

---

# **fnyzer**

***Release 1.3.4***

**Sep 22, 2021**



---

## Contents

---

<b>1</b>	<b>Installation and resources</b>	<b>3</b>
1.1	Requirements . . . . .	3
1.2	Testing your installation . . . . .	4
1.3	Resources . . . . .	4
1.4	Queries . . . . .	4
1.5	Bibliography . . . . .	4
<b>2</b>	<b>Getting started</b>	<b>5</b>
2.1	Shell script . . . . .	6
2.2	Python interpreter . . . . .	6
2.3	optimize() and FNFactory() . . . . .	7
<b>3</b>	<b>FN specification</b>	<b>11</b>
<b>4</b>	<b>FN with intermediate states</b>	<b>17</b>
<b>5</b>	<b>FN with MPC</b>	<b>19</b>
5.1	Options . . . . .	19
5.2	Objective function . . . . .	20
5.3	Extra constraints . . . . .	20
5.4	Resets . . . . .	21
<b>6</b>	<b>Auxiliary functions</b>	<b>23</b>
6.1	Cobra models . . . . .	23
6.2	Nonlinear dynamics . . . . .	23



*fnyzer* (Flexible Nets analyZER) is an open-source Python package for the analysis of Flexible Nets (FNs), a modeling formalism for dynamic systems. *fnyzer* makes use of [pyomo](#) and an optimization solver to build and solve optimization problems associated with the FNs. Once the problems are solved, the obtained results can be plotted and saved in a spreadsheet.

This documentation first explains how *fnyzer* can be installed and executed, and then describes how FNs can be specified and analysed.

## **Table of contents**



---

## Installation and resources

---

The easiest way to install *fnyzer* is to use `pip` (the standard tool for installing Python packages). Execute the following line in a shell:

```
$ pip install fnyzer
```

### 1.1 Requirements

In order to solve the optimization problems associated with the flexible nets, *fnyzer* requires a solver supported by `pyomo`. For untimed and steady state analysis of FNs, the `GLPK` solver can be used. `GLPK` can be straightforwardly installed as a Debian package, e.g. in Ubuntu:

```
$ apt-get install glpk-utils
```

Windows users can follow the instructions in this [this link](#) to install `GLPK`. In general, the set of constraints associated with the FNs include quadratic constraints, and hence, *fnyzer* requires a solver that can handle such constraints. *fnyzer* has been mainly tested with `Gurobi` and `CPLEX` (both solvers offer free academic licenses).

**Warning:** `GLPK` does not support quadratic constraints and will not be able to solve some optimization problems.

The installation of at least one solver in your system is necessary to run *fnyzer*. You can check that a solver is properly installed by executing these commands:

```
$ cplex
Welcome to IBM(R) ILOG(R) CPLEX(R) Interactive Optimizer 12.6.1.0
...
$ gurobi.sh
Gurobi Interactive Shell (linux64), Version 6.0.0
...
```

## 1.2 Testing your installation

Download the file `fnexamples.py` in your working directory. If you have CPLEX in your system, execute the following line:

```
$ fnyzer fnexamples net0cplex
```

If you have Gurobi in your system, execute the following line:

```
$ fnyzer fnexamples net0gurobi
```

If you have GLPK in your system, execute the following line:

```
$ fnyzer fnexamples net0glpk
```

After the execution, the files `net0.xls` and `net0.pkl` should be in your working directory.

## 1.3 Resources

`fnyzer` is hosted at Bitbucket:

- <https://bitbucket.org/Julvez/fnyzer>

## 1.4 Queries

Queries can be sent to [fnyzer@unizar.es](mailto:fnyzer@unizar.es)

## 1.5 Bibliography

*Flexible Nets: A modeling formalism for dynamic systems with uncertain parameters*; J. Júlvez, S. G. Oliver; Discrete Event Dynamic Systems: Theory and Applications, 2019. <https://doi.org/10.1007/s10626-019-00287-9>

*Modeling, analyzing and controlling hybrid systems by Guarded Flexible Nets*; J. Júlvez, S. G. Oliver; Nonlinear Analysis: Hybrid Systems, 2019. <https://doi.org/10.1016/j.nahs.2018.11.004>

*Steady State Analysis of Flexible Nets*; J. Júlvez, S. G. Oliver; IEEE Transactions on Automatic Control, 2019. <https://doi.org/10.1109/TAC.2019.2931836>

*Handling variability and incompleteness of biological data by flexible nets: a case study for Wilson disease*; J. Júlvez, D. Dikicioglu, S. G. Oliver; npj Systems Biology and Applications, 2018. <https://doi.org/10.1038/s41540-017-0044-x>



