Tellurium scanner proposal

Xieergai Jiang

June 2022

Introduction

Tellurium scanner is a collection of functions that extends the existing Tellurium package by simplifying the process of parameter scanning.

It is often common practice to perform analysis on variables' value range to examine their effect on the model in question. The scanner is a medium-level approach allowing for largest possible flexibility while also being straightforward for beginner use.

Our medium-level approach aims at providing more freedom compared to GUI-based high-level solutions such as COPASI. While requiring next to zero coding experience, such high abstraction methods either lack ways of implementing unaccounted for use cases or do not have straightforward ways of their implementation.

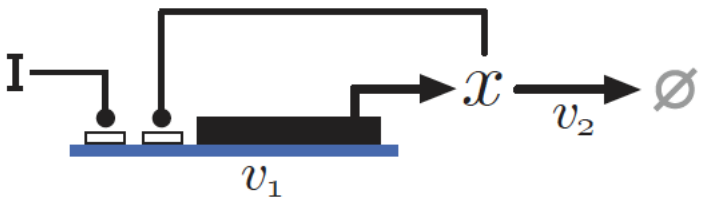
Furthermore, our approach alleviates the amount of time and coding knowledge when using computing platforms like MATLAB. Although being the most flexible approach, it demands more attention to programming (i.e.: debugging and algorithm design) and so is inaccessible either due to time constraints or a different specialization of the user.

This module extends the capabilities of Tellurium API and utilizes its simulation and file reading functionality.

Use case example

It is often the case that one uses parameter scans to examine model behaviour with respect to the parameters of said model. As a simple example, consider an autoregulator model from Control Theory for Bioengineers CH 23.1.

See the tutorial section for a more detailed walkthrough.



A simple examination of the initial concentration of x could look like so:

```
from scanner import load

model = load("pathToFile_or_antimonyString")
model.setParameters('x')
model.setTarget('x')
model.setScanRange([0,0.5,1,1.5], type = 'custom')
model.scan()
```

Here, the `setParameters` function is used to indicate what value will be varied between the scan iterations. When it comes to species, the id of it refers to its initial concentration. For more information, see the tutorial and the API below.

The `setTarget` function is used to specify whatever we are trying to examine within the model, be it concentration of a species or something else. For a more detailed explanation please see the API as well as the tutorial below.

With the `scan` command, the library will execute a parameter scan changing the initial concentration of `x` according to the values in `setScanRange` utilizing the simulation capabilities of the RoadRunner API.

Viewing results from scans

After the scan, the user may extract target data either directly from class attributes or using a function:

```
rawData = model.results
targetData = model.getData()
```

With the code above, `rawData` will be a list of RoadRunner simulation outputs, and `targetData` will be an n-dimentional array of whatever the target was set to(where n is number of scan parameters).

Depending on the type of the tracked target, this module contains some visualization capabilities:

If the target was a time series for each simulation, one may generate a plot of the time series:

```
model.plotSimple("targetID")
```

For 2-D scans with a single value per simulation as a target one can create a heatmap:

```
model.plotHeatmap("parameter1", "parameter2")
```

Alternatively, one can iterate over a given parameter with multiple graphs

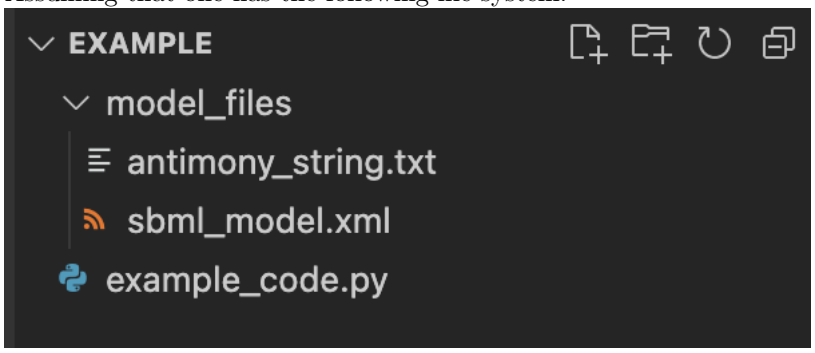
```
model.plotOver("parameterId")
```

Proposed Tutorial

The following will be the expansion on the earlier mentioned example.

