System Dynamics Modeling with se-lib User's Guide

October 12, 2023

0.0.1 Table of Contents

1 Introduction

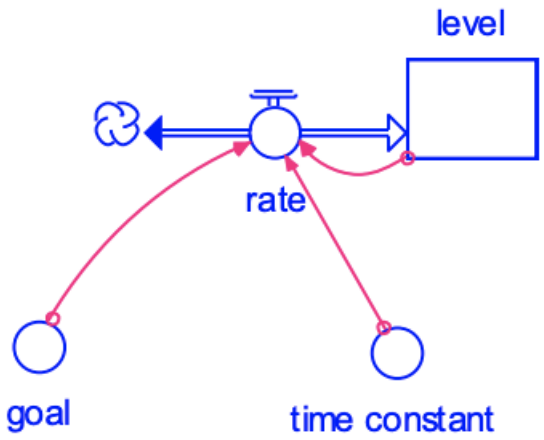
The se-lib library provides system dynamics modeling and simulation functions. It is built on top of the PySD library and internally uses the XMILE file format standard for system dynamics.

A system model is described by defining the standard elements for stocks (levels), flows (rates), and auxiliary constants or equations. Names of model elements and their equations are specified as character strings.

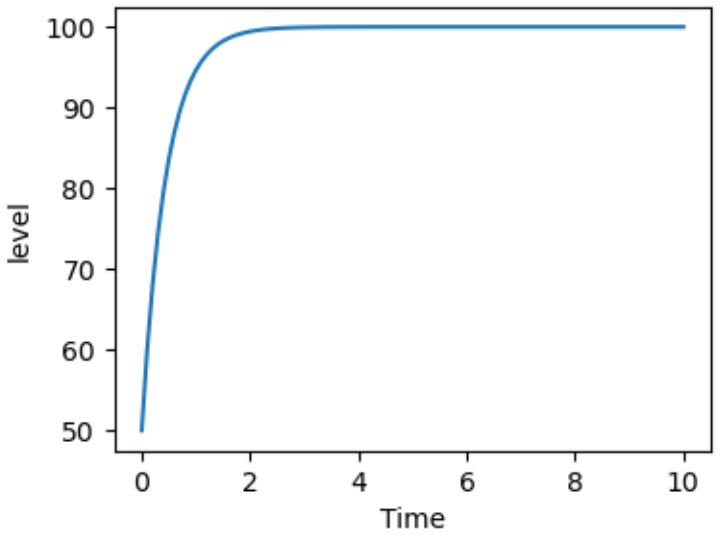
Utility functions are available for equation formulation, data collection and displaying outut. Detailed function references and examples are available online at http://se-lib.org/function_reference.html#system-dynamics. Examples are shown of the basic features next.

1.1 Basic Modeling Functions

The following example implements a simple first order delay structure. Before using any of the functions the se-lib library must be imported. First a model must be initialized. A stock is defined with an initial level, the flow and auxiliary variable for the delay. The model is run over time and a graph is plotted of the resulting level.

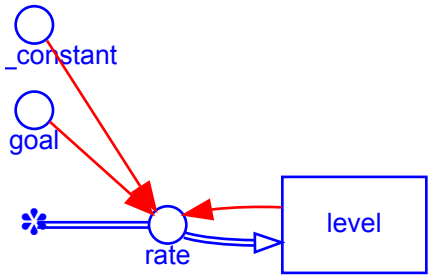


[3]:
```
# import all functions
from selib import *

# negative feedback
init_sd_model(start=0, stop=10, dt=.1)
add_stock("level", 50, inflows=["rate"])
add_auxiliary("time_constant", .5)
add_auxiliary("goal", 100)
add_flow("rate", "(goal - level) / time_constant")
run_model()

plot_graph('level')
```



