fiabilipy Documentation

Release 2.4

Akim Sadaoui, Simon Chabot

January 15, 2014

Contents

CHAPTER 1

Overview

fiabilipy is a python package providing some functions to learn engineering reliability at university. With this package, one can build easily some components, put them together to build a complete system and finally evaluate some metrics (reliability, maintainability, Mean-Time-To-Failure, and so on).

fiabilipy also provides tools to describe a system by a Markov process and evaluating the probability of being in a given state.

Installing / Upgrading Instruction on how to get fiabilipy

Tutorial Start here to learn how to make your first system.

Examples Some examples on how to perform specific tasks.

API Documentation The complete API documentation, organized by modules.

CHAPTER 2

Contributing

This software is free - as in a speech - therefore, all contributions are encouraged, from minor tweak to pull request or just a little message ;)

CHAPTER 3

Changes

See the *ChangeLog* for a full list of changes offered by each version.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

4.1 Installing / Upgrading

fiabilipy is in the Python Package Index, so it should be quite easy to install it. Multiple solutions are offered to you.

The main dependencies of fiabilipy are numpy and scipy and should be installed before.

4.1.1 Installing with pip

To install fiabilipy:

```
$ pip install fiabilipy
```

To get a specific version:

```
$ pip install fiabilipy==2.2
```

To upgrade to the last version:

```
$ pip install --upgrade fiabilipy
```

4.1.2 Installing on Archlinux

If you are using the Archlinux GNU/Linux distribution, an AUR package has been made. You can find it here. The major interest of using this package, is that it gets upgraded whenever a new version is available and you don't have to manage dependencies.

4.1.3 Installing on Windows

4.1.4 Installing from source

If you would rather install directly from the source (to contribute, for instance), check out for the latest source on our repository:

```
$ hg clone https://chabotsi.no-ip.org/hg/utc/fiabilipy/
$ cd fiabilipy
$ python setup.py install
```

4.2 Tutorial

This tutorial is intended as an introduction to fiabilipy. You will learn to build some components, to put them together and to compute some reliability metrics. You will also learn how to use the markov representation.

4.2.1 Prerequisites

Before you start, be sure fiabilipy is well installed on your system. In the python shell, the following import should run without raising an exception:

```
>>> import fiabilipy
```

If an exception is raised, check your *installation*.

4.2.2 Topics

How to build a system This tutorial shows you how to build components and how to gather them to build a system. You also learn how to access to useful reliability metrics.

How to describe a system by a Markov process This tutorial shows you how to describe a system with a markov process. Then, it shows you how to compute the probability of being in a given state (such as *insufficient*, *damaged*, *nominal* and so on).

How to build a system

A system is built by putting components together. So, let's have a look on how to build components.

Building components

A component is defined as an instance of the `Component` class having a *constant* reliability rate, let's say $\lambda = 10^{-4}h^{-1}$.

```
>>> from fiabilipy import Component
>>> from sympy import Symbol
>>> t = Symbol('t', positive=True)
>>> comp = Component('C0', 1e-4)
```

You have successfully created your first component. *C0* is the name of the component ; naming your components will be useful to draw diagrams later.

You can access to some useful information about your component, such as the MTTF (Mean-Time-To-Failure), the reliability etc.

```
>>> comp.mttf
10000.0
>>> comp.reliability(1000)
0.904837418035960
```

```
>>> comp.reliability(t)
exp(-0.0001*t)
>>> comp.reliability(t=100)
0.990049833749168
```

Gather components to build a system

Now you can build components, let's gather them to build a system. A system is described as a graph of components. There are two special components used to materialize the entry and the exit of the system : *E* and *S*. They are compulsory, don't forget them.