Basic case

Assuming that one has the following file system:



And that the antimony string has the following setup:

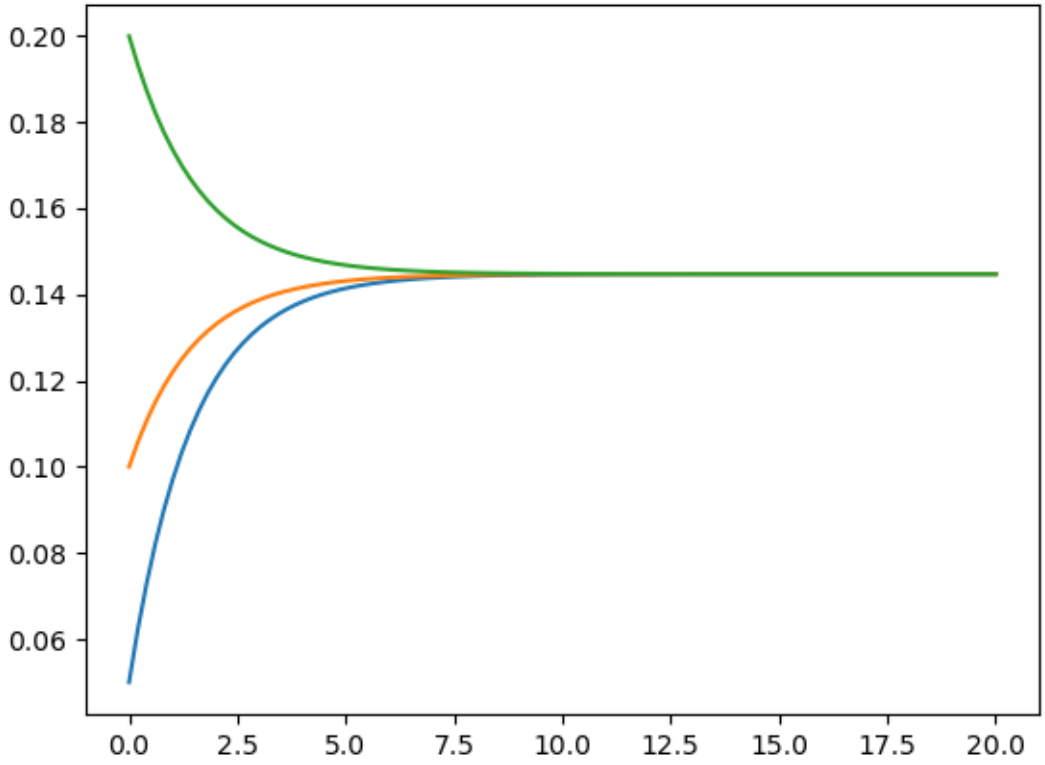
```
"""
model feedback
    compartment C1;
    C1 = 1.0;
    species X
    const X0

    J1: X0 -> X; B + k1*X^4/(k2+X^4);
    J2: X -> ; k3*X;

    k1 = .9
    k2 = .3
    k3 = .7
```

```
    B = 0.1
    X = 0.1

end
"""
```

The example_code.py could look like this:

```
from scanner import load

model = load("model_files/antimony_string.txt")

parameter_list = model.listParameters()
model.setParameters(parameter_list[0])
model.scan()
model.plotSimple("X")
```

The resulting plot will look like this:



The above code loads in the model from a specified file, selects the initial concentration of species X described in the Antimony string as the parameter, and then runs the parameter scan of time series simulations.

