()
  - sampleGradient()
  - scale()
  - stepwise()
- Custom functions
  - Simple example
  - Basic syntax
  - The binding object
  - More examples

Built-in functions

The Vis-Tools Geospatial client contains a healthy set of built-in functions that cover many of the kinds of manipulations you’re likely to want to carry out. An easy way to remember these functions is to use the built-in Examples toggle in the Edit function dialog box itself.

add()

Adds the constant value $x$ to the input value. Note that $x$ may be negative.

add($x$)

- $value \leftarrow value + x$
- $x$: number, constant to be added to the input value.
- returns: number, input value plus constant $x$.

Examples

- add(4.5) returns value + 4.5.
Edit function

**Visual element**: Size

**Description**: Node point size in pixels

**Return type**: Float

**Source**: Population, values [0, 46]

Function:

```
scale(3, 20)
```

Examples

Example: `scale(3, 16)`
Normalize value, then scale into interval [3, 16].

Example: `normalize()`
Normalize value, return value in interval [0, 1].

Example: `sampleGradient()`
Normalize value, sample gradient for color.

Example: `if(>=, 0.2, 'red', 'blue')`
If value is ≥ 0.2, return 'red' else 'blue'. Operator can be ==, !-, <, <=, >, >-=.

Example: `inRange(0.2, 0.6, '##0000', 'black')`
If 0.2 ≤ value < 0.6, return '##0000' else 'black'.

Example: `stepwise(0.2, 'red', 0.75, 'yellow', ..., 'else', 'green')`
If value < 0.2, returns 'red', if < 0.75, returns 'yellow', ..., else returns 'green'.

Example: `toFixed(##fff0)`
Return invariant value '##fff0'.

Example: `add(-0.5)`
Return value + -0.5.

Example: `multiply(2)`
Return value * 2.0.

Example: `none()`
No function - value is used as-is.

Example: `{ var x = binding.normalize(binding); return 1.0 - x; }`
Custom function body - see documentation for details.
• add(-1) returns value - 1.

fixed()

Ignores the input value and returns $x$ directly. This function can be used even if there is no input source selected to force a particular value for a visual parameter. For example, binding point node size to “None” with the function “fixed(10)” makes all nodes ten pixels in size.

Note that $x$ may be a string or a number. For example, if you are using fixed in a binding to a color visual parameter, return a valid color string. For example, “fixed(‘red’)” or “fixed(‘#00ff00’)”.

fixed($x$)

• $value \leftarrow x$
• $x$: number or string, value with which to override input value.
• returns: same as argument, exactly $x$.

Examples
• fixed(5.5) returns 5.5.
• fixed(“yellow”) returns “yellow”.

if()

Performs an inline conditional on the input value and returns either thenValue or elseValue. This function is similar to the ternary ?: operator in many C-like languages.

if($op$, $compArg$, $thenValue$, $elseValue$)

• $value \leftarrow \begin{cases} \text{thenValue} & value \ op \ compArg \\ \text{elseValue} & \text{otherwise} \end{cases}$
• $op$: string, comparison operator - one of “==”, “!”, “<”, “<=”, “>”, or “>=”.
• $compArg$: number or string, comparison argument, i.e. value $<op> <compArg>$.
• $thenValue$: number or string, value to return if condition is true.
• $elseValue$: number or string, value to return if condition is false.
• returns: number or string (depending on type of $thenValue$ or $elseValue$), $thenValue$ if condition is true; otherwise, $elseValue$.

Examples
• if(“>”, 0.8, “red”, “green”) returns “red” if input value is $> 0.8$, else “green”.
• if(“<”, 0, -1, 1) returns -1 if value is negative, otherwise 1 (equivalent to the sgn(x) function in some languages).
inRange()

Tests whether the input value is in the range [low, high] (inclusive), and returns thenValue if so; otherwise, returns elseValue.

\[
\text{inRange}(\text{low, high, thenValue, elseValue})
\]

\[\begin{align*}
\text{value} & \leftarrow \begin{cases} 
\text{thenValue} & \text{low} \leq \text{value} \leq \text{high} \\
\text{elseValue} & \text{otherwise}
\end{cases} 
\end{align*}\]

• low: number, low constant for range comparison, inclusive.
• high: number, high constant for range comparison, inclusive.
• thenValue: number or string, value to return if input value is in range.
• elseValue: number or string, value to return if input value is out of range.
• returns: number or string (depending on type of thenValue or elseValue), thenValue if value is in range; otherwise, elseValue.