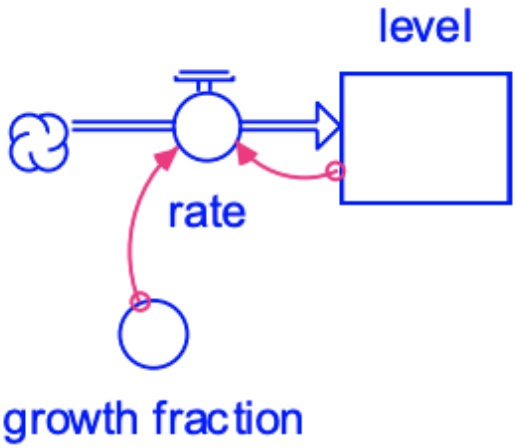
[28]:
```
# negative feedback
init_sd_model(start=0, stop=10, dt=.1)
add_stock("level", 50, inflows=["rate"])
add_auxiliary("time_constant", .5)
add_auxiliary("goal", 100)
add_flow("rate", "(goal - level) / time_constant", ["goal", "level",␣
 ↪"time_constant"])
run_model()

draw_model_diagram()
#plot_graph('level')
```

[28]:



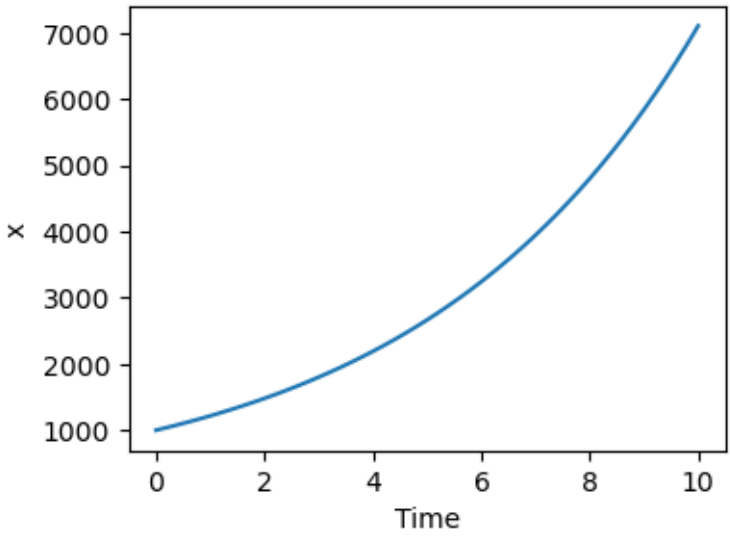
1.2 Exponential Growth

A model of exponential growth per the following structure is shown.



[4]:
```
# exponential growth

init_sd_model(start=0, stop=10, dt=.2)

add_stock("x", 1000, inflows=["dx"])
```

```
add_flow("dx", "x*growth_factor")
add_auxiliary("growth_factor", .2)

run_model()
plot_graph('x')
```



1.3 Displaying Output

The `run_model()` function will execute a simulation and return a Pandas dataframe of the output. It can displayed by the following statements demonstrated in the program below for a Rayleigh curve.

```
output = run_model()
output
```

[17]:
```
# Rayleigh curve staffing model

init_sd_model(start=0, stop=6, dt=.5)

add_stock("cumulative_effort", 0, inflows=["effort_rate"])
add_flow("effort_rate", "learning_function * (estimated_total_effort -␣
 ↪cumulative_effort)")
add_auxiliary("learning_function", "manpower_buildup_parameter * time")
add_auxiliary("manpower_buildup_parameter", .5)
add_auxiliary("estimated_total_effort", 15)

output, model_dict = run_model()
output
```

[17]:

	INITIAL TIME	FINAL TIME	TIME STEP	SAVEPER	effort_rate	learning_function	manpower_buildup_parameter	estimated_total_effort	cumulative_effort
0.0	0	6	0.5	0.5	0.000000	0.00	0.5	15	0.000000
0.5	0	6	0.5	0.5	3.750000	0.25	0.5	15	0.000000
1.0	0	6	0.5	0.5	6.562500	0.50	0.5	15	1.875000
1.5	0	6	0.5	0.5	7.382812	0.75	0.5	15	5.156250
2.0	0	6	0.5	0.5	6.152344	1.00	0.5	15	8.847656
2.5	0	6	0.5	0.5	3.845215	1.25	0.5	15	11.923828
3.0	0	6	0.5	0.5	1.730347	1.50	0.5	15	13.846436
3.5	0	6	0.5	0.5	0.504684	1.75	0.5	15	14.711609
4.0	0	6	0.5	0.5	0.072098	2.00	0.5	15	14.963951
4.5	0	6	0.5	0.5	0.000000	2.25	0.5	15	15.000000
5.0	0	6	0.5	0.5	0.000000	2.50	0.5	15	15.000000
5.5	0	6	0.5	0.5	0.000000	2.75	0.5	15	15.000000
6.0	0	6	0.5	0.5	0.000000	3.00	0.5	15	15.000000