## CHAPTER 2

---

### Getting started

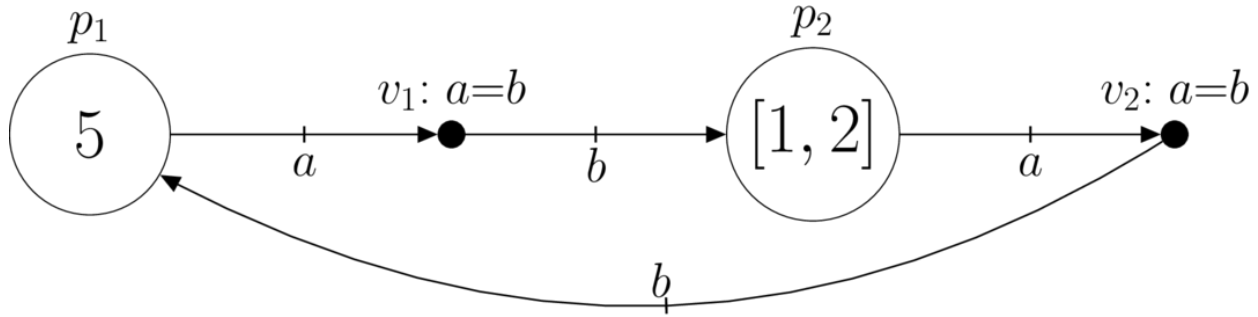
---

The file `fnexamples.py` contains examples of Flexible Nets (FNs) stored as Python dictionaries.

The first dictionary in `fnexamples.py` is `net0`:

```
net0 = {
    'name': 'net0',
    'solver': 'glpk', # Solver to be used
    'places': {      # Places and initial marking
        'p1': 5,     # Initial marking of p1
        'p2': None,  # The initial marking of p2 is not specified
    },
    'm0cons': ["1 <= m0['p2']", "m0['p2'] <= 2"], # Constraints for the
                                                    # initial marking of p2
    'vhandlers': {   # Event handlers
        'v1': [
            {'a': ('p1', 'v1'), 'b': ('v1', 'p2')}, # The same amount of tokens
            'a == b'                                # is consumed from p1 and
                                                    # produced in p2
        ],
        'v2': [
            {'a': ('p2', 'v2'), 'b': ('v2', 'p1')},
            'a == b'
        ],
    },
    'obj': {'f': "m['p1']", 'sense': 'max'}, # Objective function
    'options': {
        'antype': 'un', # Untimed analysis
        'xlsfile': 'net0.xls', # Results will be stored in this spreadsheet
        'netfile': 'net0.pkl', # The flexible net object with the resulting
                               # variables will be saved in this file
    }
}
```

This dictionary specifies a FN with two places and two event handlers. The FN does not have transitions and is therefore untimed. The graphical representation is:



The initial marking of  $p1$  is 5 and the initial marking of  $p2$  can be any value in the interval  $[1, 2]$ . Detailed information about the FN formalism can be found in the [Bibliography](#).

The objective function is to maximize the final marking of  $p1$ . The solver is GLPK and can be changed by editing the file and setting the value of 'solver' to:

```
'solver': 'cplex',
```

or

```
'solver': 'gurobi',
```

Let's see how this net can be analyzed from a shell and from the Python interpreter.

## 2.1 Shell script

*fnyzer* can be executed in a shell as follows:

```
$ fnyzer fnexamples net0
```

where *fnexample* (or *fnexample.py*) is the file containing the FN specifications and *net0* is the name of the dictionary with the FN to be optimized.

The execution of the script produces two files: *net0.xls* and *net0.pkl*. *net0.xls* is a spreadsheet with the values of the variables after the optimization; *net0.pkl* is a [PKL](#) file that stores the FN Python object.

The sheet 'Untimed' of the spreadsheet shows the value of the objective function together with the value of the variables. The maximum final marking of  $p1$  is achieved when the initial marking of  $p2$  is 2, i.e.  $m0[p2]=2$ , and the event handler  $v2$  is fired in an amount of 2, i.e.  $dm[(v2, p1)]=2$ .

## 2.2 Python interpreter

*fnyzer* can also be executed from the Python interpreter as shown below:

```
>>> from fnyzer import optimize
>>> from fnexamples import net0
>>> model, fn = optimize(net0)
```

This produces the files *net0.xls* and *net0.pkl*. The file *net0.pkl* can be used in a different session to recover the values of the variables:

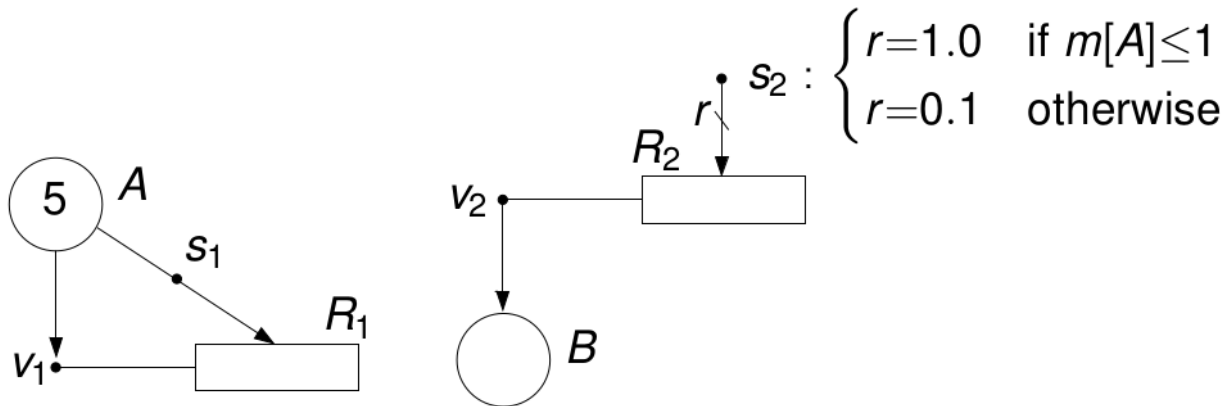
```

>>> import pickle
>>> datafile = open("net0.pkl", 'rb')
>>> fn = pickle.load(datafile)
>>> datafile.close()
>>> fn.places
{'p1': <fnyzer.netel.Place object at 0x7fba14aaf588>, 'p2': <fnyzer.netel.Place_
↪object at 0x7fba14abc278>}
>>> fn.places['p1']
<fnyzer.netel.Place object at 0x7fba14aaf588>
>>> fn.places['p1'].m
7.0
>>> fn.places['p1'].m0
5.0
>>> fn.varcs[('v2', 'p1')].dm
2.0

```

## 2.3 optimize() and FNFactory()

In the following guarded FN, the intensity produced in  $R_2$  depends on the marking of  $A$ . In particular, the marking of  $B$  increases at rate 1.0 when the marking of  $A$  is below 1 and at rate 0.1 otherwise.



The specification of this FN can be found in the dictionary *ractive* of the file `fnexamples.py`:

```

ractive = { # Repressor decay activates production
  'name': 'ractive',
  'solver': 'cplex',
  'places': {'A': 5, 'B': 0},
  'trans': {'R1': {'l0': 0, 'a0': 0}, 'R2': {'l0': 0.1, 'a0': 0}},
  'vhandlers': {
    'v1': [{ 'a': ('A', 'v1'), 'r': ('R1', 'v1') }, 'a == r'],
    'v2': [{ 'b': ('v2', 'B'), 'r': ('R2', 'v2') }, 'b == r']
  },
  'regs': {
    'off': ["m['A'] >= 1"],
    'on': ["m['A'] <= 1"],
  },
  'parts': {'Par': ['off', 'on']},
  'shandlers': {
    's1': [{ 'a': ('A', 's1'), 'r': ('s1', 'R1') }, 'a == r'],

```

(continues on next page)

(continued from previous page)

```

    's2': [{ 'r': ('s2', 'R2') }, { 'off': ['r == 0'], 'on': ['r == 1'] }],
    },
    'actfplaces': ['A'],
    'exftrans': 'all',
    'actavplaces': ['A'],
    'exavtrans': 'all',
    'obj': { 'f': "m['B']", 'sense': 'min' },
    'options': {
        'antype': 'mpc',
        'mpc': {
            'firstinlen': .1,
            'numsteps': 40,
            'maxnumins': 1,
            'flexins': False
        },
        'writevars': { 'm': ['A', 'B'], 'l': 'all', 'L': 'all', 'U': 'all' },
        'plotres': True,
        'plotvars': { 'evm': ['A', 'B'] }
    }
}