Examples

• inRange(0.1, 0.4, “green”, “red”) returns “green” if input value \(\geq 0.1\) and \(\leq 0.4\), otherwise “red”.
• inRange(-5, 0, -1, 1) returns -1 if value is in range \([-5, 0]\), otherwise 1.

multiply()

Multiplies the input value by the constant \(x\). Note that \(x\) may be negative or a non-integer.

\[
\text{multiply}(x)
\]

• \(\text{value} \leftarrow \text{value} \times x\)
• \(x\): number, constant to be multiplied with the input value.
• returns: number, value \(* x\).

Examples

• multiply(2) returns value \(\times 2\).
• multiply(0.5) returns value \(/ 2\).
• multiply(-1) returns -value.

none()

Passes the input value through untouched.

\[
\text{none()}
\]

• \(\text{value} \leftarrow \text{value}\)
• returns: same type as input value, input value untouched.

Examples

• none() returns the input value as-is.
normalize()

Calculates a normalized form of the input value, based on the range of the input data channel for all timesteps and all nodes, returning a value in the range \([0, 1]\).

**normalize()**

- \(value \leftarrow \text{norm}(value)\)
- returns: a normalized value \([0, 1]\) for the input value.

**Examples**

- normalize() returns the normalized input value.

sampleGradient()

Returns a color by normalizing the input value then sampling the color gradient. This function only make sense for bindings to color visual parameters.

**sampleGradient()**

- \(value \leftarrow \text{sampleGradient}(\text{norm}(value))\)
- returns: color, gradient sampled with normalized value.

**Examples**

- sampleGradient() returns a color based on the normalized input value.

scale()

Returns a value mapped into the given range. The range is treated inclusively.

**scale(min, max)**

- \(value \leftarrow min + \text{norm}(value) \times (max - min)\)
- min: number, minimum value in output range inclusive
- max: number, maximum value in output range inclusive
- returns: number, value mapped into \([min, max]\)

**Examples**

- scale(3, 20) returns a value between 3 and 20 inclusive that represents the input value within that range.
- scale(-1, 1) returns a value between -1 and 1 inclusive that represents the input value within that range (i.e 0 \(\rightarrow -1, 0.5 \rightarrow 0, 1 \rightarrow 1\)).
stepwise()

Returns a stepwise value. It can be used to create custom stepwise gradients or specialized quantizations.

stepwise(step1, firstValue, step2, secondValue, . . . , “else”, elseValue)

- value ←
  \[
  \begin{cases}
  \text{firstValue} & \text{value} < \text{step1} \\
  \text{secondValue} & \text{value} < \text{step2} \\
  \vdots & \vdots \\
  \text{elseValue} & \text{otherwise}
  \end{cases}
  \]

- step1: number, first step argument, where firstValue will be returned if input value is less than step1.
- firstValue: number or string, result to be returned if input value is less than step1.
- step2: number, second step argument, where secondValue will be returned if input value is less than step2.
- secondValue: number or string, result to be returned if input value is less than step2.
- . . . : more step/value pairs.
- “else”: string, the explicit string “else”.
- elseValue: number or string, result to be returned if input value is greater than or equal to the last step.
- returns: number or string, value for matching step; otherwise, elseValue.

Examples

- stepwise(0.2, ‘red’, 0.75, ‘yellow’, ‘else’, ‘green’) returns the string red for input values less than 0.2, the string “yellow” for values between 0.2 and 0.75, and “green” otherwise.
- stepwise(0.25, 0, ‘else’ 1) quantizes input values < 0.25 to 0, otherwise 1.

Custom functions

While for most cases the Vis-Tools built-in functions will suffice, it is possible to use custom user-defined functions for special cases. This section examines that capability in detail.