Specific variables can accessed using a dictionary key notation per the following:

[19]:
```
output['learning_function']
```

[19]:
```
0.0    0.00
0.5    0.25
1.0    0.50
1.5    0.75
2.0    1.00
2.5    1.25
3.0    1.50
3.5    1.75
4.0    2.00
4.5    2.25
5.0    2.50
5.5    2.75
6.0    3.00
Name: learning_function, dtype: float64
```

The value of a variable at a given time can be accessed using time as the second key index:

[20]:
```
output['learning_function'][2]
```

[20]:
```
1.0
```

Multiple variables can be provided in a list:

[21]:
```
output[['learning_function', 'cumulative_effort']]
```

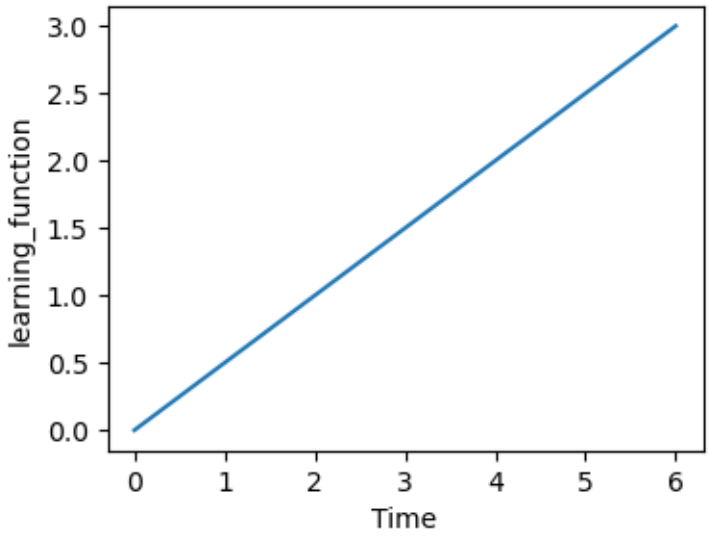
[21]:

	learning_function	cumulative_effort
0.0	0.00	0.000000
0.5	0.25	0.000000
1.0	0.50	1.875000
1.5	0.75	5.156250
2.0	1.00	8.847656
2.5	1.25	11.923828
3.0	1.50	13.846436
3.5	1.75	14.711609
4.0	2.00	14.963951
4.5	2.25	15.000000
5.0	2.50	15.000000
5.5	2.75	15.000000
6.0	3.00	15.000000