With default settings, a linear scan is performed over [half initial, initial, double initial] values with all of the species concentrations tracked; only concentration of X is tracked in our case since it is the only one in the system.

Specifying range

The module has preset functions for: linear, logarithmic, and exponential range definitions. Alternatively, one can define their own range of values to use (see *Custom scan range*).

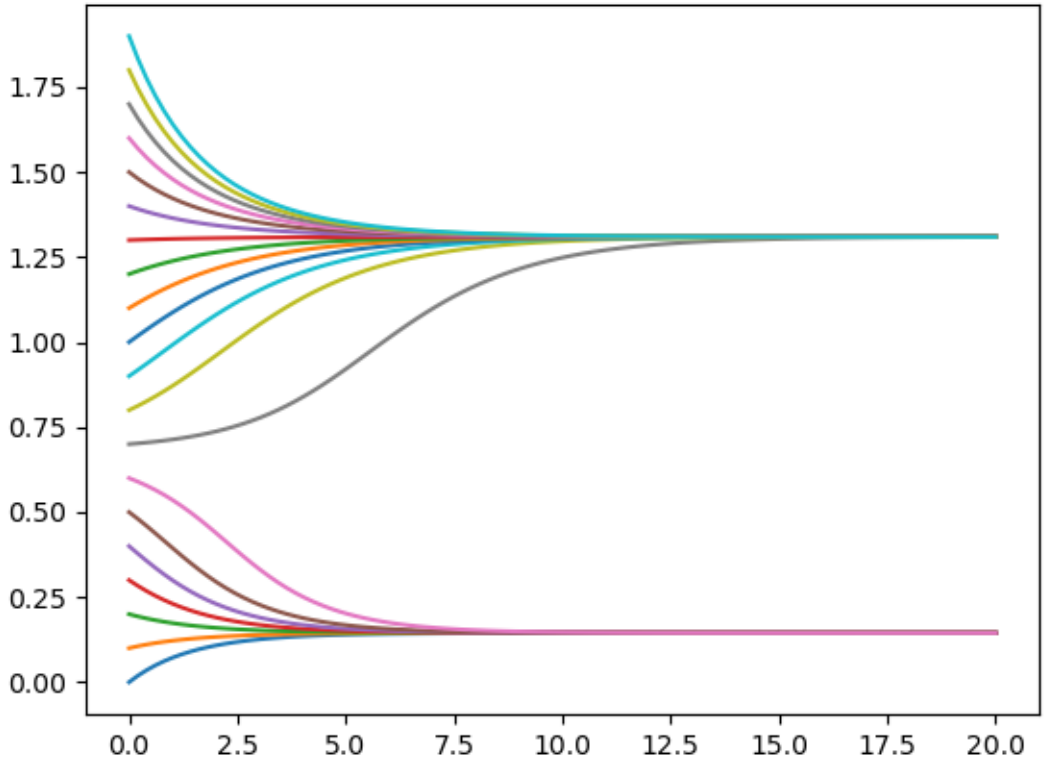
The preset functions generate a list of values ($rangeValues$) and use them as multiplicands for the initial value of a given parameter: $Val_{new} = Val_{initial} * rangeValue$

Returning to the earlier example, we can see that the system contains more than one steady state if we specify a broader parameter space

```
from scanner import load

model = load("model_files/antimony_string.txt")

model.setParameters(['X'])
model.setScanRange(0,19,20, type='linear')

model.scan()
model.plotSimple('X')
```

Here, the model will perform a scan over the parameters with the linear range[0,19] and 20 points in total. Meaning that the initial value will be multiplied by 0, 1, 2 and so on.

Resulting plot will now look like so:



Specifying target

In our module, we define **target** as any range/value that we want to track during the parameter scan. In the example above, our target was the output of the roadrunner simulation, which happened to only have the concentration of species X.