For instance, you could create a simple series system of two components as follow:

```
>>> from fiabilipy import System
>>> power = Component('P0', 1e-6)
>>> motor = Component('M0', 1e-3)
>>> S = System()
>>> S['E'] = [power]
>>> S[power] = [motor]
>>> S[motor] = 'S'
```

Once your system is created, you can access to the data you want, such and MTTF, reliability, etc.

```
>>> S.mttf
1000000/1001
>>> float(S.mttf)
999.000999000999
>>> S.reliability(t)
exp(-1001*t/1000000)
```

An example of complex system Let's build the following system :

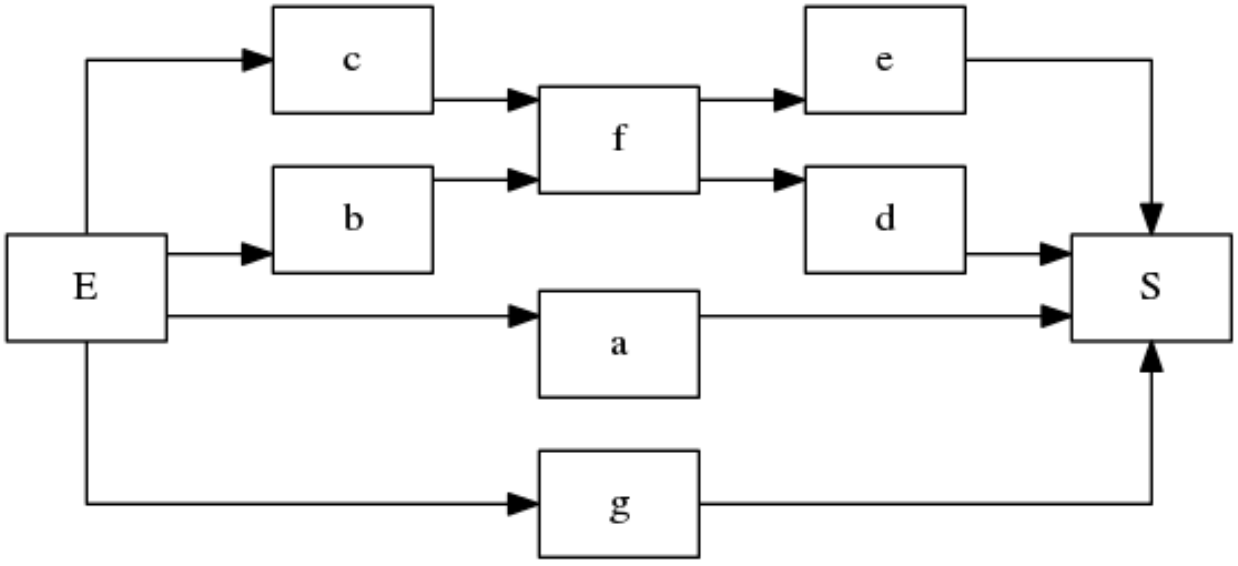


Figure 4.1: A example of _complex_ system

```
>>> a, b, c, d, e, f, g = [Component('C%i' % i, 1e-4) for i in xrange(7)]
>>> S = System()
>>> S['E'] = [a, b, c, g]
>>> S[a] = S[g] = S[e] = S[d] = 'S'
>>> S[b] = S[c] = [f]
>>> S[f] = [e, d]
```

And, you can easily access to the data you want, as previously.

```
>>> S.mttf
331000/21
>>> S.reliability(t)
13*exp(-t/2000) - 12*exp(-t/2500) - exp(-t/5000) - 6*exp(-3*t/5000) + 2*exp(-t/10000) + 4*exp(-3*t/10
```

As you may see, even if the system is complex, it is quite easy to describe it with fiabilipy.

Draw graphics

Now you know how to build system with ease, let's draw some graphics. For instance, reliability versus time.

The first thing to do, is to import the `pylab` module, which provides a lot of function to do mathematical stuff *and* to draw graphics:

```
>>> import pylab as p
```

Now, let's define a simple parallel system with two components.

```
>>> a, b = Component('a', 1e-4), Component('b', 1e-6)
>>> S = System()
>>> S['E'] = [a, b]
>>> S[a] = S[b] = 'S'
```

In order draw the graphic, we need a time range of study, for instance, from $t = 0$ to $t = 200$, by steps of 10 (the unit of time is the one you choose). Once the time range is defined, we compute the reliability of each time step:

```
>>> timerange = range(0, 20000, 100)
>>> reliability = [S.reliability(t) for t in timerange]
```

To finish, you only have to plot:

```
>>> p.plot(timerange, reliability)
>>> p.show()
```

You can admire the result.