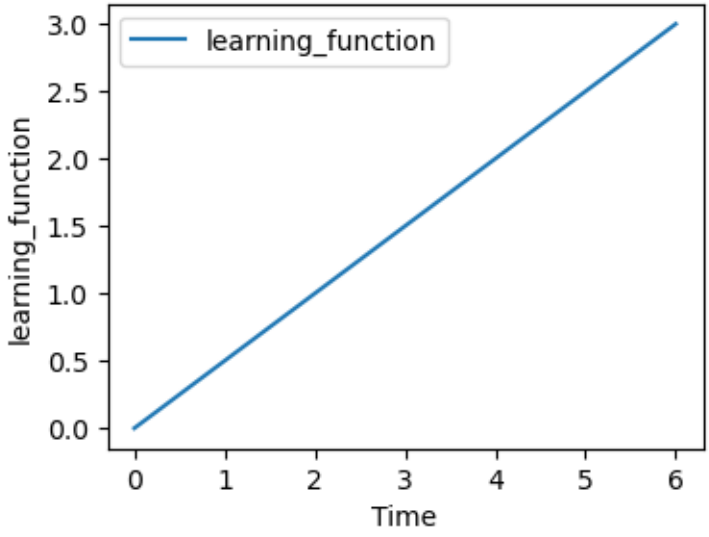
1.4 Graphs

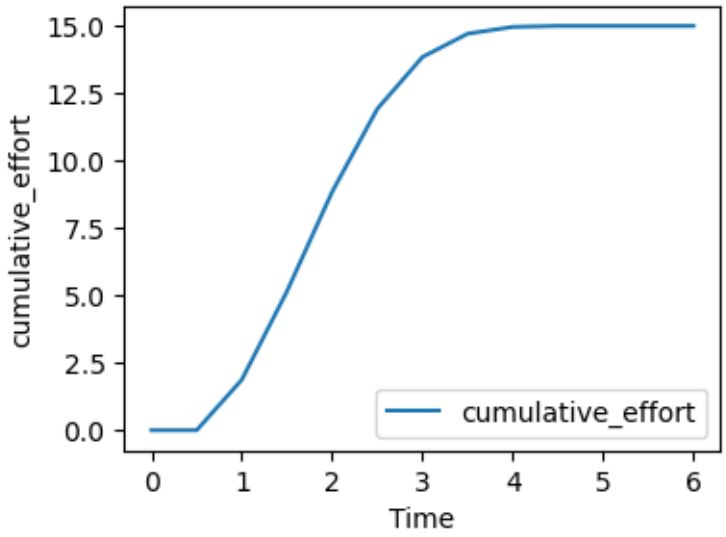
Graphs are specified in the `plot_graph` functions. It will accept a single variable to plot, or a comma separated variables printing each separately, or a list in brackets where all the variable are plotted on a single scale axis. Combinations are also accceptable.

[22]:
```
plot_graph('learning_function')
```

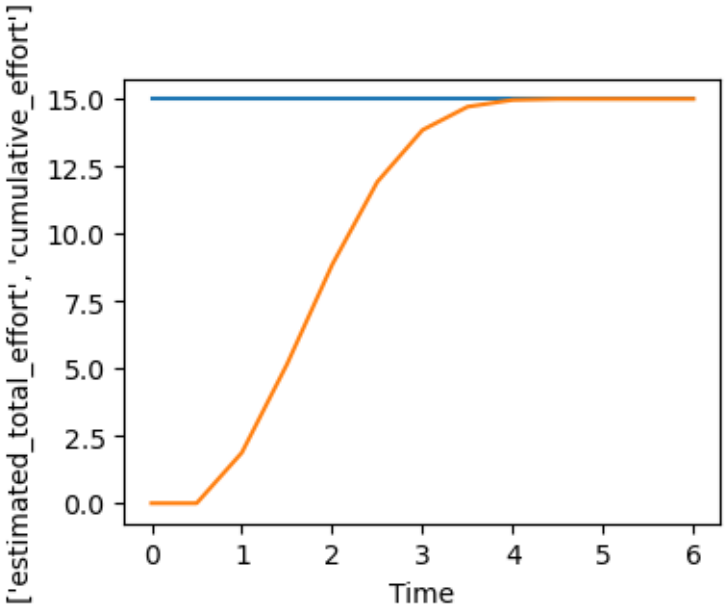


[23]:
```
plot_graph('learning_function', 'cumulative_effort')
```





[24]:
```
plot_graph(['estimated_total_effort', 'cumulative_effort'])
```



1.5 Random Number Functions

In equations for auxiliaries and rates, the random number functions supported are called as if the following import has occurred `from random import random, random.uniform`. Thus they are called with `random()` for a uniformly distributed number between 0 and 1 or

`random.uniform(min, max)` for a uniformly distributed number between the min and max. For xmile format compatibility, the functions `RANDOM_0_1` and `RANDOM_UNIFORM(min, max)` are equivalent and also acceptable.

There is difference in how the random functions are treated in se-lib because they take on random values each time step. The default xmile usage draws a single random value at the beginning of each run.