```

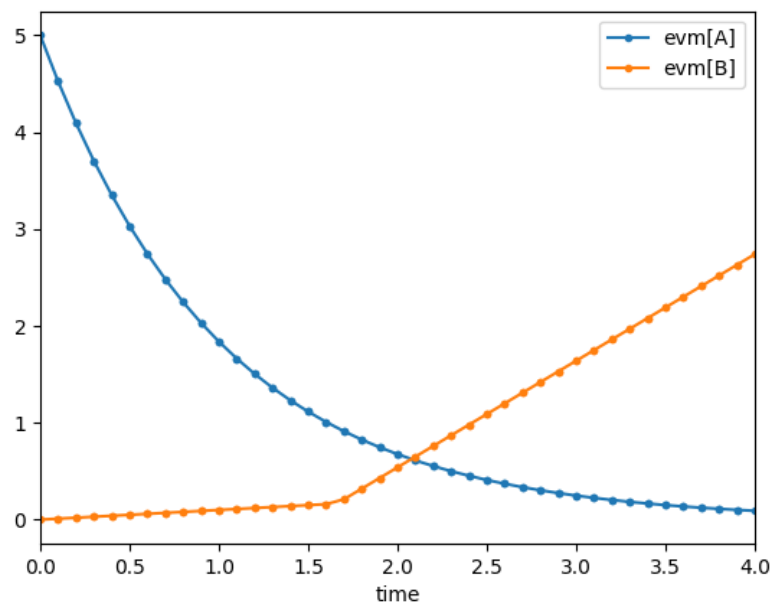
*optimize()* and *FNFactory()* are the main methods provided by *fnyzer* to optimize a FN specified by a dictionary. While *optimize()* returns a Pyomo model and a FN object, *FNFactory()* just returns a FN object. This FN object can be used to call the method *optimize()*:

```

>>> from fnyzer import FNFactory
>>> from fnexamples import ractive
>>> fnet = FNFactory(ractive)
>>> fnet.optimize()

```

This produces the files *ractive.xls* and *ractive.pkl* and the plot:



The file *ractive.pkl* can be used in a different session to save the results in a different spreadsheet and to plot the

trajectories again:

```
>>> import pickle
>>> datafile = open("ractive.pkl", 'rb')
>>> fn = pickle.load(datafile)
>>> datafile.close()
>>> fn.writexls("new_ractive.xls")
>>> fn.plotres()
```

---

**Note:** See the dictionaries *vdimdomun*, *vdimdomtr*, *vdimdomst* and *vdimdomcst* in [FlexN.py](#) and [GFlexN.py](#) for the names, dimension and domain of the variables used by *fnyzer*.

---



## CHAPTER 3

---

### FN specification

---

Flexible Nets (FNs) are specified by a Python dictionary with the fields described in this section. Notice that such dictionary can be modified by *fnyzer* to set default options and to adjust some parameters.

---

**Note:** See Supplementary File S1 of <https://doi.org/10.1038/s41540-017-0044-x> for names and meaning of the FN parameters.

---

---

**Note:** See keys in dictionaries *vdindom* (files *FlexN.py* and *GFlexN.py*) for names of the variables that are available in *fnyzer*. Those names can be used to define the objective function, to specify the variables to plot, to define extra constraints, etc.

---

The following fields can be used to specify a FN:

**'name': String denoting the name of the net**

This string will be used to generate names for 'xlsfile' and 'netfile' (see options below) if such fields are omitted.

**'solver': String denoting the solver to use**

*fnyzer* has been mainly tested with 'gurobi', 'cplex' and 'glpk'.

**Warning:** GLPK does not support quadratic constraints and will not be able to solve some optimization problems.

**'theta': Number denoting the length of the time interval for optimization**

This field is compulsory for transient analysis.

**'places': Dictionary with names of places and initial markings**

A None value means that the initial marking is not specified, and hence, it is taken as a variable that can be constrained by 'm0cons' or 'extracons', e.g.

```
'places': {'p1': 5, 'p2': None}
```

A dictionary can also be associated with each place to describe it (only the value of 'm0' will be used during analysis, e.g.

```
'p1': {'m0': 5, 'full_name': 'ATP', 'formula': 'cdcdsH'}
```

**'trans': Dictionary with names of transitions, intensities and number of actions**

This dictionary contains the names of transitions, their default intensities and their initial number of actions. A None value means that the default intensity/initial actions is not specified, and hence, it is taken as a variable that can be constrained by 'l0cons'/'a0cons' or 'extracons', e.g.

```
'trans': {'t1': {'l0': 2, 'a0': 1}, 't2': {'l0': None, 'a0': 0}}
```

**'regs': Dictionary with names and specification of regions**

Each region is specified by a list of linear constraints, e.g.

```
'regs': {'upr': ["50 >= m['p2']", "m['p2'] <= 100"],
         'lor': ["m['p2'] <= 50"]}
```

**'parts': Dictionary with names of partitions and associated regions**

Example:

```
'parts': {'Part1': ['upr', 'lor'], 'Part2': ['regalpha', 'regbeta']}
```

**'vhandlers': Dictionary with names and specification of event handlers**

Each event handler is specified by a list whose first component is a dictionary that links nicknames to connected arcs and edges, and the rest of components are the linear relations between the variables associated to the arcs and edges. Example:

```
'vhandlers': {'v1': [{'a': ('p1', 'v1'), 'b': ('v1', 'p2'), 'c': ('t1', 'v1')},
                    'a == b', 'a == c'],
              'v2': [{'a': ('p2', 'v2'), 'b': ('v2', 'p1'), 'c': ('t2', 'v2')},
                    'a <= b', 'b <= a', 'c == a']}
```

In general, any linear expression satisfying Python syntax, eg. “-2.0\*a <= -b + c” can be used.