The module can accept the id of the species to track as specified within SBML/Antimony. Alternatively, one may specify the target with a function (see *Custom Target*).

```
from scanner import load

model = load("model_files/antimony_string.txt")
model.setTarget('ID of species')
```

The result of this configuration would be time series of the species with said ID.

Custom scan range and target

As mentioned above, the user may customize both the parameter scan range as well as the target to track.

Custom scan range

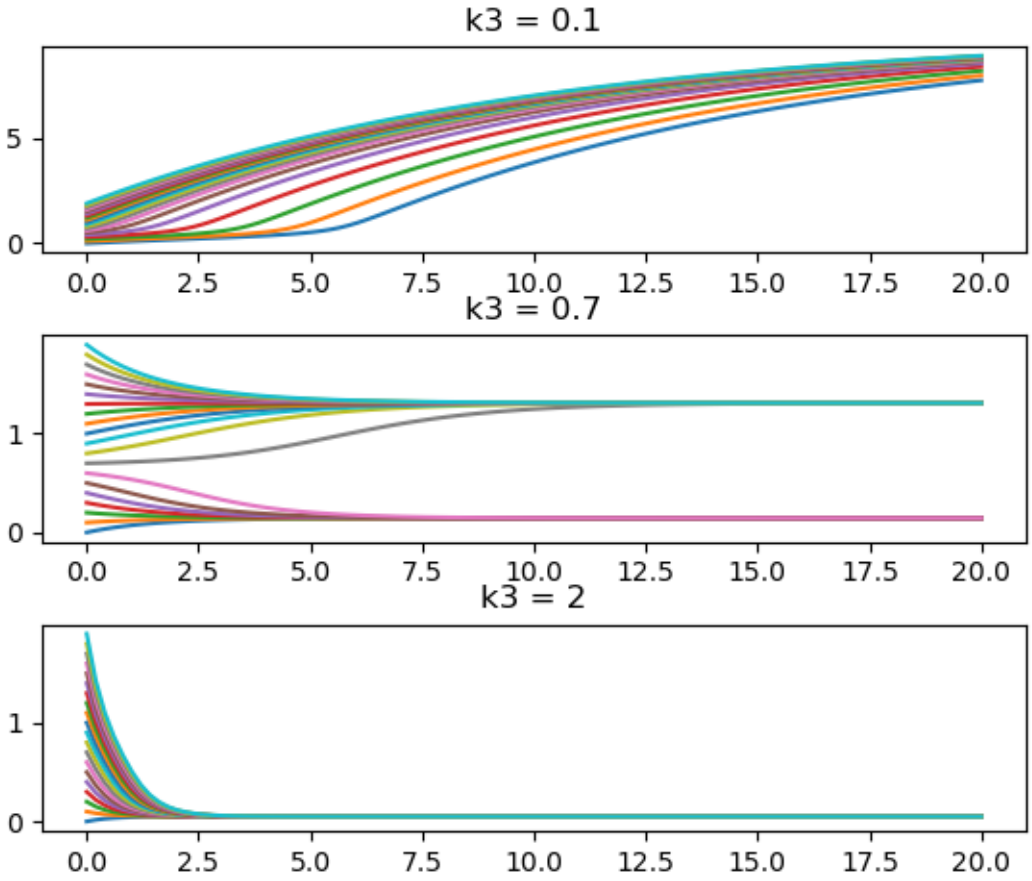
The user may provide their own range to be used for a given parameter to scan.

Continuing our example, we can show that the parameter *k3* (that accounts for species degradation) affects the steady state of the system. Let us see 3 cases of *k3* : 0.1, 0.7, and 2 with the same initial concentration scan of X as before.

Note that custom ranges are accepted as is and are passed directly to change the model parameter during the scan.

```
from scanner import load
model = load("model_files/antimony_string.txt")

k3ScanRange = [.1, .7, 2]
model.setParameters(['k3', 'X'])
model.setScanRange(k3ScanRange, 0, 19, 20,
                type=['custom', 'linear'], uniform = False)
model.scan()
model.plotOver('k3')
```

As a result, we will get the following plot:



Custom target function

The user may specify what information to extract and store using their own function.