Simple example

What if we wanted to invert a normalized value? The value naturally ranges from 0 to 1, but we want it to be the other way around, i.e. \( 1 - value \). None of the built-in functions can do that. However, that result and many more can be achieved using a custom function which is essentially a specialized JavaScript function body. Consider this custom function:

```javascript
{ return 1.0 - binding.value; }
```

That function carries out the desired function. It is simple enough, except for the binding reference, which is explained below.
## Basic syntax

As might be inferred from the previous example, the basic syntax of a custom function is:

```javascript
{ return <expression>; }
```

A custom function is the text of a JavaScript function body that will be executed for each node, for each timestep, before a value is applied to the bound visual parameter. The expression can be anything, but typically it is some manipulation of the input value or other available data. The input data and other information are provided to the custom function as an implicit argument called `binding`.

**Note:** Your custom function must be valid JavaScript. If it is not, or the code in the function throws an exception, you’ll see an error message in the message area of the client window.

**Note:** The code for your custom function may be arbitrarily complex. You can declare and use local variables, and use as many statements as you want. The custom function is evaluated as a real JavaScript function, and as long as it ultimately returns a valid return value, it will work. But be warned that your custom function is going to be executed for every node, on every timestep, so complex custom functions will detrimentally affect animation framerate.

## The binding object

The binding object is passed to a custom function and provides access to the raw bound simulation data channel value, along with other potentially useful data.

Here are the most commonly used data members of the binding object.

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>number</td>
<td>The raw data value of the bound data source for the current node and timestep.</td>
</tr>
<tr>
<td>min</td>
<td>number</td>
<td>The minimum value for this data channel over all nodes and timesteps. Along with <code>max</code>, can be used to calculate a normalized value.</td>
</tr>
<tr>
<td>max</td>
<td>number</td>
<td>The maximum value for this data channel over all nodes and timesteps. Along with <code>min</code>, can be used to calculate a normalized value.</td>
</tr>
</tbody>
</table>

The binding object is implicitly available to your custom function, so to use `min` you simply type `binding.min` within your custom function. For example, the following function normalizes the input value, which is functionally equivalent to the built-in `normalize()` function:

```javascript
{ return (binding.value - binding.min) / (binding.max - binding.min); }
```

The above fields are enough to do most numerical manipulations of the input value. There are, however, some additional data members that may be of use for other specialized functions.
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| gradient     | Gradient | The color gradient object for this binding. To use this object, calculate a normalized value, then: 
|              |          | `return gradient;`  
|              |          | `→sampleHashCss(normValue);`  
|              |          | which returns a color string like “#00ff00”. |
| gradientLow  | number   | The low value of the gradient range slider, which is a number in the range [0, gradientHigh). You may choose to incorporate this into your function if you are sampling the gradient directly. |
| gradientHigh | number   | The high value of the gradient range slider, which is a number in the range (gradientLow, 1]. You may choose to incorporate this into your function if you are sampling the gradient directly. |
| node         | Node     | An object describing the current node. This object has the following potentially valuable fields.  
|              |          | • latitude: number, latitude in degrees.  
|              |          | • longitude: number, longitude in degrees.  
|              |          | • altitude: number: altitude in meters.  
|              |          | • nodeId: number or string: node ID from the demographics file.  
|              |          | • other NodeAttribute fields added during the pre-processing step, such as InitialPopulation, BirthRate, etc. |
| timestep     | number   | Current timestep, in the range [0, timestepCount). |
| timestepCount | number   | Number of timesteps in the present simulation. |
More examples

Here are some more examples of things you can do with custom functions.

- For point nodes, to outline a particular node of interest (id 340, for example), bind:
  - Outline color to:
    
    ```
    fixed("red")
    ```
  - Outline thickness to:
    
    ```
    { return (binding.node.nodeId == 340) ? 3 : 0; }
    ```

- To invert a normalized value, use the function:
  
  ```
  { return 1.0 - binding.value; }
  ```

- To apply a log scale to a value, use the function:
  
  ```
  { return Math.log10(binding.value); }
  ```

- To apply a square root to a value, use the function:
  
  ```
  { return Math.sqrt(binding.value); }
  ```

- To apply a random color to each node, bind node Color to the function:
  
  ```
  { return Gradient.presets["Hue"].sampleHashCss(Math.random()); }
  ```

1.6 Glossary

The following terms are used to describe concepts and processes in Vis-Tools.

**animation layer** A layer that visualizes events that occur in time, such as markers for particular events in ReportEventRecorder.csv or human migration.

**Geospatial client** The Vis-Tools client that is primarily designed for the visualization of spatio-temporal simulation output.