**Warning:** Nicknames have to follow the regular expression `[_A-Za-z][_a-zA-Z0-9]*` and cannot be Python keywords.

**'shandlers': Dictionary with names and specification of intensity handlers**

The specification is similar to that of the event handlers plus the optional addition of guards for the linear relations in guarded nets. For instance, an intensity handler *s1* with unguarded and guarded arcs can be defined as:

```
'shandlers': {'s1': [{'a': ('p1', 's1'), 'x': ('s1', 't1'), 'y': ('s1', 't2'), 'z': ('s1',
→ 't3')},
                    'x==2*a', # unguarded arc
                    {'upr': ['2.1*a <= y', 'y <= 2.2*a'],
                     'lor': ['1.1*a <= y', 'y <= 1.2*a']}, # guarded arc
                    'x == 1+z']} # unguarded arc
```

**Note:** All places, transitions, vhandlers and shandlers must have different names.

**'m0cons': List of initial marking constraints**

Example:



```
'm0cons': ["100 <= m0['p1']", "m0['p1'] + 2*m0['p3']<= 110"]
```

**Warning:** there must be no spaces or separators within the variables and brackets of the constraints, i.e. “m0[‘A’] == 0.5\*m[‘A’]” or “m0[ ‘A’] == 0.5\*m[‘A’]” will raise an error.

**'l0cons': List of default intensity constraints**

Example:

```
'l0cons': ["l0['t1'] <= 8", "l0['t2'] >= 3"]
```

**'a0cons': List of initial actions constraints**

Example:

```
'a0cons': ["a0['t1']>=2.3", "a0['t2']<=12"]
```

**'mcons': List of final marking constraints**

Example:

```
'mcons': ["m['p1']>= 15", "-19+m['p1']<=m['p2']"]
```

**'mbounds': List of marking bounds**

These bounds are forced both for the final and average marking. Example:

```
'mbounds': ["m['p1']>= 7", "9<=m['p2']"]
```

**'actfplaces': List of places whose tokens must be active at the final state**

Instead of a list, it can be the string ‘all’ that accounts for all places. Example:

```
'actfplaces': ['p1', 'p2']
```

**'actavplaces': List of places whose tokens must be active all the time**

Instead of a list it can be the string ‘all’ that accounts for all places. Example:

```
'actavplaces': ['p1', 'p2']
```

**'exftrans': List of transitions with forced executions at the final state**

Instead of a list it can be the string ‘all’ that accounts for all transitions. Example:

```
'exftrans': ['t1', 't2']
```

**'exavtrans': List of transitions with forced executions over all the time interval**

Instead of a list it can be the string ‘all’ that accounts for all transitions.

```
'exavtrans': ['t1', 't2']
```

**'Ec' and 'Fc': Matrices Ec and Fc for equalities of intensitis at arcs**

Each component of Ec is a dictionary with the weights of the arcs. Example:

```
'Ec':[{('s1','t1'): 2, ('s2','t2'): -1}, {('s3','t5'): 1}]
'Fc':[0, 1]
```

implies that the following constraints must hold:

$$2 \cdot \Delta\lambda[(s1, t1)] = \Delta\lambda[(s2, t2)]$$

and

$$\Delta\lambda[(s3, t5)] = 1$$

**'E' and 'F': Matrices E and F for inequalities of intensitis at arcs**

If a component of F is set to float('inf') then a tight upper bound for the corresponding component of E will be computed. Example:

```
'E': [{('s1','t1'): 1, ('s2','t2'): 1}, {'s3','t5'): 1}]
'F': [10, float('-inf')]
```

implies that:

$$\Delta\lambda[(s1,t1)] + \Delta\lambda[(s2,t2)] \leq 10$$

must hold, and a upper bound  $b$  will be computed such that:

$$\Delta\lambda[(s3,t5)] \leq b$$

will hold.

**'wl', 'wu': Lower and upper bounds to linearize products of variables**

These values are used to linearize products of variables involving intensities and are computed automatically if they are not specified.

**'W': Upper bound to linearize implications in guarded nets**

This value is computed automatically if it is not specified.

**'extracons': List of extra constraints that must be satisfied**

These constraints will be included in the programming problem to be optimized. Example:

```
'extracons': ["avm['PrX' ]>= 40", "m['p1']+avl['t1' ]>= 3"]
```

**'obj': Dictionary with the objective function and its sense**

The key 'f' is the objective function, and the key 'sense' is the string 'min' or 'max'. Example:

```
'obj': {'f': "m['P1']", 'sense': 'max'}
```

**'options': Dictionary that specifies the following options**

---

**Note:** See dictionary *defoptions* in [FlexN.py](#) for default values of the options.

---

**'antype': String specifying the analysis type** Possible values: 'tr' = transient, 'un' = untimed, 'st' = steady state, 'cst' = constant steady state, 'mpc' = model predictive control.

