SCIGMA

User's Manual and Reference

19th December 2022

License

Loki Library License

External Software Packages

Besides Loki, SCIGMA includes (subsets of) the following external packages:
AntTweakBar, GLFW, GLEW, zlib, libpng (all available under the zlib/libpng license),
ODESSA and LAPACK (available under the modified BSD license).
The AntTweakBar, GLFW and libpng sources have been slightly modified.

Contact

SCIGMA is a code in evolution. Especially, it has not yet been tested on a wide variety of machines, which makes it susceptible to the occasional OpenGL malfunction and missing shared library. Therefore, bug reports as well as comments and feedback to scigma.contact@gmail.com are very much appreciated!

Contents

1 Setting up SCIGMA

1.1 Prerequisites

SCIGMA runs under Windows, Linux and Mac OS X. It needs graphics hardware that supports at least OpenGL 2.1 (any desktop computer or laptop bought after 2008 should be fine). The Python programming language (version>=2.7) must be installed on the system. Python is part of OS X and comes preinstalled with most Linux flavors. If you do not have Python installed on Linux, you can always install it using the package manager of your distribution. If you are using Windows and don't have Python installed, have a look at `http://portablepython.com`. It's a conveniently self-contained version of Python that does not require admin privileges to install.

1.2 Obtaining the Python package

There are precompiled binary packages for Windows and Mac OS X available at `https://github.com/scigma/scigma/releases`.
Note that the 32-bit Windows binaries have been compiled with `sjlj` exception handling, while the 64-bit binaries have been compiled with `seh` exception handling. If you are going to use software compiled with MinGW and a different type of exception handling together with SCIGMA, you will run into trouble. In that case, you will have to build SCIGMA yourself, with your own version of MinGW.
For Linux, there is currently no binary package and you need to build SCIGMA yourself as well (see section 1.4 below).

1.3 Installation and first start

Create a folder for SCIGMA-related work (e.g `$HOME/myodes` and drop the `scigma` folder into it. Now, open a Python session in the folder you just created (i.e. `$HOME/myodes` should be the current working directory). At the Python prompt type:

```
>>> import scigma
```

After a few seconds, the SCIGMA graphics window should appear.
If you are using Portable Python in Windows, the fastest way to get things running is to drop the "scigma" folder into the "Portable Python ..." folder, double-click on `Python-Portable.exe` and type

```
>>> import scigma
```

in the console that opens.
This process can also be done by creating a runscigma.py file with at least the content:

```
import scigma
input("Press enter to exit.")
print("Exiting.")
```

and then running the file from the command line:

```
python runscigma.py
```

If this installation does not work, please build from source and see section 1.4 to install missing prerequisites (C++ compiler, Fortran compiler, CMake).
Note: An error message indicating that a .dylib library cannot be opened because the developer cannot be verified may appear. If that is the case, ensure that "Allow apps downloaded from" is toggled to "App Store + idenfitied developers" the in the Security & Privacy > General Panel. You may also need to go into the scigma-osx-X/scigma folder and control click on each of the ".dylib" files to manually open them. This will whitelist them.

1.4 Building SCIGMA from source

1.4.1 Obtaining the sources

You can either download the source archive at `https://github.com/scigma/scigma/releases` or, if you are familiar with git, clone the git repository directly from `https://github.com/scigma/scigma`.
In principle, you should be able to build SCIGMA with a combination of any working C++ compiler, any working Fortran compiler and CMake on your system (if CMake can deal with the compilers). However, the instructions below are the recommended way to build things, while other ways are untested.

1.4.2 Building under Linux

You will need to have `g++`, `gfortran` and `cmake` installed, as well as the X11 and OpenGL header packages (how these are called may vary depending on your distro; for Debian and its derivatives, like Ubuntu, install the `xorg-dev` and `libglu1-mesa-dev` packages). To build SCIGMA, open a terminal and change into the `build` directory of the top level folder. Now configure with

```
$ cmake ../source
```

and build with

```
$ make
```

Now, `build` should contain the shared library `libscigma.so`.

```
$ cp libscigma.so ../scigma
```

copies this library into the `scigma` subdirectory, which now contains the complete Python package. See section 1.3 on how to use the package.

1.4.3 Building under Windows

You will need both MinGW (with the g++, gfortran and msys packages) and CMake to compile SCIGMA.
If you do not have MinGW, download 'mingw-get-setup.exe' from `http://sourceforge.net/projects/mingw/files` and start the installation manager. Mark the packages `mingw32-base`, `mingw32-gcc-gfortran`, `mingw32-gcc-g++` and `msys-base` for installation and apply the changes. Finally, add `mingw/bin` and `mingw/msys/1.0/bin` to the system path.
If you do not have CMake, download and run the installer from `http://www.cmake.org/download`.
Now you are ready to build SCIGMA:
Start the CMake gui and specify the `source` and `build` subdirectories of the top level folder as source directory and binary directory, respectively. Push the `Configure`-button. When CMake asks you for the build type, choose `MSYS Makefiles`, with native compilers. After the configuration is done, press `Generate`. Now, the `build` folder contains the Makefiles and you are done with CMake.
Start the msys terminal (in the MinGW installation folder under `msys\1.0\bin\msys.bat`) and change into the build directory. If your build directory is `C:\scigma-0.9.1\build`, for example, use

```
$ cd /c/scigma-0.9.1/build
```

as command in the msys console. Finally, enter

```
$ make
```

to build the package. If everything runs smoothly, the shared library `libscigma.dll` will appear in the `build` folder. Copy this file into the `scigma` subdirectory with

```
$ cp libscigma.dll ../scigma
```

The `scigma` folder now contains the complete Python package and can be used as described in section 1.3.