The function needs to accept the output of roadrunner simulation, i.e. the NamedArray of data.

Suppose instead of seeing the time series, we are only interested in what steady states our simulations produce. One way of implementing that would be to create a custom function that extracts the last value of the time series:

```
def customTarget(sim_output):
    '''
    User defined function.
    Used to interact with scanner.Model
    class as Model.scanTarget attribute
```

```
    Attributes
    ----------
    sim_output: NamedArray
        Array output of roadrunner simulation

    Returns
    -------
    custom: float
        In this example, returns the final value of
        the species "X" after the simulation
    '''
    # keep in mind that roadrunner
    # differentiates parameter IDs ('EX1') and
    # concentrations of a species ('[EX1]').
    # Simulation Named Arrays use concentrations of species

    EX1 = sim_output['[X]']
    lastValue = EX1[-1]

    return lastValue

model.setTarget(custom_target)
```

Proposed tests

To ensure the proper use of the module, as well as its proper functioning, the following tests are recommended:

- For model
  - Only a list of strings is accepted for setParameters()
  - Check inputted strings for setParameters() against listParameters() output
  - If using a custom traget function, check if the function accepts simulation output of the correct form
- For scanning
  - check that during scan, the initial condition case matches expected values
  - check that first run succeeds, then initialize run

API

def load

Used for loading in models/older simulations. Returns an instance of a model object

Attributes

- `mod : str or class`
  - A path to the supported file
  - Or an antimony string
  - Or an instance of roadrunner from the Tellurium package:

    `tellurium.roadrunner.extended_roadrunner.ExtendedRoadRunner`

Returns

- `Model : class`
  - Primary class to interact witht the library
  - Inherits from:

    `tellurium.roadrunner.extended_roadrunner.ExtendedRoadRunner`

Use example

```
ex = 'S1 -> S2; k1*S1; k1 = 0.1; S1 = 10'
import tellurium as te
r = te.loada(ex)

# every way below works
from scanner import load
model = load(r)
model = load(ex)
model = ('path/To/SBML')
```

class Model

This module is the primary module used for interacting with the library. Model is returned by the load function for the end user to manipulate.

Attributes

- `scanTarget(NamedArray)-> any : function`
  - Function used to define what is examined during the parameter scan. Takes in the output of .ExtendedRoadRunner.simulate() and returns the target. Initialized using self.setTarget
  - During the parameter scan, each simulation result is processed by the function and the output is stored in the 'output' attribute
  - see *Custom Target* section in the tutorial for an example of this attribute
- `output : list`
  - list of outputs of self.scanTarget function
- `results : list`
  - list of raw simulation outputs
- `scanRange : list`
  - List/lists used to define the range of a parameter scan. Initialized using self.setRange
- `simParameters : tuple`
  - arguments to pass down to .ExtendedRoadRunner.simulate
- `scanParameterIds : list of string`
  - list of string list of parameter Ids used in the parameter scan. strings must mirror Ids from self.listParameters()

Functions

- `simulate(start, stop, points)`
  - Inherited function. Runs time series simulation with (start, stop, # of points) as required parameters
- `listParameters()`
  - Returns list of possible parameters for the `setParameters()` function
  - IDs of species entail the initial concentrations
- `listParametersConcentrations()`
  - Returns list of species in the concentration form
  - The output of `self.simulate` is a NamedArray with names of the form

    `'[IdOfSpecies]'`

  - this function simply adds the concentration brackets to species for ease of access to said NamedArray