**'epsiloncompF': epsilon used in the exit condition to compute F**

**'epsilonLU': epsilon to check if L[k] and U[k] are too close** If  $U[k] \leq L[k] + \text{epsilonLU}$ , then: 1) k is removed from L, U, E, F and moved to Ec[k]; and 2) Fc[k] is set to the original U[k].

**'maxitcompF': Maximum number of iterations used in the exit condition to computet F**

**'allsatoE': Boolean** If True all the intensity arcs will be included in E and the corresponding components of F will be set to float('inf'). If True the quadratic constraints are likely to be tighter and the bounds more precise. If True more CPU time is required, specially if the net is big or has many intermediate states.

**'printres': Boolean** If True print values of variables and objective function.

**'writexls': Boolean** If True write values of variables and objective function to the spreadsheet 'xlsfile'.

**'writevars': Variables to be written to the spreadsheet** If 'all' all variables are written, otherwise they have to be specifies as a dictionary like: {'avmue': 'all', 'l0': ['t1','t2'], 'm': ['p1','p2']} where 'all' stands for all the components of the key variable

**‘writeLUnegsa’:** **Boolean** If False the LU components with one element and negative weight are not written.

**‘xlsfile’:** **Name of the spreadsheet where the results will be written**

**‘savenet’:** **Boolean** If True save net after optimization, i.e., with all the computed variables, as a Python object to file ‘netfile’. The net is saved with the method `cPickle.dump()`.

**‘netfile’:** **Name of the file where the net object will be saved**

**‘printmodel’:** **Boolean** If True print the programming problem (variables, constraints, objective function).

**‘dsspe’:** **Dictionary that specifies intervals for linearization.** Dictionary { ‘type’: , ‘q’: } that specifies how the intervals  $ds=[0,d1,...,dq-1,dq]$  to linearize the products of variables are generated.

- ‘type’ can be either ‘exp’ for exponential, ‘uni’ for uniform, ‘rand’ for uniform plus random or ‘shift’ for slightly shifted uniform.
- ‘q’ is the number of intervals (in ‘rand’ and ‘shift’ an optional key ‘shf’ can be used to define how much the values  $d_j$  in  $ds$  are shifted).

**‘epsilonalga’:** **epsilon used to set alphas and gammas according to avdeltar.** ‘epsilonalga’ must be lower than  $1/k$  where  $k$  is the number of regions of the partition with the highest number of regions (see (67), (96) and (97) of Supplementary file S1 of <https://doi.org/10.1038/s41540-017-0044-x>).

**Warning:** Avoid a very low ‘epsilonalga’ as it can cause numerical issues in the solver, e.g. the solver might violate a constraint by close to tolerance values.

**‘scalebs’:** **Numerical value to scale bounds  $w_l$ ,  $w_u$  and  $W$**  The bounds  $w_l$ ,  $w_u$  and  $W$  are multiplied by ‘scalebs’ so that they are not tight.



## FN with intermediate states

In a FN with intermediate states, the variables of the different periods and macroperiods can be accessed through the prefixes  $q$  and  $Q$  respectively, where a macroperiod is a sequence of consecutive periods with same net structure. The prefix  $q_i$  refers to variables of the period  $i$  and the prefix  $Q_j$  refers to variables of the macroperiod  $j$ . The prefixes  $q$  and  $Q$  are set in `LFlexN.qpre` and `LFlexN.Qpre` respectively.

The following fields can be used to specify a FN with intermediate states:

**'solver': String denoting the solver to use**

See the 'solver' field in [FN specification](#).

**'mapers': Links for markings and actions between consecutive periods**

Example:

```
'mapers': [{ 'net': net1, 'thetas': 4*[0.05]},
             { 'outinm': fromlto2m, 'outina': fromlto2a, 'net': net2, 'thetas': 3*[.
↪1]}],
```

Each component of the list is associated to a macroperiod. In the example, 'net': net1 specifies the net structure for the first macroperiod (net1 is a dictionary specifying a FN), and 'thetas': 4\*[0.05] specifies that the macroperiod has 4 periods of length 0.05. In the second macroperiod the net structure is given by net2 and there are 3 periods of length 0.1.