**heatmap layer** A layer that provides a way to show aggregated data by populating a heatmap image from simulation data then overlaying that onto the map. It makes sense for any data that aggregates well, such as populations of humans or vectors.

**nodes layer** A layer with controls that affect the visual representation of a simulation’s nodes. The layer provides a set of bindable visual parameters that let you drive the visuals using the inputs and outputs of the simulation.

**points** The visualization output that represents the simulation nodes as two-dimensional circles measured in pixels. This is the default representation.

**preprocessing script** A Python file that runs on your simulation’s input and output files to create the visset file that indicates how your simulation’s data will be visualized in Vis-Tools. This script indicates how to represent nodes, whether to include migration, the colors to use, and more.

**shapes** The visualization output that represents the simulation nodes as three-dimensional rectangles measured in meters.
**survey**  The Python program that is included with Vis-Tools and generates a default preprocessing script from your simulation data. This preprocessing script can then be further modified.

**visset**  A JSON file that is the input to the Vis-Tools HTML client component and describes how the simulation data is to be displayed. It contains a subset of the simulation’s demographics data (specifically, information about its nodes). The visset file is obtained by running a preprocessing script on your simulation’s input and output files.
vis_tools, 79
vis_tools.Color, 47
vis_tools.Config, 49
vis_tools.CSVReport, 42
vis_tools.CZMLWriter, 43
vis_tools.Demographics, 50
vis_tools.Gradient, 52
vis_tools.MigrationHelpers, 53
vis_tools.NamedColors, 54
vis_tools.SpatialBinary, 58
vis_tools.SpatialBinaryHeader, 61
vis_tools.SpatialReports, 62
vis_tools.Survey, 63
vis_tools.VisSet, 66
add() (vis_tools.SpatialReports.SpatialReports method), 62
add_binding() (vis_tools.VisSet.VisSet method), 74
add_combiner() (vis_tools.SpatialBinary.SpatialBinary static method), 60
add_event_markers() (vis_tools.CZMLWriter.CZMLWriter method), 46
add_infected_human_migrations() (vis_tools.VisSet.VisSet method), 73
add_marker_layer() (vis_tools.VisSet.VisSet method), 72
add_migrations() (vis_tools.CZMLWriter.CZMLWriter method), 45
add_node_field() (vis_tools.VisSet.VisSet method), 71
add_nodes() (vis_tools.CZMLWriter.CZMLWriter method), 44
add_simplified_vector_migrations() (vis_tools.CZMLWriter.CZMLWriter method), 46
add_simplified_vector_migrations() (vis_tools.VisSet.VisSet method), 74
add_spatial_reports() (vis_tools.VisSet.VisSet method), 72
add_vector_migrations() (vis_tools.CZMLWriter.CZMLWriter method), 45
add_vector_migrations() (vis_tools.VisSet.VisSet method), 73
add_weighted_network() (vis_tools.CZMLWriter.CZMLWriter method), 47
add_weighted_network_layer() (vis_tools.VisSet.VisSet method), 75
aliceblue (vis_tools.NamedColors.NamedColors attribute), 54
animation layer, 109
antiquewhite (vis_tools.NamedColors.NamedColors attribute), 54
aquamarine (vis_tools.NamedColors.NamedColors attribute), 54
azure (vis_tools.NamedColors.NamedColors attribute), 54
beige (vis_tools.NamedColors.NamedColors attribute), 54
bisque (vis_tools.NamedColors.NamedColors attribute), 54
black (vis_tools.NamedColors.NamedColors attribute), 54
blanchedalmond (vis_tools.NamedColors.NamedColors attribute), 54
blue (vis_tools.NamedColors.NamedColors attribute), 54
blueviolet (vis_tools.NamedColors.NamedColors attribute), 54
bounding_box() (vis_tools.Demographics.Demographics method), 50
brown (vis_tools.NamedColors.NamedColors attribute), 54
burlywood (vis_tools.NamedColors.NamedColors attribute), 54
cadetblue (vis_tools.NamedColors.NamedColors attribute), 54
calc_adjusted_pop_max() (vis_tools.Demographics.Demographics method), 51
calc_default_shape_size() (vis_tools.Survey.Surveyor method), 65
chartreuse (vis_tools.NamedColors.NamedColors attribute), 54
chocolate (vis_tools.NamedColors.NamedColors attribute), 54
clone() (vis_tools.SpatialBinary.SpatialBinary method), 59
collate_infected_human_migrations() (vis_tools.MigrationHelpers.MigrationHelpers
Vis-Tools

