FluctuProtST: a tool to model and simulate protein fluctuations within ODE models of signal transduction [User Guide]

François Bertaux, Inria Paris-Rocquencourt
francois.bertaux@inria.fr

Version 1.2, March 2015

Signal transduction is traditionally modeled by ODEs describing reactions between signaling proteins. However the levels of those proteins naturally fluctuate because of stochastic gene expression, promoting cell-to-cell variability in signaling response even in absence of intrinsic noise in signal transduction *(Colman-Lerner et al., Nature, 2005; Spencer et al., Nature, 2009)*. Modeling those fluctuations is therefore needed to improve the prediction capabilities of signal transduction models.

FluctuProtST is a tool that allows a user to describe and then simulate a model of signal transduction that include fluctuations of signaling proteins. It is essentially a generalization of the modeling and simulation algorithms used in *(Bertaux et al., PLoS Comp. Bio., 2014)* to model stochastic gene expression within an ODE model of TRAIL-induced apoptosis. More precisely, fluctuations of native proteins are modeled with the two-state transcriptional bursting model for describing stochastic gene expression *(see for example Singh et al., Mol. Sys. Biol., 2012)* and coupled with ODEs describing signaling reactions. Therefore, the resulting model is an hybrid stochastic/deterministic model.

FluctuProtST uses *C++* code from *Numerical Recipes (Press et al., Cambridge University Press, 2007)* to perform Gillespie simulations and ODEs integration.

General workflow

The utilization of *FluctuProtST* can be decomposed in two steps:

- Modeling step:
  - define your model in *Python*

  - when the model is defined, *FluctuProtST* can automatically generate *C++* code to use for simulating the model
- Simulation step:
  - eventually customize the *C++* generated code to your particular needs
  - compile and run the *C++* code

The two following sections describe in details how to perform those two steps.

Modeling workflow

The first step in using *FluctuProtST* is to describe your model using simple *Python* declarations, in the spirit of *PySB (Lopez et al., Mol. Sys. Biol., 2013)*. Those declarations will either describe signaling reactions or the fluctuations of a given protein. To start describing the model, simply write the following:

```
import FluctuProtST as fpst
myModel = fpst.FluctuProtSTModel("MyModel")
```

Describing protein fluctuations

Let's briefly recall how protein fluctuations are modeled. Figure 1 describes the model structure and its parameters. We sometimes call this model a *stochastic protein turnover model* to highlight the importance of protein degradation in shaping fluctuations.

Six rate parameters (k_{on}, k_{off}, k_{sm}, γ_m, k_{sp}, γ_p) fully define a protein fluctuation model. Another, equivalent way to define a fluctuation model (and in our opinion more natural) is to provide *EP* and *EM* (mean protein and mRNA levels), *HLP* and *HLM* (the protein and mRNA half-lives), and *Ton* and *Toff* (the mean durations of the promoter *On* and *Off* states).

Thus in the tool a native protein (hence naturally fluctuating) can be defined by expliciting those 6 values :

```
myModel.addNativeProtein(name="ProtA",EP=1000.,HLP=27.,
                         EM=17.,HLM=9.,Ton=0.1,Toff=2.5)
```

Alternatively, if you know the protein is long-lived (hence degraded only by dilution), you can decide to use a standard fluctuation model, as was done for most proteins in *(Bertaux et al., PLoS Comp. Bio., 2014)*:

```
myModel.addNativeProteinStdFluct(name="ProtB",EP=5000.,
                                 dilutionHalfLife=27.)
```

This function assumes that the time unit of the model is hours.

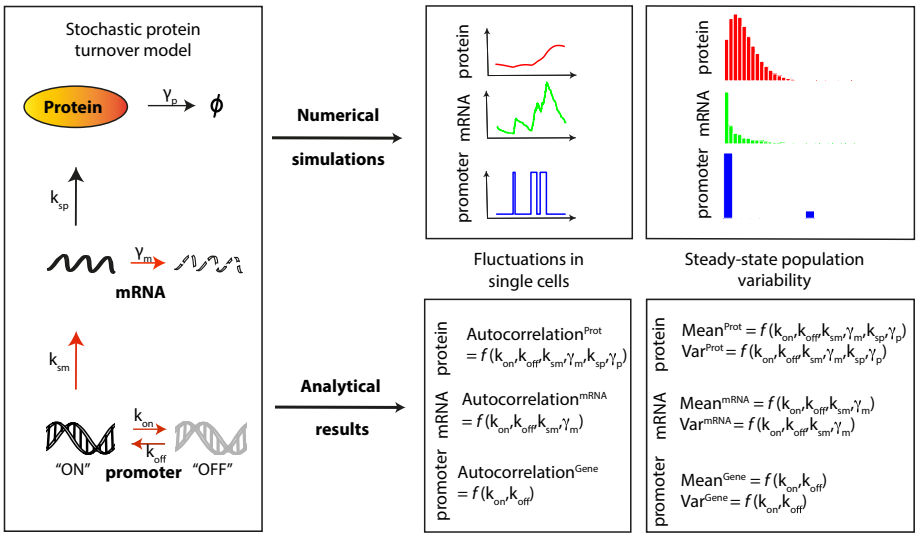

Figure 1: Description of protein fluctuation models, and how they are either numerically simulated or studied analytically.