The link between final markings(actions) to initial markings(actions) of macroperiods is given by 'outinm' and 'outina' which specify the required relationships as list of expressions, e.g.:

```
fromlto2m = ["3*m['p1'] == m0['PrX']",
             "m['p2']-1 <= m0['p2']", "m0['p2'] <= m['p2']+1",
             "m0['p3']== 9"]
fromlto2a = ["at['t1']==a0['t1']", "at['t2']>=a0['t2']", "at['t2']<=a0['t2']"]
```

**'extracons': List of extra constraints that must be satisfied**

These constraints will be included in the programming problem to be optimized.

```
'extracons': ["q0_m0['p1']<= 2", "Q1_avm['p3']>= 10"].
```

---

**Note:** The extra constraints of each particular net are ignored, i.e. only extra constraints in this list will be considered.

---

**'obj': Dictionary with the objective function and its sense**

Example:

```
'obj': {'f': "sum((Q1_m[p]-2)*(Q1_m[p]-2) for p in ['PrX','p2'])", 'sense': 'min'}
```

**'options': Dictionary that specifies options**

The keys of this dictionary are the same as the ones of the 'options' dictionary for FNs without intermediate states plus the following:

**'plotres': Boolean.** If True plot the variables specified in 'plotvars'.

**'plotvars': Dictionary specifying the variables to plot** Each key of plotvars is a variable name or 'evm' or 'eva'. Variable names are plotted as steps (stair-like plot). 'evm' and 'eva' plot the marking and actions evolution, i.e. lines connecting initial and final markings(actions) are plotted.

Example:

```
'plotvars': {'l0':['t1','t2'], 'evm':['p1','p2']}
```

---

**Note:** The 'options' of each particular net are ignored, i.e. only the options in this dictionary are considered.

---

FNs with Model Predictive Control (MPC) are specified by dictionaries similar to the ones discussed above. The MPC parameters are set in a dictionary 'mpc' located the 'options' field and the variables of the different intervals (or periods) of the control horizon can be accessed through the prefixes qi\_ and Q0\_.

---

**Note:** That the net structure of a FN with MPC is not allowed to change over time. Hence, there is only one macroperiod Q0\_.

---

## 5.1 Options

As in a FN with intermediate states, the 'options' dictionary can contain the fields 'plotres' and 'plotvars' which have the same meaning. Particular fields of the 'options' dictionary are:

**'antype': This field must be set to 'mpc'**

**'mpc': Dictionary specifying the MPC parameters**

The keys of this dictionary are:

**'firstinlen': Length of the first interval (or period) of MPC** This value is also the minimum time length allowed for flexible intervals.

**'numsteps': Number of optimization steps** This value is equivalent to the number of FNs to optimize. It holds that the final time of the optimization is equal to 'numsteps'\*'firstinlen'.

**'maxnumins': maximum number of intervals (including the first interval)** Horizon in terms of number of intervals. It must to be greater than or equal to 1.

**'flexins': Boolean** True if you want intervals to stretch until they cover up to the final time. False implies that the length of the intervals is equal to 'firstinlen'.

## 5.2 Objective function

The objective function is defined as a list of objective functions. The  $i$ th component of such a list (which in Python has index  $i-1$ ) is used when the current step being optimized has  $i$  intervals. The objective function for several intervals can refer to the variables of different periods and to the overall macroperiod (there is only one macroperiod as the net structure is not allowed to change) through the prefixes  $q_i$  and  $Q0$ . For instance, let us define:

```
obj1in = {'f': "m['A']", 'sense':'min'} # Objective for one interval
obj2in = {'f': "q0_m['A']+q1_m['B']+Q0_m['C']", 'sense':'min'} # Objective for two_
↪intervals
obj = [obj1in, obj2in] # List of objective functions
```

Then, the objective function can be set as:

```
'obj': obj
```

If ‘maxnumins’ is equal to 1, the objective function does not need to be a list.

## 5.3 Extra constraints

Both time independent and time dependent constraints can be included as fields in the dictionary that specifies the FN. Similarly to the objective function, these constraints are specified as a list. The  $i$ th component of the list is a list of constraints that is considered when the current step being optimized has  $i$  steps.

Time independent constraints can be included as:

```
'extracons': [
    ["alphan['R1']==0", "avm['p7']>=1.0"], # List of constraints when the step has_
↪one interval
    ["q0_alphan['R1']==0", "q1_m['p2']<=6.0"] # List of constraints when the step has_
↪two intervals
]
```

Notice that if the control horizon is 1, i.e. there is always one interval then ‘extracons’ is still a list of lists, e.g.

```
'extracons': [{"alphan['R1']==0"}]
```

Time dependent constraints can be included as:

```
textracons: [
    [
        {'cond': "0.05<=time and time<=0.5", 'cons': "l0['F']==0"},
        {'cond': "0.51<=time", 'cons': "l0['F']==1"}
    ], # List of constraints when the step has one interval
    [
        {'cond': "time<=0.5", 'cons': "q0_l0['F']+q1_l0['F']==0"},
        {'cond': "time>=0.51", 'cons': "q0_l0['F']==1"},
        {'cond': "time>=0.51", 'cons': "q1_l0['F']==2"}
    ], # List of constraints when the step has two intervals
]
```

The constraint ‘cons’ is considered when the condition in ‘cond’ is True where *time* is the starting time of the step.



