StrikePy v1.0

User manual

David Rey Rostro

Universidade de Vigo

davidreyrostro@gmail.com

April 9, 2022

Contents

1 Introduction

StrikePy v1.0 is a Python implementation of STRIKE-GOLDD, which is a MATLAB toolbox that analyses the local structural identifiability and observability of nonlinear dynamic models, which can have multiple time-varying and possibly unknown inputs. The goal of StrikePy is not to be better than the Matlab version or other symbolic computation oriented language implementations. Python is not optimised for this, so the aim of StrikePy is to offer a free alternative that gets the right results for the different models to be analysed.

1.1 Theoretical foundations

StrikePy takes up the most important functionality of STRIKE-GOLDD, which adopts a differential geometry approach, recasting the identifiability problem as an observability problem. Essentially, the observability of the model variables (states, parameters, and inputs) is determined by calculating the rank of a generalized observability-identifiability matrix, which is built using Lie derivatives. When the matrix does not have full rank, there are some unobservable variables. If these variables are parameters, they are called (structurally) unidentifiable. The procedure determines the subset of identifiable parameters, observable states, and observable (also called reconstructible) inputs, thus performing a "Full Input-State-Parameter Observability" analysis (FISPO). This approach is directly applicable to many models of small and medium size; larger systems can be analysed using additional features of the method. One of them is to build observability-identifiability matrices with a reduced number of Lie derivatives. In some cases these additional procedures allow to determine the identifiability of every parameter in the model (complete case analysis); when such result cannot be achieved, at least partial results – i.e. identifiability of a subset of parameters – can be obtained.

1.2 Version and publication history

- The only version of StrikePy is v1.0 and was presented in 2022.

2 License

StrikePy as STRIKE-GOLDD is licensed under the GNU General Public License version 3 (GPLv3), a free, copyleft license for software.

3 Availability

StrikePy v1.0 can be downloaded from the following website:
https://github.com/afvillaverde/StrikePy

4 Software contents

StrikePy consists of several files organised as follows:

<u>Root folder (/StrikePy/):</u>

- options.py is the file that the user must edit in order to specify the problem to solve and the options for solving it.
- strike_goldd.py is the main file. Contains most of the code of the algorithm.
- RunModels.py is a file created to run StrikePy directly. It contains a brief explanation of usage and some examples.

<u>Functions folder (/StrikePy/functions/):</u>

- elim_and_recalc.py determines identifiability of individual parameters one by one, by successive elimination of its column in the identifiability matrix and recalculation of its rank.
- rationalize.py rationalises expressions. It avoids errors when models are stated with equations that include some numerical values (equations that are not fully symbolic)
- __pyache__ is a directory containing bytecode cache files that are automatically generated by python, i.e. compiled python, or .pyc, files.

<u>Models folder (/StrikePy/models/):</u>

This folder stores the models to be analysed by the toolbox. A number of predefined models are included.

Custom options folder (/StrikePy/custom_options/):

Folder where the custom options files for the different models should be added (contains the custom options files for some of the models given as examples.).

Results folder (/StrikePy/results/):

This folder stores a summary of the results of the models analysed by the toolbox and the observability / identifiability matrix.

Documentation folder (/StrikePy/doc/):

Folder which includes this manual.

.idea folder (/StrikePy/.idea/):

Contains different files auto-generated by the IDE when the project was created. It allows the IDE to recognise all StrikePy files as a single project.

5 Requirements and installation

5.1 Requirements

StrikePy v1.0 can run on any operating system that has a version of Python installed. Dependencies that must be installed to run StrikePy:

- The numpy library (‘pip install numpy’)
- The sympy library (‘pip install sympy’)
- The symbtools library (‘pip install symbtools’)

Note: if you use the pip as installation method to use StrikePy, the 3 modules mentioned previously are installed automatically. This is explained later.

StrikePy has been developed and tested using the PyCharm IDE and Python 3.9 so it is recommended to install them to use the tool.

5.2 Download and install using GitHub

1. Download the StrikePy v1.0 module from this website: https://github.com/afvillaverde/strike-goldd/StrikePy
2. Unzip it.

5.3 Download and install using pip

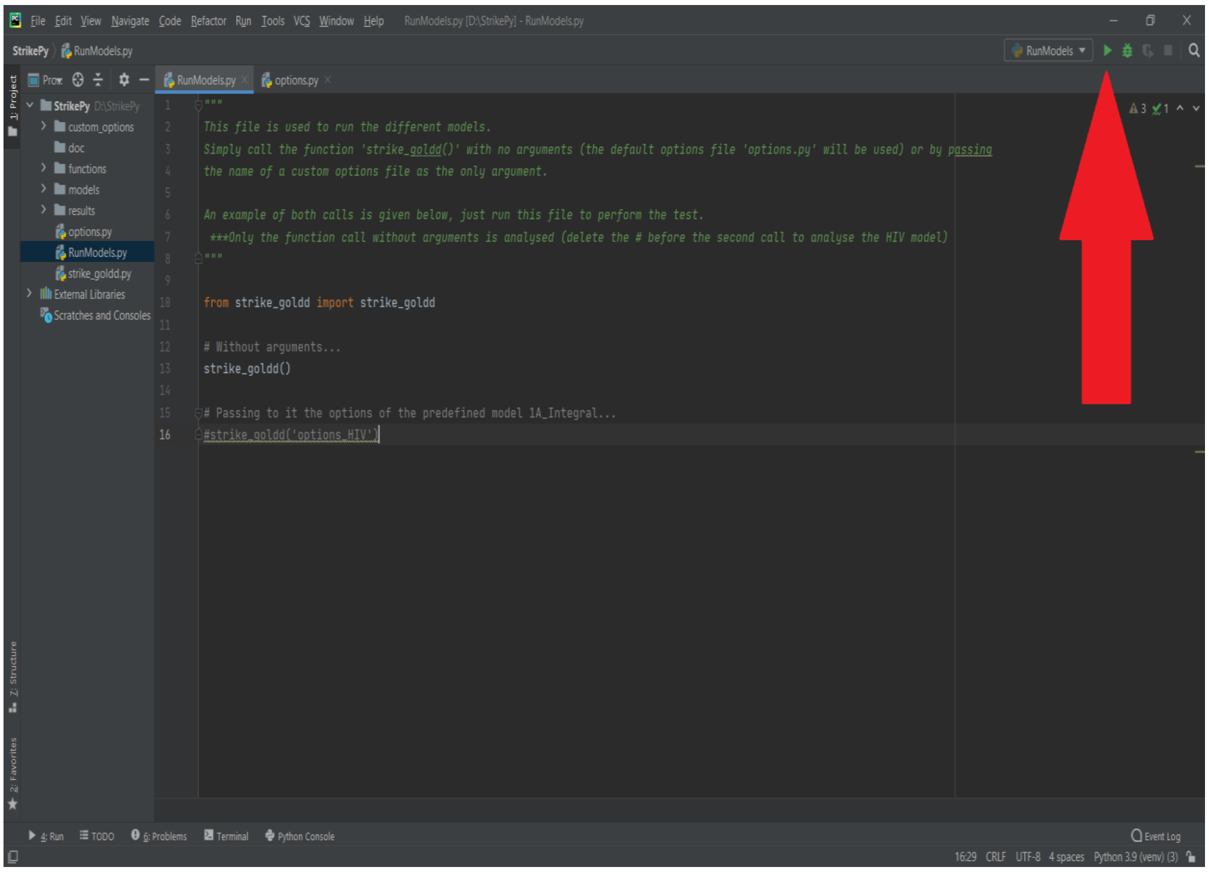
Install the StrikePy module by typing 'pip install StrikePy' in the terminal or by typing StrikePy in the installer belonging to the PyCharm IDE.

6 Quick start: using StrikePy in one minute

6.1 Using GitHub

To start using StrikePy, simply follow these steps (based on PyCharm IDE):

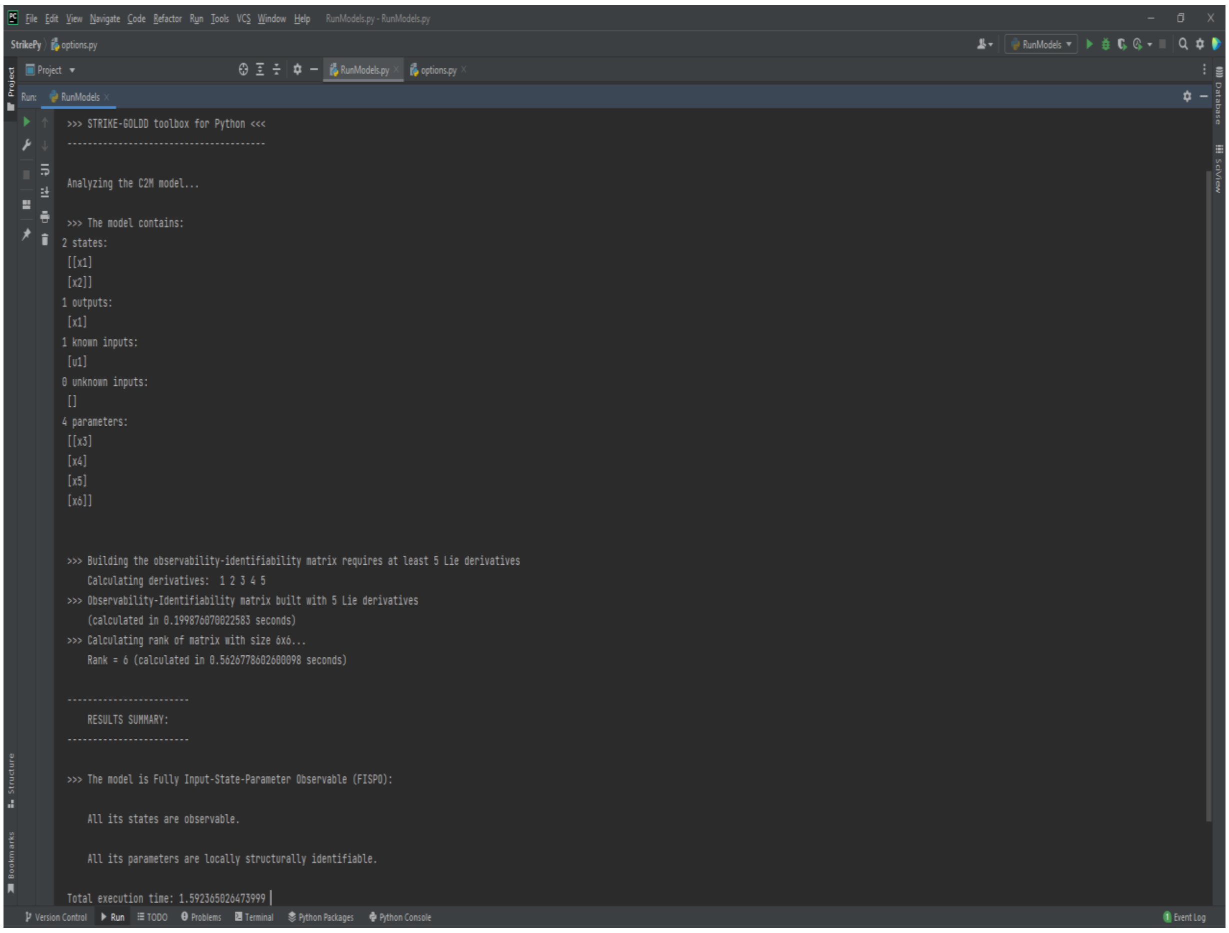
1. Complete the installation instructions in section 5.2.
2. Install the numpy, sympy and symbtools modules as described in section 5.1.
3. Define the problem by creating a new .py file and include it in the models folder.
4. Edit the options.py file to adapt it to your model or create a new options file with the ending .py and include it in the custom_options folder.
   ** You can skip steps 3 and 4 so StrikePy will analyse a predefined model with default options.
5. Run StrikePy (to do this you have to open RunModels.py and run it clicking on the button shown in Fig 1, you will see two examples of StrikePy calls in the comments of the file).



```
"""
This file is used to run the different models.
Simply call the function 'strike_goldd()' with no arguments (the default options file 'options.py' will be used) or by passing
the name of a custom options file as the only argument.

An example of both calls is given below, just run this file to perform the test.
 ***Only the function call without arguments is analysed (delete the # before the second call to analyse the HIV model)
"""


from strike_goldd import strike_goldd

# Without arguments...
strike_goldd()

# Passing to it the options of the predefined model 1A_Integral...
#strike_goldd('options_HIV')
```

Figure 1: Screenshot of the RunModels.py file

Done! Results will be reported in the screen and will also be saved in the ‘results’ folder. A screenshot of the default model execution is shown in Fig. 2.



```
 >>> STRIKE-GOLDD toolbox for Python <<<
 --------------------------------------


 Analyzing the C2M model...


 >>> The model contains:
2 states:
 [[x1]
 [x2]]
1 outputs:
 [x1]
1 known inputs:
 [u1]
0 unknown inputs:
 []
4 parameters:
 [[x3]
 [x4]
 [x5]
 [x6]]



 >>> Building the observability-identifiability matrix requires at least 5 Lie derivatives
     Calculating derivatives:  1 2 3 4 5
 >>> Observability-Identifiability matrix built with 5 Lie derivatives
     (calculated in 0.199876070022583 seconds)
 >>> Calculating rank of matrix with size 6x6...
     Rank = 6 (calculated in 0.5626778602600098 seconds)

 ------------------------
     RESULTS SUMMARY:
 ------------------------


 >>> The model is Fully Input-State-Parameter Observable (FISPO):


     All its states are observable.


     All its parameters are locally structurally identifiable.


 Total execution time: 1.592365026473999
```

Figure 2: Screenshot of the results of analysing the 'C2M' model.

6.2 Using pip

To start using StrikePy, simply follow these steps (based on PyCharm IDE):

1. Complete the installation instructions in section 5.3.
2. Locate the 'StrikePy' folder in the directory where pip installed the module.
3. Define the problem by creating a new .py file and include it in the models folder.
4. Edit the options.py file to adapt it to your model or create a new options file with the ending .py and include it in the custom_options folder.

   ** You can skip steps 3 and 4 so StrikePy will analyse a predefined model with default options.
5. Run StrikePy, some examples are shown:

   In this example, the model whose name has been placed in the 'options.py' file will be analysed. In this case, 'C2M' has been written, so this will be the model to be analysed.

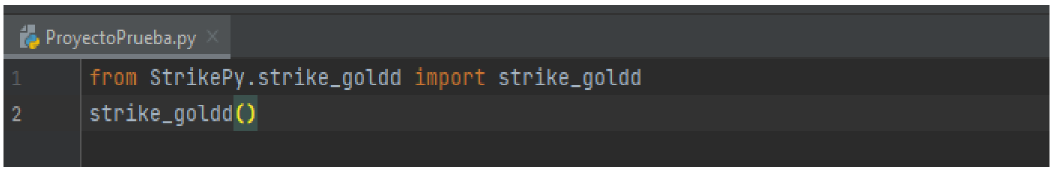
The first thing to do is to import from the StrikePy module ‘strike_goldd’ into one of the project's executable files:

```
from StrikePy.strike_goldd import strike_goldd
```

Secondly the function 'strike_goldd' must be called and in this case as we will use the 'options.py' file which is executed by default no arguments are passed to it:

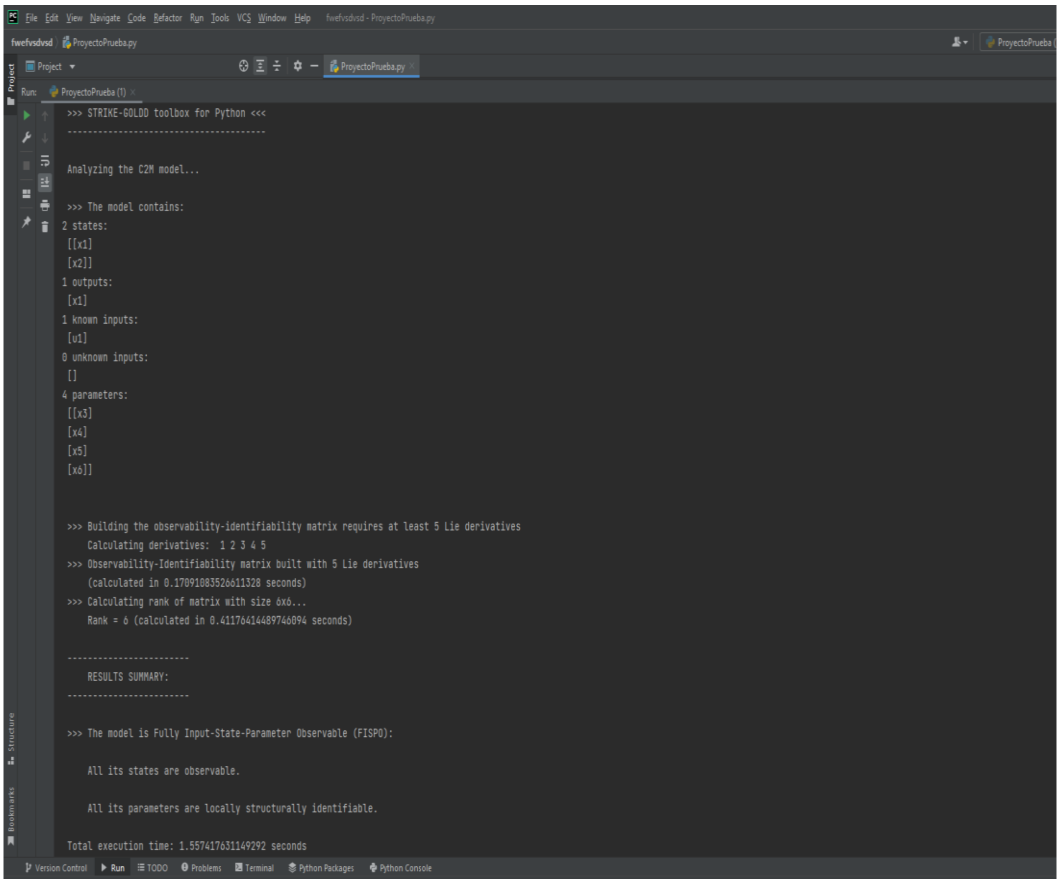
```
strike_goldd()
```

The Python file will look like this:



```
from StrikePy.strike_goldd import strike_goldd
strike_goldd()
```

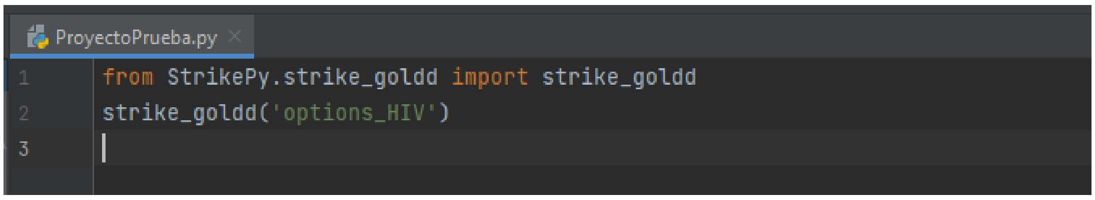
Just press play and the results will look similar to those shown in Fig. 3.



```
 >>> STRIKE-GOLDD toolbox for Python <<<
 ---------------------------------------


 Analyzing the C2M model...


 >>> The model contains:
2 states:
 [[x1]
 [x2]]
1 outputs:
 [x1]
1 known inputs:
 [u1]
0 unknown inputs:
 []
4 parameters:
 [[x3]
 [x4]
 [x5]
 [x6]]



 >>> Building the observability-identifiability matrix requires at least 5 Lie derivatives
     Calculating derivatives:  1 2 3 4 5
 >>> Observability-Identifiability matrix built with 5 Lie derivatives
     (calculated in 0.17091083526611328 seconds)
 >>> Calculating rank of matrix with size 6x6...
     Rank = 6 (calculated in 0.41176414489746094 seconds)


 ------------------------
     RESULTS SUMMARY:
 ------------------------


 >>> The model is Fully Input-State-Parameter Observable (FISPO):


     All its states are observable.


     All its parameters are locally structurally identifiable.


 Total execution time: 1.557417631149292 seconds
```

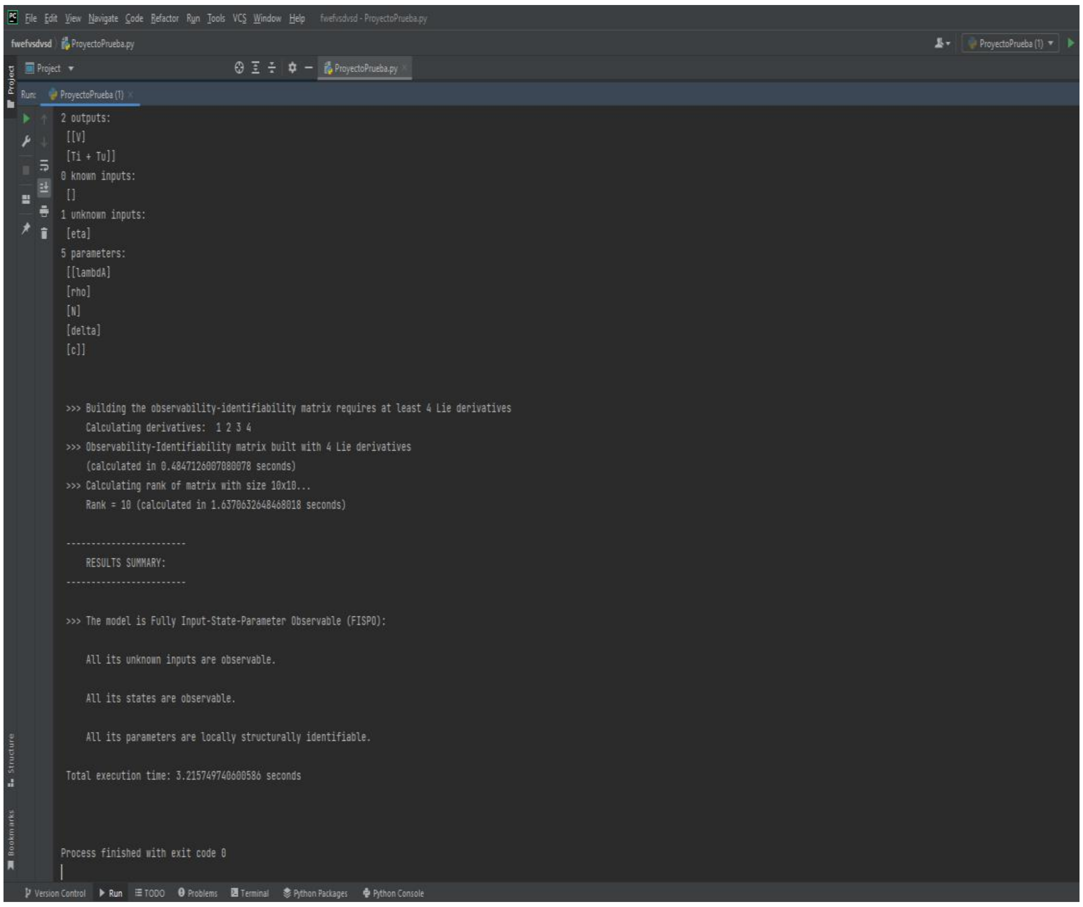
Figure 3: Screenshot of of the results of analysing the ‘C2M’ model

This second example shows how to analyse a model with StrikePy by calling a custom options file created for that model. In this case we will analyse the HIV model, so like in the first example we start by importing the StrikePy module 'strike_goldd' into the current project. Then we call the 'strike_goldd' function passing as argument the name of the custom options file, which will be located in the custom_options folder. The file will look like this:



```python
from StrikePy.strike_goldd import strike_goldd
strike_goldd('options_HIV')
```

Just press play and the results will look similar to those shown in Fig. 4



File Edit View Navigate Code Refactor Run Tools VCS Window Help fwefvsdvsd - ProyectoPrueba.py
fwefvsdvsd ProyectoPrueba.py
Project
ProyectoPrueba.py
Run: ProyectoPrueba (1)
Structure
Bookmarks
Version Control Run TODO Problems Terminal Python Packages Python Console

```
2 outputs:
 [[V]
 [Ti + Tu]]
0 known inputs:
 []
1 unknown inputs:
 [eta]
5 parameters:
 [[lambdA]
 [rho]
 [N]
 [delta]
 [c]]


 >>> Building the observability-identifiability matrix requires at least 4 Lie derivatives
     Calculating derivatives:  1 2 3 4
 >>> Observability-Identifiability matrix built with 4 Lie derivatives
     (calculated in 0.4847126007080078 seconds)
 >>> Calculating rank of matrix with size 10x10...
     Rank = 10 (calculated in 1.6370632648468018 seconds)

 ------------------------
     RESULTS SUMMARY:
 ------------------------

 >>> The model is Fully Input-State-Parameter Observable (FISPO):

     All its unknown inputs are observable.

     All its states are observable.

     All its parameters are locally structurally identifiable.

 Total execution time: 3.215749740600586 seconds


Process finished with exit code 0
```

Figure 4: Screenshot of the results of analysing the ‘HIV’ model.

7 Usage

7.1 Input: entering a model

StrikePy reads models stored as .py. The model states, outputs, inputs, parameters, and dynamic equations must be defined as vectors of symbolic variables; the names of these vectors must follow the specific convention shown in Table 1. **Important**: x, p, u, w, f, h are reserved names, which must be used for the variables and/or functions listed in Table 1 and cannot be used to name any other variables. However, it is possible to use variants of them, e.g. x_1 , x_2, p_{23}, xp, ….

Name	Reserved for:	Common mathematical notation:
x	state vector	x(t)
p	unknown parameter vector	θ
u	known input vector	u(t)
w	unknown input vector	w(t)
f	dynamic equations	$\dot{x}$(t) = f(x(t), u(t), w(t), θ)
h	output function	y = h(x(t), u(t), w(t), θ)

Table 1: **reserved variable and function names**. The names in the table are reserved for certain variables and functions. They must not be used for naming arbitrary model quantities. However, it is possible to use variants of them, e.g. x_1 , x_2, p_{23}, xp, ….

7.1.1 Example: defining the MAPK model

Here we illustrate how to define a model using the MAPK example included in the models folder. The file read by StrikePy is **MAPK.py**, which stores the model variables and this is where the declaration of the model could be edited. In the following lines we comment the different parts of the file, illustrating the process of defining a suitable model.

First, the sympy library must be imported; in this case it is imported as sym:

```
import sympy as sym
```

Then, all the parameters, states, and any other entities (such as inputs or known constants) appearing in the model must be defined as symbolic variables:

```
k1, k2, k3, k4, k5, k6 = sym.symbols('k1 k2 k3 k4 k5 k6')
ps1, ps2, ps3 = sym.symbols('ps1 ps2 ps3')
s1t, s2t, s3t = sym.symbols('s1t s2t s3t')
KK1, KK2 = sym.symbols('n1 n2')
alpha = sym.Symbol('alpha')
n1, n2 = sym.symbols('n1 n2')
```

Next we define the state variables, by creating a column vector named x:

```
x = [[ps1], [ps2], [ps3]]
```

Similarly, we define the vector of output variables, which must be named h. In this case they coincide with the state variables:

```
h = [[ps1], [ps2], [ps3]]
```

Similarly, we define the known input vector, u, and the unknown input vector, w. If there are no inputs, enter blank vectors or or do not declare them:

```
u = []
w = []
```

The vector of unknown parameters must be called p:

```
p = [[k1], [k2], [k3], [k4], [k5], [k6], [s1t], [s2t], [s3t], [KK1],
[KK2], [n1], [n2], [alpha]]
```

The dynamic equations dx/dt must also be entered as a column vector, called f. It must have the same length as the state vector x:

```
f = [[k1*(s1t-ps1)*(KK1**n1)/(KK1**n1+ps3**n1)-k2*ps1], [k3*(s2t-
ps2)*ps1*(1+(alpha*ps3**n2)/(KK2**n2+ps3**n2))-k4*ps2], [k5*(s3t-
ps3)*ps2-k6*ps3]]
```

Finally, include this line so that the main function picks up all the variables defined in the model correctly:

```
variables_locales = locals().copy()
```

7.2 Analysing a model: known vs unknown inputs

The use of StrikePy for analysing a model was already illustrated in section 6. This section provides a few more details, basically about the use of models with known and unknown inputs.

7.2.1 Example: two-compartment model with known input

Section 6 showed how to analyse the default example, which is a two-compartment model with a known input, using default settings. By default, in the option.py file the option **nnzDerU** is set to **nnzDerU** = 1. This means that the model is analysed with exactly one non-zero derivative of the known input. If we set **nnzDerU** = 0, all input derivatives are set to zero. Running the two-compartment model example with this setting yields that the model is unidentifiable. Hence, this model requires a ramp or a higher-order polynomial input to be structurally identifiable and observable. Note that for models with several inputs it is necessary to specify a vector, e.g. **nnzDerU** = [0, 1] for two inputs (or any other numbers, e.g. **nnzDerU** = [2, 2]).

7.2.2 Example: two-compartment model with unknown input

Let us now consider the two-compartment model with unknown input, and with the parameter b considered as known. This is already implemented in the model file **C2M_unknown_input_known_b** provided with the library. The analysis of this model yields that all its parameters are structurally unidentifiable and its unmeasured state and input are unobservable. This is obtained for any choice of **nnzDerW** (0, 1, 2, ...). We now consider a version of this model in which both b and k_{1e} are considered known. This is implemented in the file **C2M_unknown_input_known_b_k1e**. In this case, the analysis yields that the model is fully observable (FISPO). This is obtained for any choice of **nnzDerW** (0, 1, 2, 3, ...). Caution, in both cases there are two unknown inputs so it is necessary to specify a nnzDerU vector in the options file to avoid errors. For example **nnzDerU** = [1, 2] or **nnzDerU** = [0, 0].

7.3 Options

The model to analyse, as well as the options for performing the analysis, are entered in the **options.py** file. All options have default values that can be modified by the user. In the **options.py** file the options are classified in four groups, as follows:

(1) NAME OF THE MODEL TO BE STUDIED: The first block consists of solely one option, the name of the model to analyse. By default it is set to one of the models provided with the toolbox, the two-compartment linear model with one input:

```
modelname = 'C2M'
```

The user may select other models provided with the toolbox – included in folder /models – or define a new model as explained in Section 7.1.

(2) FISPO ANALYSIS OPTIONS: The second block consists of the following options:

checkObserver, maxLietime, nnzDerU and nnzDerW.

Their meaning is explained in the comments of the options.py file. Note that all the above options are in general scalar values. The exceptions are **nnzDerU** and **nnzDerW**, which, for models with several inputs, must be row vectors with the same number of elements as inputs, e.g., for two inputs:

```
nnzDerW = [0, 1]
```

(3) KNOWN/IDENTIFIABLE PARAMETERS: The last block is used for entering parameters that have already been classified as identifiable. This reduces the computational complexity of the calculations and may thus enable a deeper analysis, which can lead to more complete results. For example, if StrikePy has already determined that two parameters $p1$ and $p2$ are identifiable, we may enter:

```
p1, p2 = sym.symbols('p1 p2')
prev_ident_pars = [p1, p2]
```

This option can also be used to assume that some parameters are known, despite being entered as unknown in the model definition. This is useful to test what happens when fixing some parameters, without having to modify the model file.

7.4 Output

StrikePy reports the main results of the identifiability analysis on screen. Additionally, it creates several .txt in the results folder:

- A file named **id_results_MODELNAME_DATE.txt**, with a summary of the results of the identifiability analysis.
- One or several files with the generalized observability-identifiability matrices calculated with a given number of Lie derivatives. They are stored in separate files so that they can be reused in case a particular run is aborted due to excessive computation time. These files are named with this structure: **obs_ident_matrix_MODEL_NUMBER_OF_Lie_deriv.txt**.

8 Contributors

StrikePy has been developed as part of the final degree project by David Rey Rostro (Universidade de Vigo, davidreyrostro@gmail.com). All the work has been supervised by Alejandro F. Villaverde (Universidade de Vigo, afvillaverde@uvigo.gal; previously at CSIC) who developed the Matlab toolbox STRIKE-GOLDD on which StrikePy is based.

References

STRIKE-GOLDD_manual which can which can be downloaded by following this link: https://github.com/afvillaverde/strike-goldd/blob/master/STRIKE-GOLDD/doc/STRIKE-GOLDD_manual.pdf