How to describe a system by a Markov process

fiabilipy enables the users to model a system trought a Markov process. This tutorial aims to introduce how fiabilipy handles the Markov modelisation. This introcduction is done using a example.

Let's say we have a parallel-series system, represented by the following figure.

We are interested by the probabilities for the system to be :

- in normal state (i.e. every component works)
- in damaged state (i.e. one or more component don't work, but the system does work)
- in faulty state (i.e. the system doesn't work at all)

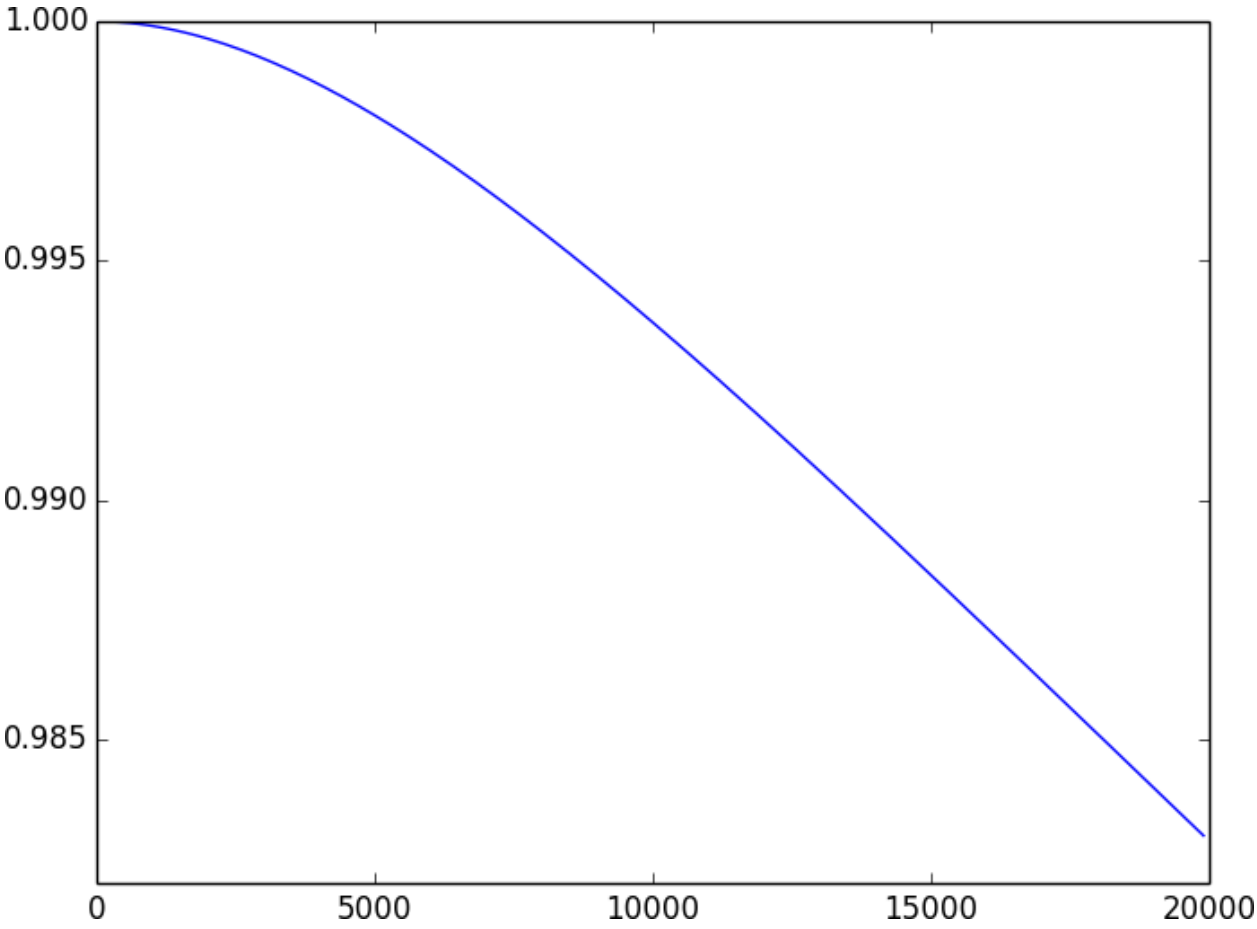


Figure 4.2: The reliability graphic

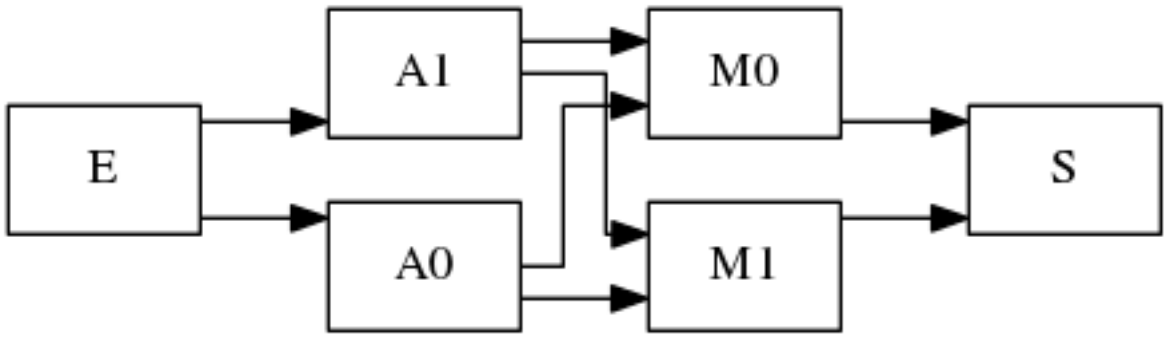


Figure 4.3: A parallel-series system

- in available state (i.e. the system is not faulty)

Compoment and process initialisation

To begin, let's start building the components:

```
>>> from fiabilipy import Component, Markovprocess
>>> A0, A1 = [Component('A{}'.format(i), 1e-4, 1.1e-3) for i in xrange(2)]
>>> M0, M1 = [Component('M{}'.format(i), 1e-3, 1.2e-2) for i in xrange(2)]
>>> components = (A0, A1, M0, M1)
```

To initialize the process, we need to give to fiabilipy the list of the components and the initial states probabilities. A state is defined by a number which, in base 2, says if the ith component is working or not. So, with 4 components, we have $2^4 = 16$ possible states. The following table represents the possible stats (W stands for *working* and N for *not working*).

Table 4.1: states table

state	A0	A1	M0	M1
0	W	W	W	W
1	W	W	W	N
2	W	W	N	W
3	W	W	N	N
...	...	...	...	...
15	N	N	N	N

Now, let's assume that, at $t = 0$, the probabilities of the system to be in state 1 is 0.9 and state 2 is 0.1, thus we have:

```
>>> initstates = {0: 0.9, 1:0.1}
>>> process = Markovprocess(components, initstates)
```

Working states definition

Now we have initialized our Markov process, we have to define the states to be tracked. This is done a writing a function which return *True* if the given state is tracked *False* otherwise. The functions will be called with one argument, let's say *x*. This variable is a boolean array, the ith case is *True* if the ith component is currently working, *False* otherwise.

Ok, let's define the states we want to track.

Normal state In this state, *every* component has to work. So, in python is can be written as:

```
>>> def normal(x):
...     return all(x)
```

This function returns *True* if every single item of *x* is *True*.