static method), 53
Color (class in vis_tools.Color), 47
combine() (vis_tools.SpatialBinary.SpatialBinary static method), 53
darkturquoise (vis_tools.NamedColors.NamedColors attribute), 55
darkviolet (vis_tools.NamedColors.NamedColors attribute), 55
deeppink (vis_tools.NamedColors.NamedColors attribute), 55
deeppskyblue (vis_tools.NamedColors.NamedColors attribute), 55
Demographics (class in vis_tools.Demographics), 50
dimgray (vis_tools.NamedColors.NamedColors attribute), 55
dimgrey (vis_tools.NamedColors.NamedColors attribute), 55
divide_combiner() (vis_tools.SpatialBinary.SpatialBinary static method), 61
dodgerblue (vis_tools.NamedColors.NamedColors attribute), 55

D
darkblue (vis_tools.NamedColors.NamedColors attribute), 55
darkcyan (vis_tools.NamedColors.NamedColors attribute), 55
darkgoldenrod (vis_tools.NamedColors.NamedColors attribute), 55
darkgray (vis_tools.NamedColors.NamedColors attribute), 55
darkgreen (vis_tools.NamedColors.NamedColors attribute), 55
darkgrey (vis_tools.NamedColors.NamedColors attribute), 55
darkkhaki (vis_tools.NamedColors.NamedColors attribute), 55
darkmagenta (vis_tools.NamedColors.NamedColors attribute), 55
darkolivegreen (vis_tools.NamedColors.NamedColors attribute), 55
darkorange (vis_tools.NamedColors.NamedColors attribute), 55
darkorchid (vis_tools.NamedColors.NamedColors attribute), 55
darksalmon (vis_tools.NamedColors.NamedColors attribute), 55
darkseagreen (vis_tools.NamedColors.NamedColors attribute), 55
darkslategreen (vis_tools.NamedColors.NamedColors attribute), 55
darkviolet (vis_tools.NamedColors.NamedColors attribute), 55
deepink (vis_tools.NamedColors.NamedColors attribute), 55
deeppskyblue (vis_tools.NamedColors.NamedColors attribute), 55
Demographics (class in vis_tools.Demographics), 50
dimgray (vis_tools.NamedColors.NamedColors attribute), 55
dimgrey (vis_tools.NamedColors.NamedColors attribute), 55
divide_combiner() (vis_tools.SpatialBinary.SpatialBinary static method), 61
dodgerblue (vis_tools.NamedColors.NamedColors attribute), 55

eemit() (vis_tools.Survey.PrepMaker static method), 65
eemit_nodes() (vis_tools.Demographics.Demographics method), 51
eemit_object() (vis_tools.SpatialReports.SpatialReports method), 63
eemit_object_json() (vis_tools.SpatialReports.SpatialReports method), 63
exclude_node_from_stats() (vis_tools.VisSet.VisSet method), 77
exclude_nodes_from_stats() (vis_tools.VisSet.VisSet method), 77
exclude_spatial_channels() (vis_tools.VisSet.VisSet method), 76

F
firebrick (vis_tools.NamedColors.NamedColors attribute), 55
floralwhite (vis_tools.NamedColors.NamedColors attribute), 55
forestgreen (vis_tools.NamedColors.NamedColors attribute), 55
from_html_hash() (vis_tools.Color.Color static method), 48
fuchsia (vis_tools.NamedColors.NamedColors attribute), 55

