SUMMON 1.8.2 Manual

Matt Rasmussen

September 15, 2007

Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology

rasmus@mit.edu

Contents

1 Introduction

1.1 What is SUMMON

SUMMON is a python extension module that provides rapid prototyping of 2D visualizations. By heavily relying on the python scripting language, SUMMON allows the user to rapidly prototype a custom visualization for their data, without the overhead of a designing a graphical user interface or recompiling native code. By simplifying the task of designing a visualization, users can spend more time on understanding their data.

SUMMON was designed with several philosophies. First, recompilation should be avoided in order to speed up the development process. Second, design of graphical user interfaces should also be minimized. Designing a good interface takes planning and time to layout buttons, scrollbars, and dialog boxes. Yet a poor interface is very painful to work with. Even when one has a good interface, rarely can it be automated for batch mode. Instead, SUMMON relies on the python terminal for most interaction. This allows the users direct access to the underlining code, which is more expressive, and can be automated through scripting.

Lastly, SUMMON is designed to be fast. Libraries already exist for accessing OpenGL in python. However, python is relatively slow for real-time interaction with large visualizations (trees with 100,000 leaves, matrices with a million non-zeros, etc.). Therefore, all real-time interaction is handled with compiled native C++ code. Python is only executed in the construction and occasional interaction with the visualization. This arrangement provides the best of both worlds.

1.2 Features

Listed below is a short summary of the features offered in this version of SUMMON.

- Python module extension
- Fast OpenGL graphics
- Drawing arbitrary points, lines, polygons, text with python scripting
- Binding inputs (keyboard, mouse, hotspots) to any python function
- Separate threads for python and graphics (allows use of python prompt and responsive graphic at the same time)
- Transparently handles graphics event loop, scrolling, zooming, text layout (auto-clipping, scaling, alignment), detecting clicks, allowing you to focus on viewing your data
- SVG output (also GIF/PNG/JPG/etc with ImageMagick)
- cross-platform (windows, linux)

2 Installing SUMMON

The latest version of SUMMON can be obtained from http://people.csail.mit.edu/rasmus/summon/. Download the *.tar.gz archive and unzip it with the command:

```
tar zxvf summon-1.8.2.tar.gz
```

Before running or compiling SUMMON, the following libraries are required:

- python 2.4 (or greater)
- GL
- GLUT
- SDL (for threading)

2.1 Compiling SUMMON

SUMMON can be installed using the standard distutils (http://docs.python.org/inst/inst.html). For example, in the `summon-1.8.2` directory run:

```
python setup.py install
```

To install SUMMON in another location other than in `/usr` use:

```
python setup.py install --prefix=<another directory prefix>
```

2.2 Configuring SUMMON

SUMMON expects to find a configuration file called `summon_config.py` somewhere in the python path. Distutils installs a default module located in your python path. To customize SUMMON with your own key bindings and behavior, you can write your own `summon_config.py` file. Just be sure it appears in your python path somewhere *before* SUMMON default configuration file. Alternatively, you can create a configuration file `.summon_config` in your home directory. The configuration file is nothing more than a python script that calls the SUMMON function `set_binding` in order to initialize the default keyboard and mouse bindings.

3 Using SUMMON

SUMMON can be used as stand-alone program and as a module in a larger python program. The stand-alone version is installed in `PREFIX/bin/summon` and is called from the command line as follows:

```
usage:  summon [python script]
```

On execution, SUMMON opens an OpenGL window and evaluates any script that it is given in the python engine. After evaluation, the SUMMON prompt should appear which provides direct access to the python engine. Users should be familiar with the python language in order to use SUMMON.

The SUMMON prompt acts exactly like the python prompt except for the OpenGL window and the appearance of several automatically imported modules such as `summon`. All of the commands needed to interact with the visualization are within the `summon` module.

To learn how to use SUMMON, example scripts have been provided in the `summon/examples/` directory. Examples of full fledged visualizations, SUMMATRIX and SUMTREE, are also given in the `summon/bin/` directory. Their example input files are given in `summon/examples/summatrix/` and `summon/examples/sumtree/`, respectively.