## 5.4 Resets

Both the initial marking and the initial actions of a step can be reset if a given condition over the state of the previous interval is satisfied. The conditions and the reset constraints are specified in a list of dictionaries. Example:

```
resets = [
  {'cond': "m['A']>=10 or avl['T']>=41",
   'm0cons': ["m0['A'] == 0.5*m['A']", "m0['B'] == 0.4"],
   'a0cons': ["a0['R'] == 2*at['R']", "a0['F'] == 5.0"]},
  {'cond': "0.2<=time and 100<=m['A']",
   'ntimes': 1,
   'm0cons': ["m0['A'] == m['A']+5", "m0['B'] == 0.5*m['B']"]}]
```

where *time* is the starting time of the step. The variables that can be used in the conditions are: *time*, *m*, *avl*, *at*, *l* and *avl* which respectively refer to the time, final marking, average marking, number of remaining actions, intensity and average intensity of the previous interval.

‘m0cons’ and ‘a0cons’ are used to reset initial markings and actions respectively. ‘m0cons’ can only include the variables *m* (final marking of previous interval) and *m0* (initial marking of current step). Similarly, ‘a0cons’ can only include *at* and *a0*. If ‘ntimes’ is included in the reset dictionary, the reset will take place at most ‘ntimes’.



---

## Auxiliary functions

---

In order to facilitate the specification of FNs, *fnyzer* provides:

- A function to convert *Cobra* models into FN dictionaries.
- A class to approximate nonlinear dynamics by piecewise linear functions.

### 6.1 Cobra models

The function *cobra2fn()* converts a *Cobra* model into a FN specification. It is defined as:

```
def cobra2fn(comodel, knockouts = [], solver = 'cplex'):
```

where *comodel* is a *Cobra* model, *knockouts* is a list of names of genes to be knocked out before generating the FN, and *solver* is the name of the solver to be used in future analysis.

The following lines assume that *Cobra* is installed in your system and that the file *MODELXXX.xml* is a model in SBML format.

```
from fnyzer import cobra2fn
import cobra
model = cobra.io.read_sbml_model('MODELXXX.xml')
fndic = cobra2fn(model)
```

*fndic* is now a dictionary containing the FN specification that can be edited and analyzed, see [Getting started](#).

### 6.2 Nonlinear dynamics

The class *regsheap*, see file *regsheap.py*, can be used to create and store in a priority queue regions that represent a piecewise linear approximation of a nonlinear function. Such regions comply with the format required for FNs.

The parameters required are the following (see below for an example):

**'proregs': Dictionary with the initial borders of the regions**

**'func': Nonlinear function to be approximated**

The dynamics of the regions are given by 'func' which is approximated by a linear function. The function 'func' takes a matrix as an argument, the first column of the matrix is a constant vector 1 and the rest of the columns of the matrix are the parameters of the function sorted alphabetically.

The linear dynamics of each regions is obtained by linear regression (ordinary least squares). Once the regions are created, the region with maximum msr (mean squared residual) is split. A warning is reported if the predicted dynamics of a generated region can get negative values (such a region should be further split to get better precision and avoid negative values).

**'parnames': Dictionary with names of net elements and parameters**

Dictionary that associates names of the elements connected with the intensity handler with the names of the parameters used in the linear expression:

parnames['lap'] refers to the name of the variable to which the linear regression is assigned, usually an intensity sarc.

parnames['place'] refers to the name of the parameter associated with the intensity edge ('place', 'shandler').

**'maxregs', 'maxmsr': Maximum number of regions and mean squared residual**

The regions are further split until 'maxregs' regions are created or the msr (mean squared residual) of every region is lower than 'maxmsr'.

**'sampoints': Number of points used to sample the function**

Number of points to be used in each dimension for the linear approximation (regression).

**'pref': The names of the regions are prefixed with 'pref'**

The use of *regsheap* is exemplified by the FN *hillpwnet* in the file [hillpwnet.py](#). The net models a reaction with a rate that follows a particular Hill equation. In order to analyze the FN, download the file in your working directory and execute the following line in a shell to get steady state information:

```
$ fnyzer hillpwnet hillpwnet
```

Execute the following to get a trajectory of the FN with MPC:

```
$ fnyzer hillpwnet hillpwnetmpc
```

---

**Note:**

- The function to be approximated is assumed to depend just on the marking  $m$ .
  - The resulting approximation is assigned to just one intensity arc.
-