1.4.4 Building under Mac OS X

The recommended way to obtain all three prerequisites (C++, Fortran and CMake) is to first install MacPorts (if you have another package manager like homebrew or fink already installed that should work as well). Installation instructions and the program itself can be found at `https://www.macports.org/install.php`. Once you are done with installing MacPorts, you will have `g++` already available as part of the Xcode Command Line Tools. You can then proceed to install `gfortran` with

```
$ sudo port install gfortran
```

as well as CMake with

```
$ sudo port install cmake
```

To build SCIGMA, open a terminal and change into the `build` directory of the top level folder of SCIGMA's source distribution. Configure with

```
$ cmake ../source
```

and build with

```
$ make
```

Now, `build` should contain the shared library `libscigma.dylib`.

```
$ cp libscigma.dylib ../scigma
```

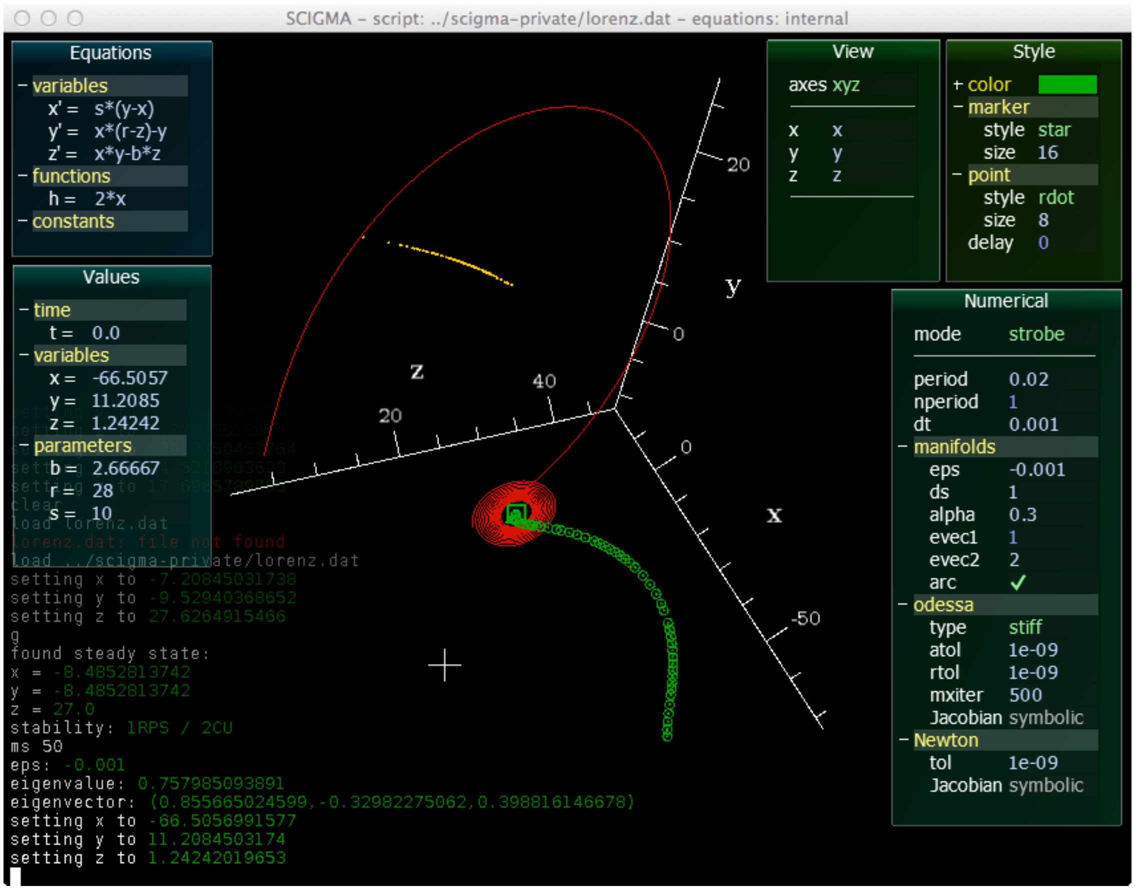
copies this library into the `scigma` subdirectory, which now contains the complete Python package. See section 1.3 on how to use the package.

1.4.5 Running

Navigate to scigma-osx folder in terminal and run

```
$ python3 runscigma.py
```

If that python file is missing, please see section 1.3 for creating it.