- `getData()`
  - returns the information from Model.output in an ndarray
  - for n number of scan parameters, each dimention represents one scan parameter
  - order of dimentions follows order in `scanParameterIds` attribute
- `setParameters(params)`
  - selects parameters for scanning
  - changes the `scanParameterIds` attribute
- `setTarget(target)`
  - selects the scan target, accepts a function that manipulates NamedArray output of ExtendedRoadRunner.simulate()
  - changes the default `scanTarget` function
- `setScanRange(*rangeArgs, type = linear, uniform = True)`
  - creates the range/ranges for the parameters during the scan, accepts values to pass to the scanner or a list of values to pass directly to the scan
  - fills the `scanRange` attribute
- `scan()`
  - runs the parameter scan, executes a dry run(no saving) by default
  - utilizes the following attributes to start the scan:
    - `self.scanParameterIds`
    - `self.scanRange`
    - `self.simParameters`
    - `self.scanTarget`
  - Results of the simulations and the output of `scanTarget` are stored in `self.results` and `self.output` respectively
- `plotSimple('targetID')`
  - creates a simple plot of a given parameter time series
  - assumes that target was a time series
- `plotOver('parameterID')`
  - creates a series of plots w.r.t. a given parameter

  - assumes that target was a time series and that a 2D scan was performed

- `plotHeatMap('parameter1', 'parameter2')`
  - creates the elasticity heatmap
  - assumes 2D scan and a single value target output

Possible extra features

Saving and Loading from save

The library can create a COMBINE archive with the model encoded in SBML, and the parameter scan described with SEDML.

In order to save the current parameter scan:

```
model.save("name_of_file")
```

Subsequently, one would be able to load the previously loaded archive with:

```
from scanner import load

fromSave = True
model = load("path_to_archive", fromSave)
```

Live progress in terminal

One may implement a reporting tool that displays the state of the parameter scan in terminal with items such as how many simulations are left, time to simulate, etc.

Kripke structure based model checking

Instead of relying on computationally costly ODE solving, it is possible to significantly reduce computational complexity and, consequently, the speed of modeling using so called "model checking" as proposed by Barnat et al[1]

References

1. J. Barnat, L. Brim, D. Šafránek and M. Vejnár, **Parameter Scanning by Parallel Model Checking with Applications in Systems Biology**, 2010 Ninth International Workshop on Parallel and Distributed Methods in Verification, and Second International Workshop on High Performance Computational Systems Biology, 2010, pp. 95-104, doi: 10.1109/PDMC-HiBi.2010.21.
2. H. Sauro, **Control Theory for Bioengineers**, ISBN 13: 978-0982477380

3. Hoops S., Sahle S., Gauges R., Lee C., Pahle J., Simus N., Singhal M., Xu L., Mendes P. and Kummer U. (2006). **COPASI: a COmplex PAthway SImulator**. Bioinformatics 22, 3067-74.

4. **MATLAB**. (2018). 9.7.0.1190202 (R2019b). Natick, Massachusetts: The MathWorks Inc.

5. Medley et al. (2018). **Tellurium notebooks—An environment for reproducible dynamical modeling in systems biology.** PLoS Computational Biology, 14(6), e1006220.

6. E. T. Somogyi(a), J. K. Medley (c), M. T. Karlsson (b), M. Swat 1, M. Galdzicki (c), K. Choi (c), W. Copeland (c), L. Smith (c), C. Welsh (c) and H. M. Sauro (c) (2013-2022). **libRoadRunner library**

   (a) Biocomplexity Institute, Indiana University, Simon Hall MSB1, Bloomington, IN 47405

   (b) Dune Scientific, 10522 Lake City Way NE, 302 Seattle WA

   (c) Department of Bioengineering, University of Washington, Seattle, WA, 98195

7. Lucian P. Smith, Frank T. Bergmann, Deepak Chandran, Herbert M. Sauro, **Antimony: a modular model definition language**, Bioinformatics, Volume 25, Issue 18, 15 September 2009, Pages 2452–2454, https://doi.org/10.1093/bioinformatics/btp401