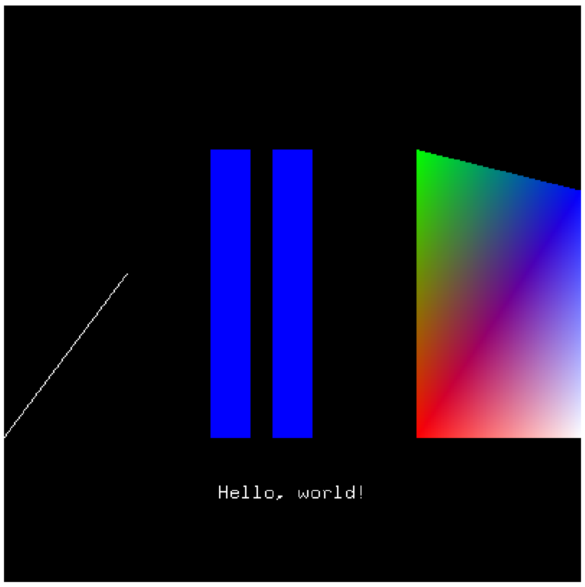
3.1 Example script

For an introduction to the basic commands of SUMMON, let us walk through the code of the first example. To begin, change into the `summon/examples/` directory and open up `01_basics.py` in a text editor. Also execute the example with following command.

```
python 01_basics.py
```

The visualization (Figure 1) should immediately appear in your OpenGL window. The following controls are available:

left mouse button	scroll
right mouse button	zoom (down: zoom-out, up: zoom-in)
Ctrl + right mouse button	zoom x-axis (down: zoom-out, up: zoom-in)
Shift + right mouse button	zoom y-axis (down: zoom-out, up: zoom-in)
arrow keys	scroll
Shift + arrow keys	scroll faster
Z	zoom in
z	zoom out
h	home (make all graphics visible)
Ctrl + l	toggle anti-aliasing
Ctrl + d	duplicate window
Ctrl + p	output SVG of the current view
Ctrl + Shift + p	output PNG of the current view
Ctrl + x	show/hide crosshair
q	close window
` + left click	display pop-up menu (note: *back-tic* is usually above *tab* key)



Hello, world!

Figure 1: Screenshot of SUMMON with `examples/01_basics.py`

In your text editor, the example `01_basics.py` should contain the following python code:

```
#!/usr/bin/env python-i
# SUMMON examples
# 01_basics.py - basic commands

# make summon commands available
from summon.core import *
import summon

# syntax of used summon functions
# add_group( <group> )   = adds a group of graphics to the screen
# group( <elements> )    = creates a group from several graphical elements
# lines( x1, y1, x2, y2, ... )  = an element that draws one or more lines
# quads( x1, y1, ..., x4, y4, ... )  = an element that draws one or more quadrilaterals
# color( <red>, <green>, <blue>, [alpha] ) = a primitive that specifies a color


# create a new window
win = summon.Window("01_basics")

# add a line from (0,0) to (30,40)
win.add_group(lines(0,0, 30,40))

# add two blue quadrilaterals inside a group
win.add_group(group(color(0, 0, 1),
                    quads(50,0, 50,70, 60,70, 60,0),
                    quads(65,0, 65,70, 75,70, 75,0)))

# add a multi-colored quad, where each vertex has it own color
win.add_group(quads(color(1,0,0), 100, 0,
                    color(0,1,0), 100, 70,
                    color(0,0,1), 140, 60,
                    color(1,1,1), 140, 0))


# add some text below everything else
win.add_group(text("Hello, world!",     # text to appear
                   0, -10, 140, -100,   # bounding box of text
                   "center", "top"))    # justification of text in bounding box

# center the "camera" so that all shapes are in view
win.home()
```

The first line of the script imports the SUMMON module `summon` and all of the basic SUMMON functions (`group`, `lines`, `color`, etc) from the `summon.core` module into the current environment. A new SUMMON graphics window is created using the `summon.Window` object.