SCIGMA – script: ../scigma-private/lorenz.dat – equations: internal
z
y
x
20
40
0
-50

```
Equations
- variables
    x' =  s*(y-x)
    y' =  x*(r-z)-y
    z' =  x*y-b*z
- functions
    h =  2*x
- constants

Values
- time
    t =   0.0
- variables
    x =   -66.5057
    y =   11.2085
    z =   1.24242
- parameters
    b =   2.66667
    r =   28
    s =   10

View
axes xyz
x    x
y    y
z    z

Style
+ color
- marker
    style  star
    size   16
- point
    style  rdot
    size   8
  delay    0

Numerical
mode      strobe
period    0.02
nperiod   1
dt        0.001
- manifolds
    eps     -0.001
    ds      1
    alpha   0.3
    evec1   1
    evec2   2
    arc     ✓
- odessa
    type    stiff
    atol    1e-09
    rtol    1e-09
    mxiter  500
    Jacobian symbolic
- Newton
    tol     1e-09
    Jacobian symbolic
```

```
clear
load lorenz.dat
lorenz.dat: file not found
load ../scigma-private/lorenz.dat
setting x to -7.20845031738
setting y to -9.52940368652
setting z to 27.6264915466
g
found steady state:
x = -8.4852813742
y = -8.4852813742
z = 27.0
stability: 1RPS / 2CU
ms 50
eps: -0.001
eigenvalue: 0.757985093891
eigenvector: (0.855665024599,-0.32982275062,0.398816146678)
setting x to -66.5056991577
setting y to 11.2084503174
setting z to 1.24242019653
```

Figure 1: Screenshot of a typical SCIGMA session

2 Overview

SCIGMA is a Python package that allows you to

- specify a set of ordinary differential equations
- visualize the phase and/or parameter space
- select initial conditions and parameters interactively
- plot trajectories
- find stationary states
- examine the stability of those states
- plot invariant manifolds of those states
- do all of the above for iterated maps, as well as Poincaré maps and stroboscopic maps of ordinary differential equations
- automate these tasks with a simple scripting language or directly from Python
- read the results back into your own Python programs

Figure 1 shows SCIGMA's user interface. The largest part of the window is taken up by a diagram of the phase space and/or parameter space. On the sides, there are some panels that give you information about the current equation system, as well as all kinds of numerical and graphic settings. In the bottom left corner, there is a console at which you can enter equations and commands, as well as adjust and query the settings.

The first thing you have to do when working with SCIGMA is to let the program know which equations you want to look at. There are several ways to do that, which are all described in section 3. Once you have done that, you will be ready to produce some nice graphics. Most of SCIGMA's work is done by a handful of commands, namely

- plot, which plots orbits of maps and trajectories of differential equations,
- guess, which finds stationary states of trajectories and differential equations,
- mstable/munstable, which trace out invariant manifolds,
- and cont, which performs single-parameter continuation of a steady state.

These commands, together with some more bookkeeping and visual commands are described in section 4. Some other commands that you might find useful include

- fit, which scales the currently visible viewing area/volume such that objects fit onto the screen,
- clear, which deletes all objects in the graphics window,
- and reset, which, on top of clearing the graphics window, also deletes the current equation system.

Of course there are a lot of options and settings that modify the behavior of SCIGMA's commands, the most important one being the mode setting. mode governs whether SCIGMA considers the system as a discrete map or as a differential equation, as well as whether it turns a set of differential equations into a stroboscopic map or Poincaré map. This and other settings are described in section 5.
Finally, section 6 describes how to navigate the viewing area (or viewing volume in three-dimensional projection) with the mouse, as well as how to interactively select objects and pick initial conditions for the numerical commands.

3 Specifying the equations

In SCIGMA, there are three ways of encoding equations, namely

- directly at the command line in the graphics window, or from a script file (best suited for the explorative investigation of smaller systems);
- by specifying a set of equations in a Python script - Python is more expressive (e.g. loops and arrays are available), but you will have to specify the Jacobian yourself.
- as a precompiled shared library (fastest, but less convenient than both of the above).

The next three sections describe all of these possibilities.

Before we dive into the specifics, let us discuss what all three approaches have in common: SCIGMA distinguishes between three types of user defined symbols in equations: *variables, parameters* and *functions.* The number of variables determines the dimension of the phase space for an ODE or a map. Parameters are values that are constant in time and may be used for bifurcation analysis. Functions are quantities that depend on variables, or parameters, or both, which are not strictly necessary to describe the problem, but might be convenient as auxiliary functions or otherwise interesting. For illustration, look at these equations for an harmonic oscillator:

$$\begin{aligned} \dot{x} &= -y \\ \dot{y} &= \omega^2 x \\ \omega &= \sqrt{\frac{k}{m}} \end{aligned} \tag{1}$$

Here, x and y are variables, k and m are parameters and ω is a (constant) function.
Regardless of how the equations are specified, you can always set the value of a variable or a parameter like this

```
x=1
k=2
```

at the SCIGMA console, or in the `Values` panel.

3.1 At the SCIGMA console

Variables

To define a variable for an ODE system, all you have to do is provide its time derivative.
`x' = x-1`
defines a variable x with $\dot{x} = x - 1$. The initial condition / current value for x is set like this:
`x = 0.5`
For an iterated map, the syntax to define a variable by providing the iterative equation, is the same:
`n' = n*2+1`
How your equations are interpreted depends on whether the mode option of the program is set to `'ode', 'strobe' or 'Poincare'` (interpretation as ODE) or to `'map'` (interpretation as iterated map).