Available state In this state, the system is available. So, there exists a path of working components for *E* to *S*. That is to say either A0 or A1 are working and M0 or M1 are. So, the function describing the faulty state may be:

```
>>> def available(x):
...     return (x[0] or x[1]) and (x[2] or x[3])
```

Damaged state Actually, when you have described what the available state is, you have made the harder part. Because, the other states can be described as combinations of it. For instance, the system is damaged when the system is in available state *and not* in the normal state. Therefore:

```
>>> def damaged(x):
...     return available(x) and not(normal(x))
```

Faulty state The system is faulty when not available. So, it's quite simply to describe:

```
>>> def faulty(x):
...     return not available(x)
```

Compute the probabilities

Now you have written the functions describing the states, it is really simple to ask fiabilipy the probabilities you want. For instance, to know the probability of the system being available at $t = 150h$, simply write:

```
>>> process.value(150, available)
0.97430814090407503
```

At $t = 1000h$, the probability that every component is still working is:

```
>>> process.value(1000, normal)
0.30900340684254302
```

Drawing plots Now you are able to compute the probabilities you want, for the states you want, for the time you want, let's plot those probabilities. The following code gives you a example of how to plot the variation of the probabilities.

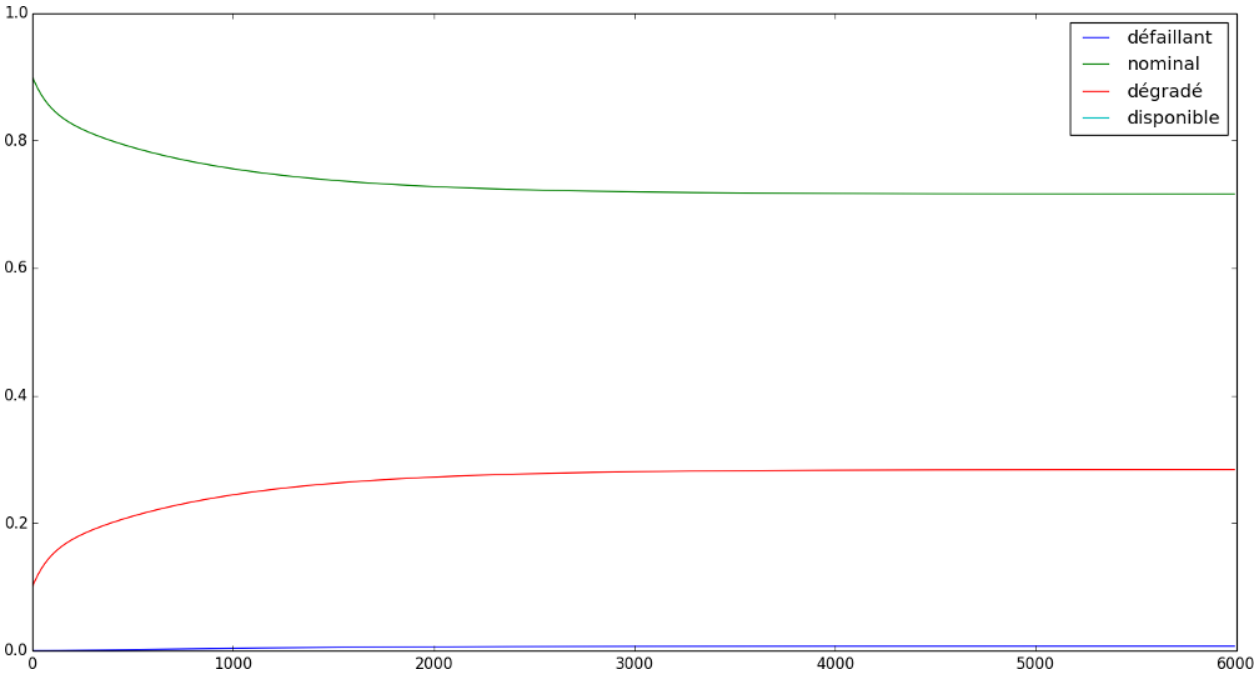
```
>>> import pylab as p
>>> states = {u'normal': normal,
...           u'available': available,
...           u'damaged': damaged,
...           u'faulty': faulty,
...          }
>>> timerange = range(0, 6000, 10)
>>> for (name, func) in states.iteritems():
...     proba = [process.value(t, func) for t in timerange]
...     p.plot(timerange, proba, label=name)
>>> p.legend()
>>> p.show()
```

And, this code gives you the following figure:

4.3 Examples

This page gathers different examples of python scripts using fiabilipy. If you are using fiabilipy, feel free to send us your scripts, we will add them to this page !