All graphics are added and removed from the window in sets called *groups*. Groups provide a way to organize graphical elements into a hierarchy. The first graphical group added to the window is a line. The line is created with the `lines` function, which takes a series of numbers specifying the end-point coordinates for the line. The first two numbers specify the x and y coordinates of one end-point (0,0) and the last two specify the other end-point (30,40). Next, the line is added to the window using the `add_group` function.

The next part of the example adds two quadrilaterals to the window with the `quads` and `group` commands. The arguments to the `quads` function are similar to the `lines` function, except four vertices (8 numbers) are specified. In the example, two quadrilaterals are created and grouped together with the `group` function.

Note, both the `lines` and `quads` functions can draw multiple lines and quadrilaterals (hence their plural names) by supplying more coordinates as arguments.

The third group illustrates the use of color. Color is stateful, as in OpenGL, and all vertices that appear after a color object in a group will be affected. The `color` function creates a color object, which can appear within graphical elements such as `lines` and `quads` or directly inside a group. Since each vertex in this example quad has a different color, OpenGL will draw a quadrilateral that blends these colors.

Lastly, an example of text is shown. Once again the text is added to the window using the `add_group` function. The arguments to the text function specify the text to be displayed, a bounding box specified by two opposite vertices, and then zero or more justifications (”left”, ”right”, ”center”, ”top”, ”bottom”, ”middle”) that will affect how the text aligns within its bounding box. There are currently three types of text: `text` (bitmap), `text_scale` (stroke), `text_clip` (stroked text that clips). The bitmap text will clip if it cannot fit within its bounding box. This is very useful in cases where the user zooms out very far and no more space is available for the text to fit. See the example `10_text.py` for a better illustration of the different text constructs.

The final function in the script is `win.home()`, which causes the SUMMON window to scroll and zoom such that all graphics are visible. This is a very useful command for making sure that what you have drawn is visible in the window. The command can also be executed by pressing the ’h’ key within a SUMMON window. This key comes in handy when you ”lose sight” of the visualization.

This is only a simple example. See the remaining scripts in the `summon/examples/` for examples of SUMMON’s more powerful features.

3.2 Example visualizations: SUMMATRIX and SUMTREE

In the `summon/bin/` directory are two programs, `summatrix` and `sumtree` that use SUMMON to visualize large datasets. They are simply python scripts and so can be easily extended. In my own work, I have extended the tree visualization program to integrate more closely with biological data (executing CLUSTALW and MUSCLE on subtrees, displaying GO terms, etc.). Since these visualization programs are simply python scripts, others can easily overlay and integrate their own data.

Also in both visualizations the underling data is accessible through global python variables. That means if you have a very specific question like, ”How many genes in my subtree have a particular GO term?”, you can quickly write a few lines of python to walk the tree and answer the question yourself. It would be very difficult to anticipate all such questions during the development