Parameters

If you assign a value to a new symbol
`a = 2` ,
this defines a new parameter. Parameters are also created implicitly when a symbol appears in an expression for the first time (see below).

supported mathematical operations
`+, -, *, /, **, sqrt, exp, ln, log10,`
`sin, cos, tan, asin, acos, atan, atan2,`
`sinh, cosh, tanh, asinh, acosh, atanh,`
`pow, sigmoid, pulse, abs, sign, step, mod`

Table 1: List of mathematical operations understood by SCIGMA's internal parser. `**` stands for exponentiation, as in FORTRAN or Python. `sigmoid` takes two arguments x and β and is defined as $s = (1+\exp(-x/\beta))^{-1}$; `pulse` also takes two arguments: $p = \exp(x/\beta)/(\beta\exp(x/\beta)+\beta)$, which is the derivative of the sigmoid function. Therefore, the area under the pulse is always equal to 1. The `step` function behaves like its GLSL equivalent, taking two parameters as well, where the second parameter is the position of the step. `mod` is equivalent to the C function `fmod`

Functions

If the right hand side of your equation contains at least one symbol like `x`, `n`, and `a`, you create a function rather than a parameter:
`xSquare = x**2`
Note that `xSquare` keeps track of the changes in `x` and updates itself accordingly, not only if `x` changes its value, but even if it is completely redefined; as another example, consider this definition for the time derivative of `y`:
`y' = x**2 + a**b`
Here, `a` is an already known parameter with value `2`, while `b` is implicitly initialized as a new parameter with value `0`, which happens whenever you enter an equation with a symbol that is not yet defined. `x` is of course a variable since our first definition. For the moment, we have $\dot{y} = x^2 + 2^0$. Note that instead of the last line, you could also have written
`y' = xSquare + a**b .`
When you redefine a variable, parameter or function, this affects all other functions depending on it. For example after
`b = x+y ,`
the time derivative of `y` becomes $\dot{y} = x^2 + 2^{x+y}$. Because previous definitions are updated, the order in which you enter a set of equations does not matter; SCIGMA takes care that the equations remain consistent.

Evaluation operator

If you do want to extract the current value of a function instead of the function itself, use the $-operator:
`$(a**2)`
prints out `4`, for example. Do NOT use this operator on the right hand of equations, this will erratically change the value of the left hand side function.

Builtin mathematical functions

There is a number of builtin mathematical operations understood by SCIGMA's parser that can be used in expressions. Please see table 1 for the full list.

Deleting variables, parameters and functions

To delete all information pertaining to a symbol `x`, use
`!x .`
This deletes function definitions, map and ODE definitions and the symbol itself (unless there are symbols that depend on `x`; in this case, these have to be deleted first). You can also do this
`!x' ,`
which takes the variable `x` out of the dynamical system, but keeps it available as parameter.

command	**effect**
`x' = ` *value* | *expression*	define variable `x`
`x = ` *expression*	define function `x`
`x = ` *value*	define parameter `x` / set value of variable `x`
`$`*expression*	replace expression by its current value
`!x`	delete symbol `x`
`!x'`	turn variable `x` into parameter

Table 2: List of different command types understood by SCIGMA's internal parser. *value* is any expression that evaluates to a constant, i.e. either a number, or a builtin function of just numbers like `sin(3.14)**2`. *expression* is any expression that depends on at least one other defined symbol like `sin(2*a)` or `f*g+1`.

Example

Table 2 shows a summary of the available commands of SCIGMA's internal equation parser. With these commands, we are now ready to specify equations (1) from the beginning of this chapter:

```
x'=-y
y'=omega**2*x
omega=sqrt(k/m)
```

Before observing the dynamics, we must also assign sensible values to the parameters `k` and `m`, for example

```
k=1
m=2
```

Now the system can be used with SCIGMA. Note again that the order in which the five equations above are entered does not matter.

Instead of entering the equations step by step at runtime you can also collect them in a file and use the load command. Either way, the equations will also be mirrored in the `Equations` panel, which can be used to modify the equations as well.

3.2 In a Python script

You can load a set of equations from a Python file at runtime with the equations command. SCIGMA scans this script for the following symbols:

`v_names` is a list of variable names:

```
v_names = ['x','y','z']
```

`v_values` is a list of initial values for the variables:

```
v_values = [0,0,0]
```

If `v_values` is defined, it must have the same length as `v_names`. If it is not defined the variables are not initialized on equation loading.

`p_names` is a list of parameter names:

```
p_names = ['a','b','c']
```

If `p_names` is not defined, SCIGMA assumes that there are no parameters.

`p_values` is a list of initial values for the parameters:

```
p_values = [1,2,3]
```

If `p_values` is defined, it must have the same length as `p_names`. If it is not defined the parameters are not initialized on equation loading.