Describing signaling reactions

Reactions between protein species can be defined the following way:

```
myModel.addReaction(name="SignalBindsA",reactants=["Signal","ProtA"],
                    products=["Signal_A"],rate=("kb",0.0001))
```

Reversible reactions can also be defined with a single declaration:

```
myModel.addReversibleReaction(name="SignalBindsA",
                              reactants=["Signal","ProtA"],
                              products=["Signal_A"],
                              rates=[("kb",0.0001),("ku",0.01)])
```

Finally, a 3-reactions *catalysis* reaction scheme (binding of a catalyst with a substrate, unbinding or transformation of the substrate with release of the catalyst) can be defined with a single declaration:

```
myModel.addCatalyticReaction(name="SignalActivatesA")
                             susbtrate="ProtA",
                             catalyst="Signal",
                             product="ActivatedProtA",
                             rates=[("kb",0.0001),("ku",0.01),("kc",1.)])
```

In each case, elementary reactions are assigned mass-action kinetics. Also, for each of species defined, if it not known as a *NativeProtein*, it will be added to the model as a *ModifiedProtein*, i.e. a non-native form. Thus it imposes to start by defining the fluctuations of native proteins before defining signaling reactions involving them.

Defining degradation of non-native forms

By default, non-native forms appearing in the model are assumed not to be degraded. Although one can use *addReaction()* to define a degradation reaction, a simpler way is to use *setAllModifiedProteinDegradation* or *setModifiedProtein-Degradation*:

```
myModel.setAllModifiedProteinDegradation(halfLife=10.)
myModel.setModifiedProteinDegradation(name="Signal",
                                      halfLife=0.5)
myModel.setModifiedProteinDegradation(name="ActivatedProtA",
                                      halfLife=5.)
```

Simulation workflow

When the model is defined, use the following line to generate *C++* code for simulating the model:

```
fpst.buildCppFromModel(model=myModel,
                               targetFolderPath="myModelCppCode")
```

C++ code will be written into the target folder. From there, compilation should be straightforward after importation of the folder as a project in your favorite *C++* IDE.

For those who have *Qt* and *make* installed, a very efficient way to proceed is:

```
qmake -project; qmake;      # generates Makefile
make;                       # compiles
```

Customization of the simulation code

To customize your simulation of the model, edit *main.cpp*. The classes you would be likely to play with are *CellState*, *HybridSimulator* and *ModelParameters*. By default, *main.cpp* provides a very simple utilization scenario, which is relatively self-explanatory.

- Loading parameters and constructing the simulator:

```
ModelParameters* modelParameters = new ModelParameters () ;
HybridSimulator* hybridSimulator = new HybridSimulator (modelParameters) ;
```

- Constructing a cell, and simulate it for some time to reach the steady-state protein level distributions:

```
CellState *cell = new CellState (modelParameters) ;
hybridSimulator->simulate ( cell , 7.*24. ) ;
```

- Apply a stimulus and simulate the cell response

```
cell->set_NameOfSpecies_Level (0.1) ;
hybridSimulator->simulate ( cell , 12. ) ;
```

- Check the response of the cell

```
cout << "Level of _NameOfSpecies_ after stimulus = " ;
cout << cell->get_NameOfSpecies_Level () << endl ;
```

Examples

Two examples are provided with *FluctuProtST*. The first one is a simple, toy model (see *model_ToyExample.py*). The second one (see *model_hEARM.py*) is the model of TRAIL-induced apoptosis from *Bertaux et al., PLoS Comp. Bio., 2014* and extending the ODE model of *Spencer et al., Nature, 2009*. The latter also illustrate how the *C++* code generated automatically can be customized to perform specific in-silico experiments with the model (compare folders *hEARM_cpp_code_generated* and *hEARM_cpp_code_customized*). More precisely, custom functions have been defined in *Experiments.hpp* and are used to repeat the simulations needed for producing *Figure 3* of the paper. Matlab code to produce the plots is provided in *hEARM_cpp_code_customized/results-analysis*.