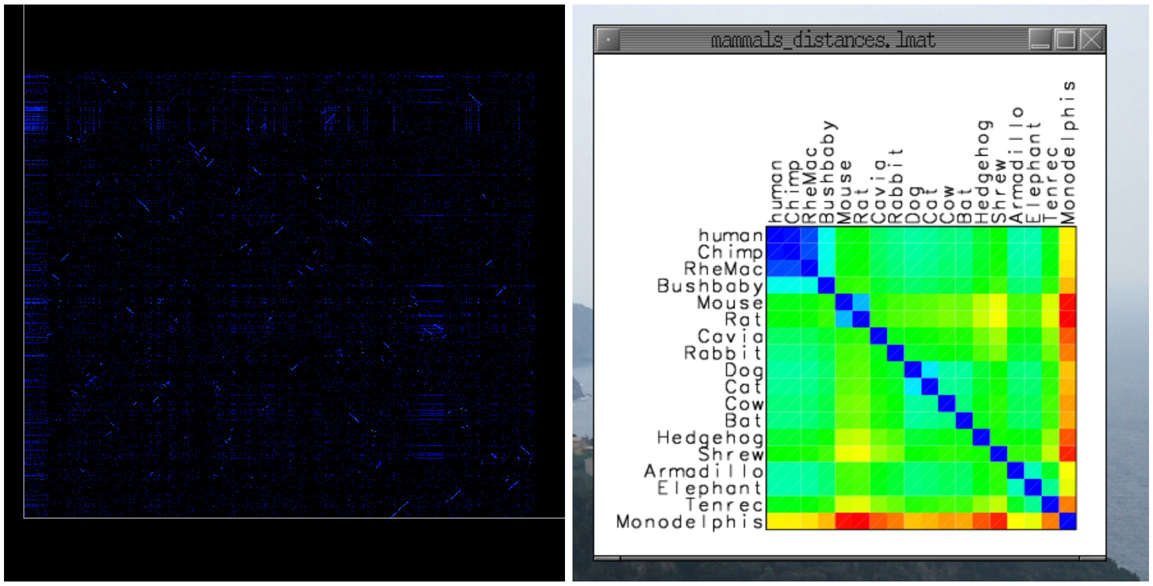


Figure 2: Example screenshots of SUMMATRIX on both sparse and dense matrices

of a non-scriptable visualization.

Example input files for both programs can be found under the `summon/examples/summatrix/` and `summon/examples/sumtree/` directories. Both programs will print their usage if run with no arguments. View/execute the `view_*.sh` scripts for examples of how to call `summatrix` and `sumtree`.

4 SUMMATRIX: large-scale sparse matrix visualizer

SUMMATRIX is a visualization for matrices (Figure 2) built with SUMMON. It can be executed either from the command-line with the supplied demo (`summatrix`) or instantiated as an object (`summon.matrix.MatrixViewer`) from within your own python program. SUMMATRIX also includes support for visualizing data clustering and other matrix permutations. To see all of SUMMATRIX's options run `summatrix` on the command-line with no options.

4.1 Matrix formats

`summatrix` visualizes dense and sparse matrices stored in a variety of formats.

- **dense format.** The *dense matrix format* is for small matrices with many non-zero entries. The first line of a *dense matrix file* (`*.mat`) contains two white-space delimited integers:

  ```
  nrows ncols
  ```

where `nrows` and `ncols` are the number of rows and columns in the matrix. The remaining `nrows` lines of the file specify each row of the matrix with `ncols` white-space delimited numbers (integer or float). For example:

```
val1 val2 val3 ...  valN
```

`summatrix` will load a *dense matrix* with the `-d` and `--dense` options.

See `examples/summatrix/clustering/data.mat` for an example.

- **compressed row format (sparse).** The *compressed row format* is for large matrices with only a relatively few non-zero entries. The first line of a *compressed row matrix file* (`*.mat`) contains three white-space delimited integers:

  ```
  nrows ncols nnz
  ```

  where `nrows` is the number of rows, `ncols` is the number of columns, and `nnz` is the total number of non-zero entries in the matrix. The remaining `nrows` lines of the file specify each row of the matrix in the following format:

  ```
  col1 val1 col2 val2 col3 val3 ...
  ```

  which indicates that for this *row* of the matrix we have

  ```
  matrix[row][col1] = val1
  matrix[row][col2] = val2
  matrix[row][col3] = val3
  ...
  ```

  Note that `colN` is an integer and 1-based (1st column is numbered 1, last column is numbered *ncols*) and `valN` can be a float or integer. Any entry of the matrix not specified is assumed to be zero.

  `summatrix` will load a *compressed row matrix* with the `-r` and `--rmat` options.

  See `examples/summatrix/dog_human.mat` for an example.