`f` is a function defining the right hand side of the dynamical system. `f` can have one of these signatures:

```
def f(x,xdot):
    ...                 # for autonomous systems without parameters
def f(x,p,xdot):
    ...                 # for autonomous systems with parameters
def f(t,x,xdot):
    ...                 # for non-autonomous systems without parameters
def f(t,x,p,xdot):
    ...                 # for non-autonomous systems with parameters
```

SCIGMA decides on the type of system by looking at the number of arguments and the name of the first argument. Therefore, the first argument must always be `t` for non-autonomous systems. SCIGMA will pass the variables in the first argument (autonomous systems) or second argument (non-autonomous systems) and the parameters in the second or third argument. The right hand side must be written to the third (autonomous systems) or the fourth argument (non-autonomous systems). Note that `x`, `p` and `xdot` are raw pointers to memory, so you are responsible to not access them beyond their limits, which are given by the length of the `v_names` and `p_names` lists, respectively (the example below shows how the arguments are used).

`dfdx` is a function defining the jacobian of the dynamical system. If `dfdx` is defined, it must have the same signature as `f`, for example

```
def dfdx(x,jac):
    ...                 # for autonomous systems without parameters
```

SCIGMA expects the Jacobian in column major ordering, i.e. if there are `N` variables, the first `N` entries of `jac` contain the partial derivatives of the right hand side with respect to the first variable, the next `N` entries the partial derivatives with respect to the second variable and so on (compare also the example below). You do not need to provide zero elements of the Jacobian.

`f_names` is a list of additional function names:

```
f_names = ['f','g','h']
```

If defined, these are some additional functions of interest which are computed and stored along with the variables. They are evaluated by calling

`func`, which, must again have the same signature as `f`:

```
def func(x,values):
    ...                 # for autonomous systems without parameters
```

`func` must be defined if `f_names` is defined and has at least one entry.

Example

The Python script describing the example equations (1) would look like this:

```
import math

v_names=['x','y']
p_names=['k','m']
p_values=[1,2]
f_names=['omega']

def f(x,p,xdot):
    xdot[0]=-x[1]
    xdot[1]=p[0]/p[1]*x[0]

def dfdx(x,p,jac):
    jac[1]=p[0]/p[1]
    jac[2]=-1

def func(x,p,values):
    values[0]=math.sqrt(p[0]/p[1])
```

Please note that you cannot use the additional functions inside the body of `f` or `dfdx` (this is because internally, `func` is not evaluated for every single call of `f` and `dfdx`).

3.3 In a Shared Library

. . . coming soon . . .

4 Commands

In the list of commands below, mandatory arguments are written like `<this>`, optional arguments are written like `[this]`. If there are two or more distinct possibilities for an argument, they will be denoted `<like|this> [or|like|this]`. Most commands have abbreviations, which are given at the end of the description of the command.

4.1 General

`equations ['internal'|filename]`
: specifies the source of the dynamical equations. You can provide the equations either internally at the command line (this is the standard setting, to which you can return by giving `'internal'` as argument), or in a Python script file specified by `filename`. See sections 3.1 and 3.2 for details on the format of internal equations and of the Python file, respectively. If neither `'internal'`, nor any `filename` is given as an argument, the command opens a file dialog. This command may be abbreviated by `eq`.

`load [filename]`
: loads and runs a the script file specified in `filename`. If `filename` is not given, the command opens a file dialog. This command may be abbreviated by `l`.

`select <name>`
: makes the object specified by `name` the currently selected object. This command may be abbreviated by `sel`.

`delete <name>`
: deletes the orbit, manifold or special point specified by `name` from memory, including all of the data. This command may be abbreviated by `del`.

`clear`
: Deletes all orbits, manifolds and special points. This command may be abbreviated by `cl`.

`reset`
: Deletes all orbits, manifolds and special points, deletes the current set of equations and resets the view to a two-dimensional plot of x vs y between $(-1, -1)$ and $(1, 1)$. This command may be abbreviated by `res`.

`quit`
: closes the current window and frees the associated resources (i.e. makes them available to Python's garbage collector). This command may be abbreviated by `q`.

4.2 Numerical

You will notice that some of the commands below come in two versions, namely with and without an appended asterisk (`*`). For these commands, the version with the asterisk shows *all* iterates if nperiod is not 1, while the command without the asterisk just shows the *nperiod-th* iterates.

`plot [n] [name] / plot* [n] [name]`
: takes the current state as initial condition and plots `n` steps of the resulting orbit or trajectory. The orbit will be assigned `name` as identifier for further reference. If `n` is not given, only one step will be taken. A negative `n` will reverse the direction of the iteration/integration. If `name` is not given, a unique name of the form `'trN'` will be generated.
The behavior of this command depends on the current setting of the mode option:
If mode is `map`, the program iterates the map and plots every nperiod-th point.
If mode is `ode`, the program integrates the differential equations for `n` time steps dt and plots the trajectory.
If mode is `strobe`, the program will plot every nperiod-th point of the stroboscopic map of the differential equations with a time interval period (this is effectively the same behavior as for `ode`, with period×nperiod as time step instead of dt).