gainsboro (vis_tools.NamedColors.NamedColors attribute), 55
Geospatial client, 109
get() (vis_tools.SpatialReports.SpatialReports method), 62
get_config() (vis_tools.VisSet.VisSet method), 68
get_demographics() (vis_tools.VisSet.VisSet method), 68
get_demographics_filename_or_list() (vis_tools.Config.Config method), 49
get_demographics_filenames() (vis_tools.Config.Config method), 49
get_event_names() (vis_tools.Survey.Surveyor method), 65
get_event_recorder() (vis_tools.VisSet.VisSet method), 69
get_first_demographics_filename() (vis_tools.Config.Config method), 49
get_inset_channels() (vis_tools.Survey.Surveyor method), 65
get_node_attributes_simple_fields() (vis_tools.Survey.Surveyor method), 65
get_output_directory() (vis_tools.VisSet.VisSet method), 68
get_products_directory() (vis_tools.VisSet.VisSet method), 68
get_timestep_count() (vis_tools.Config.Config method), 49
ghostwhite (vis_tools.NamedColors.NamedColors attribute), 55
gold (vis_tools.NamedColors.NamedColors attribute), 55
goldenrod (vis_tools.NamedColors.NamedColors attribute), 55
Gradient (class in vis_tools.Gradient), 52
gray (vis_tools.NamedColors.NamedColors attribute), 55
green (vis_tools.NamedColors.NamedColors attribute), 56
greenyellow (vis_tools.NamedColors.NamedColors attribute), 56
grey (vis_tools.NamedColors.NamedColors attribute), 55

H
heatmap layer, 109
hide_layer() (vis_tools.VisSet.VisSet method), 76
honeydew (vis_tools.NamedColors.NamedColors attribute), 56
hotpink (vis_tools.NamedColors.NamedColors attribute), 56

I
include_spatial_channels() (vis_tools.VisSet.VisSet method), 77
incorporate_altitudes() (vis_tools.Demographics.Demographics method), 51
indianred (vis_tools.NamedColors.NamedColors attribute), 56

Index 115
Vis-Tools

k_nodes_as_points  (vis_tools.VisSet.VisSet attribute), 67
k_nodes_as_shapes  (vis_tools.VisSet.VisSet attribute), 67
k_spatial_filename_glob  (vis_tools.SpatialReports.SpatialReports attribute), 62
k_spatial_leaf_re  (vis_tools.SpatialReports.SpatialReports attribute), 62
k_vector.Migrations_file_name  (vis_tools.VisSet.VisSet attribute), 67
k_visset_file_name  (vis_tools.VisSet.VisSet attribute), 67
khaki  (vis_tools.NamedColors.NamedColors attribute), 56
lavender  (vis_tools.NamedColors.NamedColors attribute), 56
lavenderblush  (vis_tools.NamedColors.NamedColors attribute), 56
lawngreen  (vis_tools.NamedColors.NamedColors attribute), 56
lemonchiffon  (vis_tools.NamedColors.NamedColors attribute), 56
lerp()  (vis_tools.Color.Color static method), 48
lightblue  (vis_tools.NamedColors.NamedColors attribute), 56
lightcoral  (vis_tools.NamedColors.NamedColors attribute), 56
lightcyan  (vis_tools.NamedColors.NamedColors attribute), 56
lightgoldenrodyellow  (vis_tools.NamedColors.NamedColors attribute), 56
lightgray  (vis_tools.NamedColors.NamedColors attribute), 56
lightgreen  (vis_tools.NamedColors.NamedColors attribute), 56
lightgrey  (vis_tools.NamedColors.NamedColors attribute), 56
lightpink  (vis_tools.NamedColors.NamedColors attribute), 56
lightsalmon  (vis_tools.NamedColors.NamedColors attribute), 56
lightseagreen  (vis_tools.NamedColors.NamedColors attribute), 56
lightskyblue  (vis_tools.NamedColors.NamedColors attribute), 56
lightslategray  (vis_tools.NamedColors.NamedColors attribute), 56
lightslategrey  (vis_tools.NamedColors.NamedColors attribute), 56
lightsteelblue  (vis_tools.NamedColors.NamedColors attribute), 56
lightyellow  (vis_tools.NamedColors.NamedColors attribute), 56
lime  (vis_tools.NamedColors.NamedColors attribute), 56
limegreen  (vis_tools.NamedColors.NamedColors attribute), 56
linen  (vis_tools.NamedColors.NamedColors attribute), 56
magenta  (vis_tools.NamedColors.NamedColors attribute), 56
main()  (in module vis_tools.Survey), 66
make_index()  (vis_tools.Demographics.Demographics method), 52
make_series()  (vis_tools.CSVReport.CSVReport method), 42
maroon  (vis_tools.NamedColors.NamedColors attribute), 56
mediumaquamarine  (vis_tools.NamedColors.NamedColors attribute), 56
mediumblue  (vis_tools.NamedColors.NamedColors attribute), 56
mediumorchid  (vis_tools.NamedColors.NamedColors attribute), 56
mediumpurple  (vis_tools.NamedColors.NamedColors attribute), 56
mediumseagreen  (vis_tools.NamedColors.NamedColors attribute), 57
mediumslateblue  (vis_tools.NamedColors.NamedColors attribute), 57
mediumspringgreen  (vis_tools.NamedColors.NamedColors attribute), 57
mediumturquoise  (vis_tools.NamedColors.NamedColors attribute), 57
mediumvioletred  (vis_tools.NamedColors.NamedColors attribute), 57
midnightblue  (vis_tools.NamedColors.NamedColors attribute), 57
MigrationHelpers  (class in vis_tools.MigrationHelpers), 53
minimum_deltas()  (vis_tools.Demographics.Demographics method), 51
mintcream  (vis_tools.NamedColors.NamedColors attribute), 57
missing_columns()  (vis_tools.CSVReport.CSVReport method), 43
mistyrose  (vis_tools.NamedColors.NamedColors attribute), 57
moccasin (vis_tools.NamedColors.NamedColors attribute), 57