[25]:
```
init_sd_model(start=0, stop=3, dt=.5)
add_auxiliary("random_parameter", "20*random()")
run_model()
```

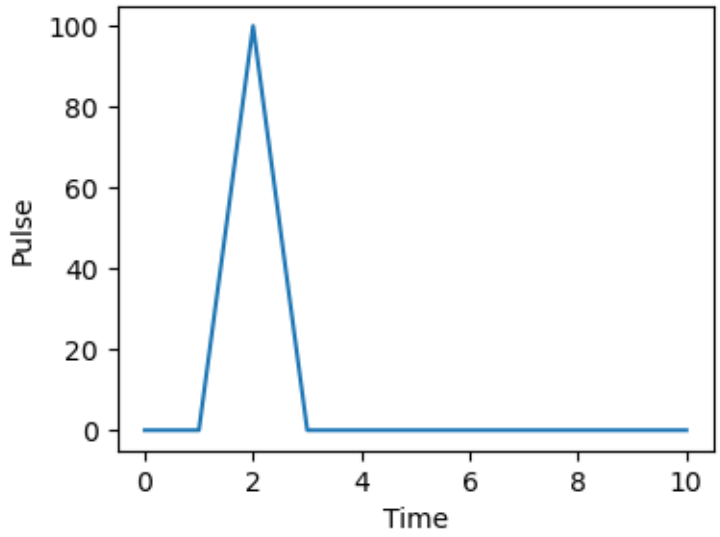
[25]:
```
(     INITIAL TIME  FINAL TIME  TIME STEP  SAVEPER  random_parameter
 0.0             0           3        0.5      0.5          0.087466
 0.5             0           3        0.5      0.5         11.174929
 1.0             0           3        0.5      0.5         13.506344
 1.5             0           3        0.5      0.5          4.342211
 2.0             0           3        0.5      0.5          5.449781
 2.5             0           3        0.5      0.5          7.323554
 3.0             0           3        0.5      0.5         17.220789,
 {'stocks': {},
  'flows': {},
  'auxiliaries': {'random_parameter': {'equation': '20*(GET_TIME_VALUE(0,0,0) +
.00001) / (GET_TIME_VALUE(0,0,0) + .00001) * RANDOM_0_1()',
    'inputs': []}}})
```

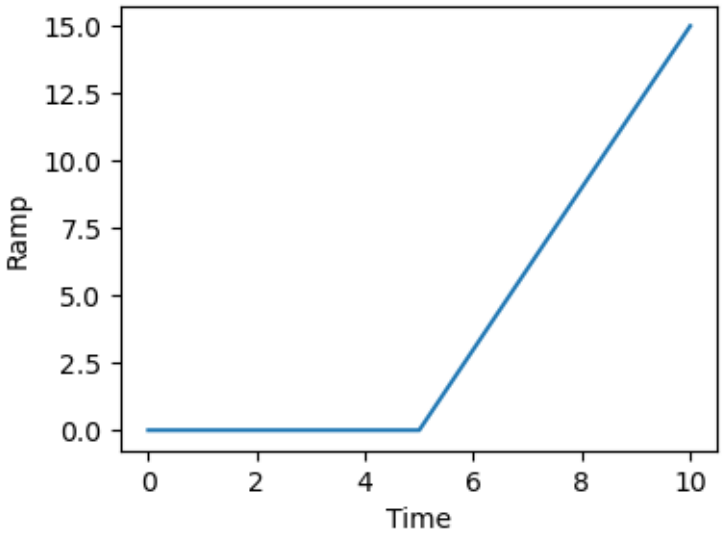
1.6 Utility Functions

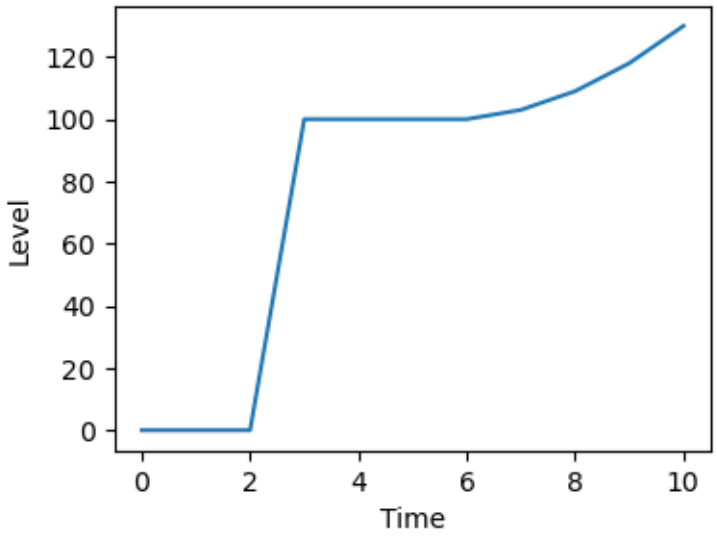
1.6.1 Test Functions

Standard test functions are available for pulse, ramp and step inputs as shown below. The full set of functions available in PySD can be used but have not all been tested.