If mode is `Poincare`, the program constructs a Poincare map for the intersection of the secvar=secval plane, iterates it and plots every nperiod-th point. maxtime determines how far the program should integrate before giving up on finding an intersection. secdir determines whether the value of secvar should be increasing or decreasing at the intersection.
The appearance of the resulting trajectory in depends on the current values of color, marker.style, marker.size, point.style, point.size and delay. This command may be abbreviated by `p / p*`.

`guess [name] / guess* [name]`
takes the current state as starting point and tries to find a stationary state or the current map or ode by Newton iteration. If a stationary state is found, it will be assigned `name` as identifier for further reference. If `name` is not given, a unique name of the form `'fpN'` ('fixed point', for differential equations) or `'ppN'` ('periodic point', for maps) will be generated. If mode is not `ode`, the program will look for a stationary state of the nperiod-th iterate of the map. Stationary states of differential equations will be marked with a circle, if they are unstable, and with circles with a dot in the center, if they are stable. Stationary states of maps will be marked with an empty square, if they are unstable, and with a square with a dot in the center, if they are stable. The further appearance depends on the current values of color, and marker.size. This command may be abbreviated by `g / g*`.

`evals [name]`
prints the eigenvalues of the object specified by `name`. If `name` is not given, the currently selected object is used.

`evecs [name]`
prints the eigenvectors of the object specified by `name`. If `name` is not given, the currently selected object is used.

`mstable [n] [origin] [name] / mstable* [n] [origin] [name]`
takes a stationary state (the 'origin') previously found with guess and plots `n` steps of the 1-dimensional stable manifold of the origin along an stable eigenvector, which is determined by the value of evec1. `name` will be assigned to the manifold as identifier for further reference. If `n` is not given, just the initial segment is created. If `origin` is not given, the currently selected object is used. If `name` is not given, a unique name of the form `'mfN'` will be generated. If mode is `ode`, one step will correspond to an integration of ±dt. Otherwise, the manifold for the nperiod-th iterate of the map will be constructed, with an arc length step of (roughly) arc. The accuracy of the manifold stepping algorithm for maps can be adjusted with alpha. The appearance of the manifold depends on the current values of color, marker.style, marker.size, point.style, point.size and delay. This command may be abbreviated by `ms / ms*`. Note that both sides (both directions of the eigenvector) of the manifold will be drawn. To draw only one side, please use the commands `ms1 / ms1*`, soon to be deprecated, along with the setting eps to change directions and distance.

`munstable [n] [origin] [name] / munstable* [n] [origin] [name]`
works analog to `mstable`, except on unstable manifolds. This command may be abbreviated by `mu / mu*`. To draw just one side of the manifold, use `mu1 / mu1*` (soon to be deprecated) along with the setting eps to change directions.

`rtime [name]`
If `name` denotes an orbit or stationary state of a stroboscopic map or Poincare map, this prints the return time. If `name` is not given, `rtime` prints the return time of the currently selected object. This command may be abbreviated by `rt`.

`circle <diameter> [n]`
takes the current position as center and places `n` initial conditions equally spaced on the circle with the specified diameter. `diameter` is given in units of the currently visible portion of phase space. If `n` is not given, 100 is used. This command may be abbreviated by `cir`.

`fill <n>`
inserts `n` equally distributed points between consecutive pairs of currently selected points

(that means you need to have at least two points selected to use `fill`). The resulting set of points becomes the new initial condition. The coordinates of the new points are linear interpolations of the coordinates of the old points.

`cont <n> <name>`
: continues the steady state by changing the parameter indicated by `name` for `n` steps of the step size setting `ds` using pseudo-arclength continuation. Associated settings include `a0`, `a1`, `ds`, `dsmin`, `dsmax`, `epsl`, `epsu`, `epss`, `rl0`, and `rl1`; all which parallel the AUTO continuation program settings. The advanced setting list is not described here, but can be seen via the AUTO panel. After each continuation run, SCIGMA will auto-select the end point of the run. Specifying two names separated by a “,” will conduct 2-parameter continuation.

`cycle <n> <name>`
: continues a limit cycle by changing the parameter indicated by `name` for `n` steps of the step size setting `ds` using pseudo-arclength continuation. Reports the maximum of the limit cycle. To use, first select a Hopf bifurcation point.

4.3 View and style

`hide <name>`
: hides the orbit, manifold or special point specified by `name` if it is currently on display, but keeps it in memory.

`show <name>`
: displays the orbit, manifold or special point specified by `name` if it is currently hidden.

`2d`
: changes to two-dimensional projection.

`3d`
: changes to three-dimensional projection.

`xrange <min> <max>`
: adjusts the range of the x dimension. `min` must be smaller than `max`.

`yrange <min> <max>`
: adjusts the range of the y dimension. `min` must be smaller than `max`.

`zrange <min> <max>`
: adjusts the range of the z dimension. `min` must be smaller than `max`.

`crange <min> <max>`
: adjusts the range of the color dimension. `min` must be smaller than `max`.

`fit`
: adjusts the viewing volume to include all visible objects. This command may be abbreviated by `f`.

5 Settings

Settings can be changed on the command line with the syntax given below. If only the name of an option is entered, SCIGMA prints its value on the console. Instead of using the command line, you can also access all settings using the `Numerical`, `View` and `Style` panels.