multiply_combiner() (vis_tools.SpatialBinary.SpatialBinary static method), 60

N

NamedColors (class in vis_tools.NamedColors), 54

navaojwhite (vis_tools.NamedColors.NamedColors attribute), 57

navy (vis_tools.NamedColors.NamedColors attribute), 57

nodes layer, 109

O

oldlace (vis_tools.NamedColors.NamedColors attribute), 57

olive (vis_tools.NamedColors.NamedColors attribute), 57

olivedrab (vis_tools.NamedColors.NamedColors attribute), 57

opacity_one() (vis_tools.VisSet.VisSet static method), 75

orange (vis_tools.NamedColors.NamedColors attribute), 57

orangered (vis_tools.NamedColors.NamedColors attribute), 57

orchid (vis_tools.NamedColors.NamedColors attribute), 57

P

palegoldenrod (vis_tools.NamedColors.NamedColors attribute), 57

palegreen (vis_tools.NamedColors.NamedColors attribute), 57

paleturquoise (vis_tools.NamedColors.NamedColors attribute), 57

palevioletred (vis_tools.NamedColors.NamedColors attribute), 57

papayawhip (vis_tools.NamedColors.NamedColors attribute), 57

peachpuff (vis_tools.NamedColors.NamedColors attribute), 57

peru (vis_tools.NamedColors.NamedColors attribute), 57

pink (vis_tools.NamedColors.NamedColors attribute), 57

plum (vis_tools.NamedColors.NamedColors attribute), 57

points, 109

population_range() (vis_tools.Demographics.Demographics method), 51

powerblue (vis_tools.NamedColors.NamedColors attribute), 57

PrepMaker (class in vis_tools.Survey), 65

preprocessing_script, 109

print() (vis_tools.SpatialBinary.SpatialBinary method), 59

purple (vis_tools.NamedColors.NamedColors attribute), 57

R

read_partial() (vis_tools.CSVReport.CSVReport method), 42

red (vis_tools.NamedColors.NamedColors attribute), 57

remove() (vis_tools.SpatialReports.SpatialReports method), 63

rosybrown (vis_tools.NamedColors.NamedColors attribute), 57

royalblue (vis_tools.NamedColors.NamedColors attribute), 57

S

saddlebrown (vis_tools.NamedColors.NamedColors attribute), 57

salmon (vis_tools.NamedColors.NamedColors attribute), 57

sample() (vis_tools.Gradient.Gradient method), 53

sandybrown (vis_tools.NamedColors.NamedColors attribute), 57

seagreen (vis_tools.NamedColors.NamedColors attribute), 57

seashell (vis_tools.NamedColors.NamedColors attribute), 57

set_asset_map_path() (vis_tools.VisSet.VisSet method), 57

set_config() (vis_tools.VisSet.VisSet method), 57

set_custom_base_layer() (vis_tools.VisSet.VisSet method), 70

set_demographics() (vis_tools.VisSet.VisSet method), 69

set_demographics_files() (vis_tools.VisSet.VisSet method), 70

set_event_recorder() (vis_tools.VisSet.VisSet method), 70

set_inset_chart() (vis_tools.VisSet.VisSet method), 71

set_node_vis_type() (vis_tools.VisSet.VisSet method), 69

set_output_directory() (vis_tools.VisSet.VisSet method), 69

set_products_directory() (vis_tools.VisSet.VisSet method), 69

set_sim_dates() (vis_tools.CZMLWriter.CZMLWriter method), 44

Index 117
set_sim_duration() (vis_tools.CZMLWriter.CZMLWriter method), 44
set_sim_id() (vis_tools.VisSet.VisSet method), 67