- **index format (sparse).** The *index format* is for large matrices with only a relatively few non-zero entries. The first line of an *index matrix file* (`*.imat`) contains three white-space delimited integers:

  ```
  nrows ncols nnz
  ```

  where `nrows` is the number of rows, `ncols` is the number of columns, and `nnz` is the total number of non-zero entries. The remaining `nnz` lines of the file specify each non-zero entry with the format:

```
row col val
```

which indicates that $matrix[row][col] = val$. Note that `row` and `col` are integers and 0-based (1st row and column are numbered 0, last row and column are numbered $nrows - 1$ and $ncols - 1$, respectively) and `val` can be a float or integer. Any entry of the matrix not specified is assumed to be zero.

`summatrix` will load an *index matrix* with the `-i` and `--imat` options.

See `examples/summatrix/human_mouse.imat` for an example.

- **labeled format (sparse).** This format is similar to *index format* except instead of specifying row and column by an integer, they are specified by a unique string (*label*). The total number of rows, columns, and non-zeros does not need to be specified. Instead each line of the file has the following format:

  ```
  rowlabel collabel val
  ```

  which indicates that $matrix[rowlabel][collabel] = val$. Note that `row` and `col` are strings and `val` can be a float or integer. Rows and columns are drawn in the order (left-to-right, top-to-bottom) that are first mentioned in the file. A different ordering can specified with the `--order` option and a file containing the labels in the desired order, one label per line. Any entry of the matrix not specified is assumed to be zero.

  `summatrix` will load a *labeled matrix* with the `-l` and `--lmat` options.

  See `examples/summatrix/mammals_distances.lmat` for an example.

4.2 Row and column label formats

SUMMATRIX can display row and column labels. They are specified with the `--rlab=`*filename* and `--clab=`*filename* command-line options. If your matrix is square and the row and column labels are the same, you can specify them with only one option using `--rclab=`*filename*. The label file format is simple a text file with one label per line in the order that the rows (or columns) appear in the matrix.

Labels will be used when the user clicks (middle-click or Ctrl+left click) on an entry in the matrix. Labels can also be displayed on the sides of the matrix with the `--showlabels` option or by pressing the "l" key during visualization. The `inline` option will draw the labels in the same window as the matrix. The `panels` option will draw the labels in neighboring windows, such that the labels are never out of view while zooming and scrolling.

4.3 Visualizing data clustering

SUMMATRIX allows easy visualization of clustering. However, you must use another third-party program to actually cluster your data. If you are looking for ideas, I recommend taking a look at the CLUTO clustering toolkit at `http://glaros.dtc.umn.edu/gkhome/views/cluto`.

4.3.1 Row and column permutation

Using such a clustering program you can visualize your clustering as a permutation of the rows (or columns) of your data matrix. SUMMATRIX will permute your matrix if you specify any of the options `--rperm=`*filename*, `--cperm=`*filename*, or `--rcperm=`*filename*. Where *filename* is a permutation file, which is a text file with *nrows* (or *ncols*) lines with one integer per line. The integer indicates which row (or column) should appear in this position. The integer is 0-based (1st row is 0, last is $nrows - 1$).

See `examples/summatrix/clustering/data.rperm` for an example of a row permutation file.

4.3.2 Row and column partitioning

SUMMATRIX can also draw horizontal and vertical lines that divide the matrix into sub-divisions that represent your clusters. This is done with the `--rpart=`*filename*, `--cpart=`*filename*, and `--rcpart=`*filename* options. The file *filename* should be in the *partition format*, which is a text file with *nrows* (or *ncols*) lines with one string per line. Each string indicates the cluster of the corresponding row (or column). For example, if your matrix is clustered by rows into three clusters called "ClusterA", "ClusterB", and "ClusterC", then those three strings should used in the partition file. Clusters can also be named with integers as in the supplied example.

See `examples/summatrix/clustering/data.rpart` for an example of a row partition file.

If a permutation file is specified the matrix will be permuted, and dividers will be drawn when ever two neighboring rows (or columns) are associated with different clusters. If no permutation file is given, the rows (or columns) will be sorted automatically by their cluster label.

5 SUMMON function reference

See `summon.html` for complete function reference.