5.1 Numerical

`mode <'m[ap]'|'o[de]'|'s[trobe]'|'P[oincare]'>`
: specifies which type of dynamical problem is investigated. If `mode` is set to `map`, the equations are interpreted as a discrete map, otherwise they are interpreted as system of ordinary differential equations. The behavior of most numerical commands, as well as the availability and relevance of other numerical options depends on the value of `mode`. The default value is `ode`.

`period <value>`
: sets the return time for stroboscopic maps (mode is `strobe`). If equations are internal, `value` can be any expression that evaluates to a constant (*value* in the sense of table 2), for example `$(2*pi/omega)`. If equations are not internal, `value` must either be a number, or the value of a symbol defined in the equation system, prefixed with the $-operator, for example `$T`. The default value is `1.0`.

`nperiod <value>`
: sets the number of periods for the iteration of maps. `value` must be an integer. All algorithms (basically plot, guess and the manifold commands) acting on maps (regular, stroboscopic and Poincaré), will use the `nperiod`-th iterate of the map. The default value is `1`.

`dt <value>`
: sets the time step for the integrator. This is relevant for all settings of mode except the `map` setting. `dt` must be positive real number. If you want to plot backwards in time, use a negative step size. The default value is `0.001`.

`secvar <varname>`
: sets the variable of the Poincaré section. `varname` must be a variable that is part of the current equation system. The default value is `'x'`.

`secval <value>`
: sets the variable value of the Poincaré section. `value` must be a positive real number, the default value is `0.0`.

`secdir <'+'|'-'>`
: sets the direction in which crossings of Poincaré section are detected. If `'+'`, crossings with increasing value of secvar are detected, if `'-'`, crossings with decreasing value of secvar are detected. The default value is `'+'`.

`maxtime <value>`
: determines, until which value of t SCIGMA integrates to find a section of the Poincaré plane, before throwing the towel. `value` must be a positive real number, the default value is `100.0`.

`eps <value>`
: sets the distance of the initial segment of invariant manifolds from a stationary state when using the (to be deprecated commands) mu1, ms1. `value` must be a non-zero real number. By changing the sign, you can plot the two parts of one-dimensional manifolds. The default value is `0.0001`.

`ds <value>`
: sets the arc length step for invariant manifolds of maps (not used if mode is `ode`) or for pseudo-arclength continuation. If used for manifolds, `value` must be a positive real number; if used with continuation, the direction of computation may be reversed by making `ds` negative. The default value is `0.01`.

`dsmin <value>`
: sets the minimum pseudo-arclength stepsize used in continuation if `IADS`, which controls the frequency of adaptation, is set greater than 0 (default is 1). `value` must be a positive real number. The default value is `1E-06`.

`dsmax <value>`
: sets the maximum pseudo-arclength stepsize used in continuation if `IADS`, which controls the frequency of adaptation, is set greater than 0 (default is 1). `value` must be a positive real number. The default value is `1`.

`arc <value>`
: sets the approximate distance between points in phase space on a manifold created by the commands mstable and munstable.

`rl0 <value>`
: sets the lower bound on the continuation parameter. The default value is `-1E300`.

`rl1 <value>`
: sets the upper bound on the continuation parameter. The default value is `1E300`.

`epsl <value>`
: sets the relative convergence criterion for parameters in the Newton/Chord equations. The default value is `1E-07`.

`epsu <value>`
: sets the relative convergence criterion for solution components in the Newton/Chord equations. The default value is `1E-07`.

`epss <value>`
: sets the relative convergence criterion for the arclength component in the Newton/Chord equations. The default value is `1E-06`. It is recommended that this value is 100 to 1000 times greater than `epsl`, `epsu`.

`alpha <value>`
: sets the maximum accepted angle for consecutive line segments on invariant manifolds of maps (not used if mode is `ode`). `value` must be a positive angle in rad. Internally, SCIGMA uses an adapted step size algorithm to approximate invariant manifolds of maps. If two consecutive line segments have an angle of more than `alpha`, the internal step size is reduced. The default value is `0.3`.

`evec1 <value>`
: sets the first eigenvector that is used when computing invariant manifolds. `value` must be a positive integer, no larger then the number of dimensions in the current equations system. The eigenvalues of any stationary state are sorted according to stability (i.e. with ascending real value for ODEs, with ascending modulus for maps). For a stable (unstable) manifold the `evec1`-th most stable (unstable) eigenvalue is selected. The default value is `1`, for the selection of the most stable (unstable) eigenvalue.

`Newton.tol <value>`
: sets the tolerance of the Newton-Raphson algorithm that is used for finding stationary states, and also to converge onto Poincaré planes. `value` must be a positive real number, the default is `1e-9`.

`atol <value>`
: sets the absolute tolerance of the ODESSA integration algorithm. `value` must be a positive real number, the default is `1e-9`.

`rtol <value>`
: sets the relative tolerance of the ODESSA integration algorithm. `value` must be a positive real number, the default is `1e-9`.

`type <'non-stiff'|'stiff'>` sets whether the ODESSA integrator should use an algorithm for non-stiff or stiff equations. The default value is `'stiff'`.