[26]:
```
init_sd_model(start=0, stop=10, dt=1)

add_stock("Level", 0, inflows=["Pulse", "Ramp"])
add_flow("Pulse", "pulse(100, 2)") # pulse of 100 at time 2
add_flow("Ramp", "ramp(3, 5) ") # ramp with slope 3 at time 5

run_model()
plot_graph('Pulse')
plot_graph('Ramp')
plot_graph('Level')
```


100
80
60
40
20
0
Pulse
0
2
4
6
8
10
Time


15.0
12.5
10.0
7.5
5.0
2.5
0.0
Ramp
0
2
4
6
8
10
Time



1.7 Advanced Usage

All of the features of PySD can be used in conjunction with se-lib functions. See PySD usage documentation.

1.8 Appendix A - Function Reference

System Dynamics¶

init_sd_model¶

selib.init_sd_model(start, stop, dt)¶

Instantiates a system dynamics model for simulation

add_stock¶

selib.add_stock(name, initial, inflows=[], outflows=[])¶

Adds a stock to the model

Parameters:

name (str) – The name of the stock

initial (float) – Initial value of stock at start of simulation

inflows (list of float) – The names of the inflows to the stock

outflows (list of float) – The names of the outflows to the stock

add_flow¶

selib.add_flow(name, equation, inputs=[])¶

Adds a flow to the model

Parameters:

name (str) – The name of the flow

equation (str) – Equation for the flow using other named model variables

inputs (list) – Optional list of variable input names used to draw model diagram

add_auxiliary¶

selib.add_auxiliary(name, equation, inputs=[])¶

Adds auxiliary equation or constant to the model

Parameters:

name (str) – The name of the auxiliary

equation (str) – Equation for the auxiliary using other named model variables

inputs (list) – Optional list of variable input names used to draw model diagram

plot_graph¶

selib.plot_graph(*outputs)¶

displays matplotlib graph for each model variable

Parameters:

variables (str or list) – comma separated variable name(s) or lists of variable names to plot on single graphs

Return type:

matplotlib graph

save_graph¶

selib.save_graph(*outputs, filename='graph.png')¶

save graph to file

Parameters:

variables (variable name or list of variable names to plot on graph) –

filename (file name with format extension) –

run_model¶

selib.run_model(verbose=True)¶

Executes the current model

Returns:

If continuous, returns 1) Pandas dataframe containing run outputs for each variable each timestep and 2) model dictionary.

If discrete, returns 1) network dictionary with run statistics and 2) entity run data

set_logical_run_time¶

selib.set_logical_run_time(condition)¶

Enables a run time to be measured based on a logical condition for when the simulation should be run (like a while statement). The logical end time will be available from the 'get_logical_end_time()' function in lieu of the fixed end time for a simulation.

get_logical_end_time¶

selib.get_logical_end_time()¶

Returns the logical end time as specified in a previous 'set_logical_run_time()' function call, in lieu of the fixed end time for a simulation.

Returns:

logical_end_time – end time when the 'set_logical_run_time()" condition expires

Return type:

float

draw_model_diagram¶

selib.draw_model_diagram(filename=None, format='svg')¶

Draw a diagram of the current model.

Parameters:

filename (string, optional) – A filename for the output not including a filename extension. The extension will specified by the format parameter.

format (string, optional) – The file format of the graphic output. Note that bitmap formats (png, bmp, or jpeg) will not be as sharp as the default svg vector format and most particularly when magnified.

Returns:

g – Save the graph source code to file, and open the rendered result in its default viewing application. se-lib calls the Graphviz API for this.

Return type:

graph object view

[ ]: