

# StrikePy v1.0

## User manual

Developed by David Rey Rostro ([davidreyrostro@gmail.com](mailto:davidreyrostro@gmail.com))  
Supervised by Alejandro F. Villaverde ([afvillaverde@uvigo.gal](mailto:afvillaverde@uvigo.gal))

Universidade de Vigo

April 21, 2022

## Contents

1	Introduction .....	2
1.1	Theoretical foundations .....	2
1.2	Version and publication history .....	2
2	License .....	2
3	Availability.....	3
4	Software contents .....	3
5	Requirements and installation .....	4
5.1	Requirements .....	4
5.2	Download and install using GitHub.....	4
5.3	Download and install using pip .....	4
6	Quick start: using StrikePy in one minute.....	4
6.1	Using GitHub .....	5
6.2	Using pip .....	7
7	Usage.....	8
7.1	Input: entering a model .....	8
7.1.1	Example: defining the MAPK model .....	8
7.2	Analysing a model: known vs unknown inputs .....	10
7.2.1	Example: two-compartment model with known input .....	10
7.2.2	Example: two-compartment model with unknown input .....	10
7.3	Options.....	10
7.4	Output .....	11
8	Contributors.....	12

# 1 Introduction

StrikePy v1.0 is a Python implementation of the FISPO algorithm from the MATLAB toolbox STRIKE-GOLDD. It analyses the structural local identifiability and observability of nonlinear dynamic models, which may have multiple time-varying and possibly unknown inputs. Its main aim is to provide a free alternative to MATLAB, not to outperform the MATLAB version or other tools implemented in symbolic computation oriented languages.

## 1.1 Theoretical foundations

StrikePy implements the core functionalities of the FISPO algorithm included in STRIKE-GOLDD. It adopts a differential geometry approach, recasting the structural local identifiability problem as an observability problem. Essentially, the observability of the model variables (states, parameters, and unknown 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. StrikePy 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 current version of StrikePy is v1.0 (released in April 2022).

# 2 License

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

### 3 Availability

StrikePy can be downloaded from GitHub and from PyPI:

- <https://github.com/afvillaverde/StrikePy>
- <https://pypi.org/project/StrikePy/>

### 4 Software contents

StrikePy consists of the following files:

Root folder (/StrikePy/):

- **options.py** is the only file that the user must edit. It specifies the problem to solve and the options for solving it.
- **strike\_gold.py** is the main file. It contains most of the code of the algorithm.
- **RunModels.py** is a file that executes StrikePy, by calling `strike_gold.py`. It contains a brief usage explanation and some examples.

Functions folder (/StrikePy/functions/):

- **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)
- **\_\_pycache\_\_** is a directory containing bytecode cache files that are automatically generated by python, i.e. compiled python, or .pyc, files.

Models folder (/StrikePy/models/):

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

Custom options folder (/StrikePy/custom\_options/):

This folder can be used to store custom options files for specific models.

### Results folder (/StrikePy/results/):

This folder stores a summary of the results of the models analysed by the toolbox, as well as their observability / identifiability matrix.

### Documentation folder (/StrikePy/doc/):

This folder stores the present manual.

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

This folder 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

- A Python 3.9 installation.
- The numpy, sympy, and symbtools libraries.

### 5.2 Download and install using GitHub

1. Download StrikePy from: <https://github.com/afvillaverde/StrikePy>
2. Unzip it.
3. Install the the required libraries by typing in the python command prompt:

```
pip install numpy sympy symbtools
```

### 5.3 Download and install using pip

StrikePy v1.0 is included in the PyPI repository. To install it, type:

```
pip install strikepy
```

(The command above also installs the the numpy, sympy, and symbtools libraries.)

## 6 Quick start: using StrikePy in one minute

### 6.1 Using GitHub

To start using StrikePy, simply follow these steps:

1. Complete the installation instructions in [section 5.2](#).
2. Define the problem by creating a new .py file and include it in the models folder.
3. Edit the options.py file to adapt it to your model, OR create a new options file with extension .py and include it in the custom\_options folder.  
\*\* QUICK DEMO EXAMPLE: If you skip steps 2 and 3, StrikePy will analyse a predefined model with default options.
4. Run the code included in RunModels.py, using a Python environment of your choice (for example, in the execution shown in Fig. 1 below we have used the JupyterLab tool included in Anaconda).

Done! Results will be reported in the screen, and they will also be saved in a file created in the 'results' folder, named 'id\_results\_MODELNAME\_DATE.txt'. A screenshot of the execution in JupyterLab is shown in Figs. 1 and 2. Fig.1 shows the results of the 'strike\_goldd()' command, which analyses the 'C2M' model specified in the default options file, 'options.py'. Fig. 2 shows the outcome of the 'strike\_goldd('options\_HIV')' command, which analyses the HIV model defined in the 'options\_HIV.py' file.

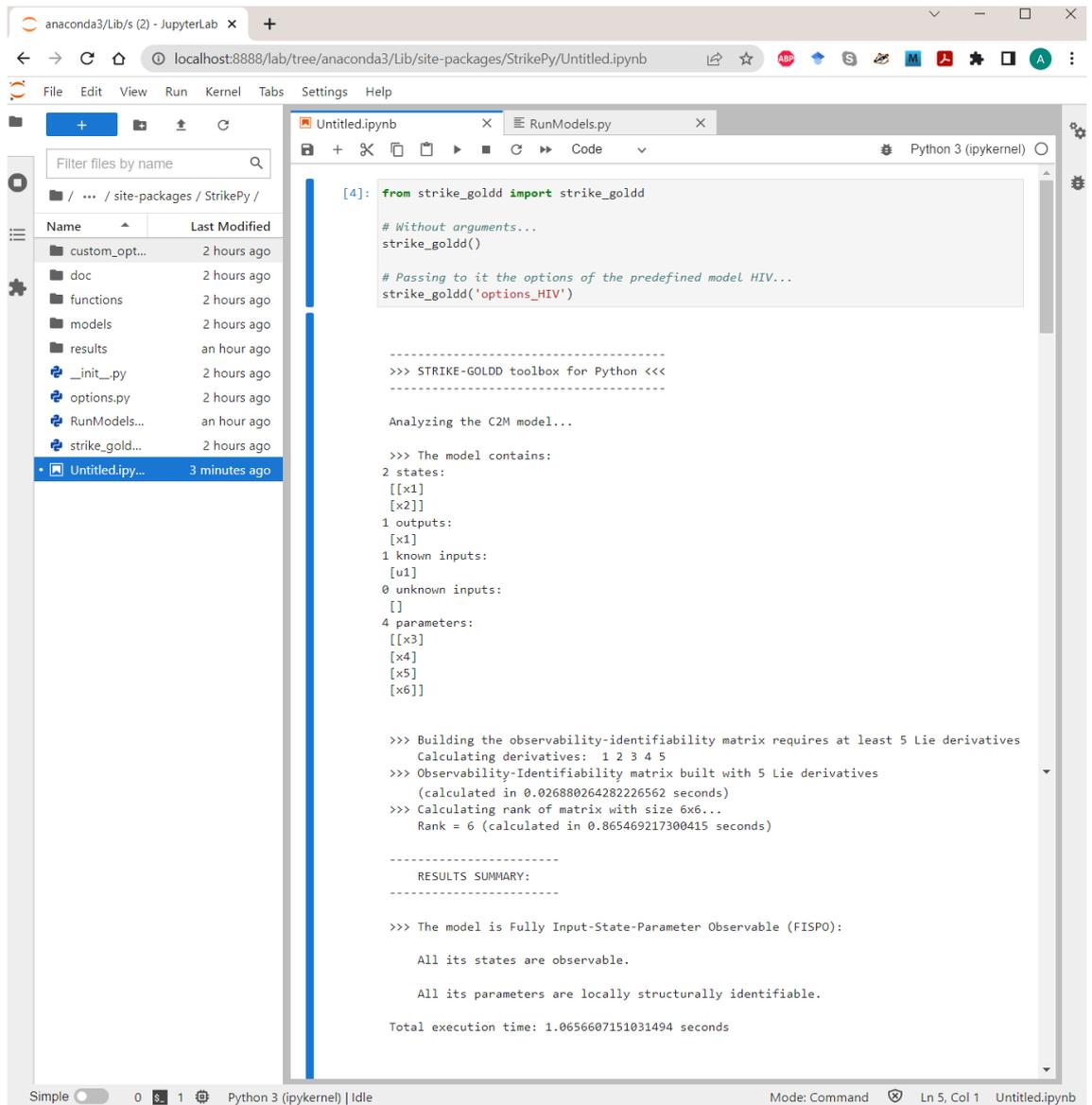


Figure 1: Screenshot of the results of analysing the 'C2M' model.

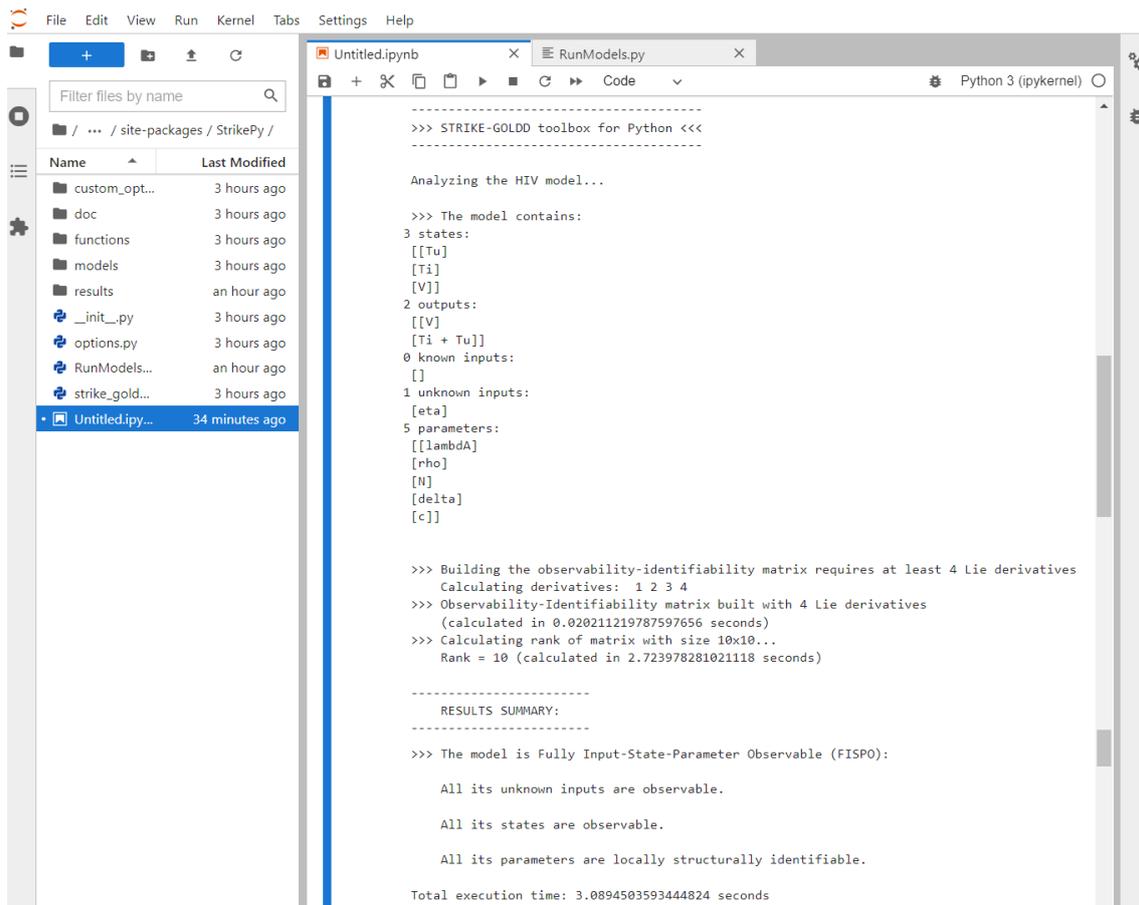


Figure 2: Screenshot of the results of analysing the ‘HIV’ model.

## 6.2 Using pip

To start using StrikePy, simply follow these steps:

1. Complete the installation instructions in [section 5.3](#).
2. Locate the 'StrikePy' folder (the directory where it was installed by pip can be found with ‘pip show strikepy’), and define the problem by creating a new .py file and include it in the models folder.
3. 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.  
**\*\* If you skip steps 2 and 3, StrikePy will analyse a predefined model with default options.**
4. Run the code included in RunModels.py (see Step 4 in [Using GitHub](#), as well as the corresponding results).

## 7 Usage

### 7.1 Input: entering a model

StrikePy reads models stored as .py files. 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.  $x_1, x_2, p_{23}, xp, \dots$

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

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.  $x_1, x_2, p_{23}, xp, \dots$

#### 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 it is where the definition of the model can be modified. In the following lines we discuss 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 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 options.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, ...). It should be noted that in both cases there are two unknown inputs, so it is necessary to specify in the options file a **nnzDerW** vector with two elements, e.g. **nnzDerW = [1, 2]** or **nnzDerW = [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 by David Rey Rostro (Universidade de Vigo, [davidreyrostro@gmail.com](mailto:davidreyrostro@gmail.com)) as part of his B. Eng. thesis. The work was supervised by Alejandro F. Villaverde (Universidade de Vigo, [afvillaverde@uvigo.gal](mailto:afvillaverde@uvigo.gal)), who developed the toolbox STRIKE-GOLDD on which StrikePy is based.