`mxiter <value>`
: sets the number of internal steps before the ODESSA integrator bails out if it does not converge. `value` must be a positive integer, the default is `500`. switches between

5.2 View and style

`axes <'xy','xyz','xyc','xyzc'>`
: sets the whether the current view is 2D (`'xy'`), 2D with a color map (`'xyc'`), 3D (`'xyz'`) or 3D with a color map (`'xyzc'`). The default is `'xy'`.

`color <name|r g b [a]|index>`
: sets the drawing color for new objects. There are three ways of specifying the color:

- using the name, e.g. `color green`; available names are `'red'`, `'green'`, `'blue'`, `'yellow'`, `'pink'`, `'lime'`, `'azure'`, `'orange'`, `'brown'`, `'forest'`, `'navy'`, `'teal'`, `'rose'`, `'aqua'`, `'sky'`, `'beige'`, `'black'`, `'dark_gray'`, `'gray'`, `'light_gray'`, `'white'`, `'cyan'` and `'magenta'`.
- using rgb (and potentially alpha) values, e.g. `color 1 0 1 [1]` for magenta; all values must be real numbers between `0.0` and `1.0`, alpha values are optional.
- using an index, e.g. `color 6` for orange; the index must be an integer from `0-22` which points into the list of colors given above.

The default value is `color red` = `color 0.75 0 0` = `color 0`.

`delay <value>`
: sets the artificial plotting delay. `value` must be a non-negative real number. If not zero, SCIGMA waits for `delay` seconds before plotting the next step of an orbit or manifold that is currently computed. This can be used to better observe the evolution of a state in phase space over time. Also, if marker.style is not `none`, the current point of the orbit or trajectory is marked with a small graphic, which is easier to track. Plotting with delay only works correctly, however, if a single step can be computed faster than `delay`, otherwise the speed of the integration/manifold plotting algorithm determines how fast things are plotted. The default value is `0.0`.

`marker.style`
: sets the marker style for plotting with delay. Possible values are `'dot'`, `'plus'`, `'ring'`, `'rdot'`, `'rplus'`, `'rcross'`, `'quad'`, `'qdot'`, `'qplus'`, `'qcross'`, `'hash'`, `'star'` and `'none'`. The default value is `'star'`.

`marker.size`
: sets the marker size for plotting with delay and for marking stationary states. `value` must be a positive real number (values larger than `64.0` may not be supported by the OpenGL implementation). The default value is `16.0`.

`point.style`
: sets the point style for plotting orbits, trajectories and one-dimensional manifolds. Possible values are `'dot'`, `'plus'`, `'ring'`, `'rdot'`, `'rplus'`, `'rcross'`, `'quad'`, `'qdot'`, `'qplus'`, `'qcross'`, `'hash'`, `'star'` and `'none'`. The default value is `'none'` (in this case, the points are connected by line segments).

`point.size`
: sets the point size for plotting orbits, trajectories and one-dimensional manifolds. `value` must be a positive real number (values larger than `64.0` may not be supported by the OpenGL implementation). The default value is `8.0`.

6 Mouse navigation

6.1 Rotating

If the current projection is three-dimensional (see axes option and the 2d/3d commands), you can rotate the viewing volume by pressing the left mouse button somewhere over an empty region of the graphics window and dragging the mouse.

6.2 Shifting

You can shift the viewing area/volume along the horizontal and vertical axes of the screen by pressing the left mouse button somewhere over an empty region of the graphics window and dragging the mouse while also pressing the shift key. Alternatively, if you let the cursor hover over one of the axes, such that it is highlighted, you can shift along this axes by pressing the left mouse button and dragging the mouse.

6.3 Zooming

You can change the scale of the viewing area/volume by scrolling with the mouse wheel, while the cursor is over an empty region of the graphics window. This zooms in and out equally for all axes. Alternatively, if you let the cursor hover over one of the axes, such that it is highlighted, you can scale just this axis by scrolling with the mouse wheel.

6.4 Picking points

If the projection is two-dimensional, double clicking somewhere in the graphics window will make the two coordinates of this point current. You will also see a white crosshair shape marking the point. In three-dimensional projection, the first double click will define a line of sight. If you rotate the viewing volume, you will see the line of sight as it runs through the viewing volume. On it, there is a white crosshair shape that follows the movement of the mouse. If you are satisfied with the position of the crosshair, double click again, and the three coordinates of the point will be made current. Also, the point is marked by the crosshair.
Both in two and three dimensions you can pick more than one point. To do so, press the control key while performing the steps above multiple times. As a result, you will get a number of crosshair shapes, marking your set of newly picked initial conditions. Commands like plot will now act simultaneously on all of these.

6.5 Picking objects

Whenever the mouse hovers over an object like a trajectory in the graphics window, the object is highlighted. If you now double click, this will select the object under the cursor as current object. The current object is used by commands like munstable, if no object is explicitly given as an argument. Also, the coordinates of the object (in case of orbits and trajectories: the last point) are made current.

7 The Python layer

See 3.2.
. . . more coming soon . . .

8 The C++ layer

. . . coming soon . . .