- *Voters intersection*



4.3.1 Voters intersection

The older the voters are, the less reliable they are. In their youngness, voters are more reliable than single components. The goal of this example is to find when a voter starts to be less reliable than a single component.

Step by step

Let's start with a simple 2/3 voters. It's really easy to get its reliability. First, we have to import the functions and classes we will use.

```
>>> from fiabilipy import Voter, Component
>>> from sympy import Symbol
```

Let's build the voter, with an unknown reliability λ.

```
>>> l = Symbol('l', positive=True, null=False) #Lambda
>>> t = Symbol('t', positive=True) #our time variable
>>> comp = Component('C', l)
>>> voter = Voter(comp, 2, 3)
```

Here is the voter, now let's get its reliability.

```
>>> voter.reliability(t)
3.0*(1 - exp(-l*t))*exp(-2*l*t) + 1.0*exp(-3*l*t)
```

To have a polynomial expression, we substitute $\exp(-\lambda t)$ to x. Once more, this is easy in python...

```
>>> from sympy import exp
>>> x = Symbol('x')
>>> voter.reliability(t).subs(exp(-l*t), x).nsimplify()
x**3 + 3*x**2*(-x +1)
```

Using this notation, the reliability of a single component is x. So to find when the given voter is equivalent to the single component, we simply have to solve $x^3 + 3x^2(-x + 1) - x = 0$.

```
>>> from sympy import solve
>>> crossing = (voter.reliability(t) - comp.reliability(t)).nsimplify()
>>> solve(crossing.subs(exp(-l*t), x))
[0, 1/2, 1]
```

And, the task is done.

The complete code

The code below gives a generic function to solve this problem.

```
from fiabilipy import Voter, Component
from sympy import Symbol, solve, exp

def voterintersection(M, N):
    assert 1 < M < N, 'the given voter is not real'

    l = Symbol('l', positive=True, null=False)
    t = Symbol('t', positive=True)
    x = Symbol('x')

    comp = Component('C', l)
    voter = Voter(comp, M, N)

    crossing = (voter.reliability(t) - comp.reliability(t)).nsimplify()
    roots = solve(crossing.subs(exp(-l*t), x))

    return roots
```

4.4 API Documentation

4.4.1 `system` – System building and description

`Component` – Compoment building

`Voter` – Voter building

`System` – System building and description

4.4.2 `markov` – Markov system building and description

4.5 ChangeLog

4.5.1 V2.4

- A better cache system.
- A documentation in English, using Sphinx.
- Add some metrics computation. One can compute maintainability, and therefore availability, for complex systems and voters.
- An Archlinux package is made.

- The code has been cleaned. (thank you pylint ;))
- Symbolic computation can be performed, using `sympy`

```
>>> from sympy import symbols
>>> from fiabilipy import Component
>>> l, t = symbols('l t', positive=True)
>>> a = Component('a', lambda_=l)
>>> a.reliability(t)
exp(-l*t)
```

4.5.2 V2.3.1

- Update the module name from `fiabili` to `fiabilipy`.

4.5.3 V2.3

- Create a pypi package.
- Update the documentation.
- A Markov graph can be drawn.

4.5.4 V2.2

- Add a `markov` module.

4.5.5 V0.2

- Some metrics are cached to faster the computation (MTTF, MTTR (Mean-Time-To-Repair), etc).
- A documentation is started (in French... sorry).

4.5.6 V0.1

- *Show time*.
- System, Component and voter can be built.
- Reliability can be computed.
- MTTF can be computed.
- Compute the minimal cuts of order 1 and 2.