set_start_date() (vis_tools.VisSet.VisSet method), 71
set_target_client() (vis_tools.VisSet.VisSet method), 69
set_verbose() (vis_tools.VisSet.VisSet method), 68
shapes, 109
show_layer() (vis_tools.VisSet.VisSet method), 76
sienna (vis_tools.NamedColors.NamedColors attribute), 58
silver (vis_tools.NamedColors.NamedColors attribute), 58
skyblue (vis_tools.NamedColors.NamedColors attribute), 58
slateblue (vis_tools.NamedColors.NamedColors attribute), 58
slategray (vis_tools.NamedColors.NamedColors attribute), 58
slategrey (vis_tools.NamedColors.NamedColors attribute), 58
snow (vis_tools.NamedColors.NamedColors attribute), 58
SpatialBinary (class in vis_tools.SpatialBinary), 58
SpatialBinaryHeader (class in vis_tools.SpatialBinaryHeader), 61
SpatialReports (class in vis_tools.SpatialReports), 62
springgreen (vis_tools.NamedColors.NamedColors attribute), 58
steelblue (vis_tools.NamedColors.NamedColors attribute), 58
subtract_combiner() (vis_tools.SpatialBinary.SpatialBinary static method), 59
summary() (vis_tools.Survey.Surveyor method), 64
survey, 110
survey() (vis_tools.Survey.Surveyor method), 64
Surveyor (class in vis_tools.Survey), 64

T
tan (vis_tools.NamedColors.NamedColors attribute), 58
teal (vis_tools.NamedColors.NamedColors attribute), 58
thistle (vis_tools.NamedColors.NamedColors attribute), 58
to_rgba_array() (vis_tools.Color.Color method), 48
to_rgba_array_str() (vis_tools.Color.Color method), 48
tomato (vis_tools.NamedColors.NamedColors attribute), 58
turquoise (vis_tools.NamedColors.NamedColors attribute), 58

U
update_nodes() (vis_tools.VisSet.VisSet method), 78
update_ranges() (vis_tools.VisSet.VisSet method), 78
update_urls() (vis_tools.VisSet.VisSet static method), 79
update_visset() (in module vis_tools.Survey), 66
validate() (vis_tools.VisSet.VisSet method), 78
value_range() (vis_tools.SpatialBinary.SpatialBinary method), 59
violet (vis_tools.NamedColors.NamedColors attribute), 58
vis_tools (module), 79
vis_tools.Color (module), 47
vis_tools.Config (module), 49
vis_tools.CSVReport (module), 42
vis_tools.CZMLWriter (module), 43
vis_tools.Demographics (module), 50
vis_tools.Gradient (module), 52
vis_tools.MigrationHelpers (module), 53
vis_tools.NamedColors (module), 54
vis_tools.SpatialBinary (module), 58
vis_tools.SpatialBinaryHeader (module), 61
vis_tools.SpatialReports (module), 62
vis_tools.Survey (module), 63
vis_tools.VisSet (module), 66
visset, 110
VisSet (class in vis_tools.VisSet), 67

W
wheat (vis_tools.NamedColors.NamedColors attribute), 58
white (vis_tools.NamedColors.NamedColors attribute), 58
whitesmoke (vis_tools.NamedColors.NamedColors attribute), 58
write() (vis_tools.VisSet.VisSet method), 78
write_as() (vis_tools.VisSet.VisSet method), 79
write_binary() (vis_tools.SpatialBinary.SpatialBinary method), 59
write_czml() (vis_tools.CZMLWriter.CZMLWriter method), 47

Y
yellow (vis_tools.NamedColors.NamedColors attribute), 58
yellowgreen (vis_tools.NamedColors.NamedColors attribute), 58