# QualitativeModelFitting

***Release 0.0.1***

**Dec 12, 2019**

# Contents

*QualitativeModelFitting* (*qmf*) is a package designed for validating a model against arbitrary observations. The concept stems from that of unit testing in software development. Using *qmf*, each part of a model is tested by statements derived from literature or in house data. These statements are encoded as a *qmf* input string which is used together with an antimony string as input to the `qualitative_model_fitting.Runner` class.

Click below for more information on usage.

# CHAPTER 1

# The qmf input string

*qmf* defines its own syntax for retrieving user input. In *qmf* this is known as an *input* or *observation* string. An input string is divided into blocks and each block has a type. For now, there are only *timeseries* blocks and an *observation* block. You can have as many *timeseries* blocks as you like, but there must only be one *observation* block.

## 1.1 The timeseries block

This is where you define the timeseries that you can use in later comparisons. Each timeseries block you specify requires a separate time series simulation with its own independent variables (i.e. starting conditions) and therefore the more you have, the longer the programs execurion time.

The syntax of a *timeseries* block looks like this:

```
timeseries name {component1=amount1, component2=amount2, ...} 0, 100, 101
```

Spaces are ignored, so:

```
timeseries name {
    component1=amount1,
    component2=amount2,
    ...} start, stop, num
```

is syntactically equivalent and sometimes preferred, when a *timeseries* has lots of independent variables. The *name* argument is a handle for this timeseries and is used later within the observation block to refer to it. The final three arguments are *start*, *stop* and *num* which are the *start* and *stop* points of numerical integration and *num* how many equally spaced time points to have between them.

### 1.1.1 Examples

```
timeseries SInactive {S=0} 0, 50, 51
timeseries SActive {S=1} 0, 50, 51
```

These two timeseries encode the two situations where a hypothetical stimulus *S* is on in *SActive* or off in *SInactive*. Both timeseries will be integrated from *0* to *50* using a wrapper around tellurium and roadrunner packages.

## 1.2 The Observation Block

As the name suggests, this is where we define our observations. Observations can be one of several types. The simplest look like the following:

```
name: statement
```

where

- *name*: The name of your observation. Arbitrary.
- *statement*: A binary comparison instruction

The *statement* has the following form:

- *clause operator clause*

Where:

- *operator*: One of the comparison operators (>, <, >=, <=, ==, !=).
- *clause*: an entity for comparison (see below)

## 1.3 Clause

### 1.3.1 Constants and expressions

A *clause*, in analogy to part of a sentence, can have one of several forms. At its simplest, a clause can be a constant value or a numerical expression.

```
0
5*10
4 + 4*9
```

The usual precedent rules in math are applied correctly.

### 1.3.2 Model variables

More often, we want a particular model variable at a particular time:

```
model_component[timeseries_name]@t=x
```

Which will resolve to a single number representing the amount of *model_component* in condition *timeseries_name* at time *x*. For example we could do:

```
A[SActive]@t=0
```

Which returns that scalar number. Sometimes we do not want a scalar but the amount of a variable between two time points.

```
model_component[timeseries_name]@t=(x, y)
```

Which be resolved to a vector of numbers representing the amount of *model_component* in condition *timeseries_name* between the time ranges of *x* and *y*. Since a vector cannot directly be compared with a scalar, to use a range of values in a comparison we need to use a function (see below).

## 1.4 Functions

Functions can take two forms:

- *Type1*: Those which tell the *Runner* how to make a comparison between scalar and vector
- *Type2*: Those which convert vectors to scalars prior to making the comparison.

These two function types have a slightly different syntax:

Type1:

```
name: function(clause operator clause)
```

Type2:

```
name: function(clause) operator clause
```

**Note:** The *Type1* function type takes as argument the whole *clause operator clause* statement while the *Type2* function takes only a clause as argument.

**Note:** Point 2 here assumes that the first *clause* is the time interval clause and the second is a scalar.

**Note:** Comparing a vector with another vector (i.e. element wise) is not yet supported.

### 1.4.1 Type1 functions

There are two *Type1* functions: *any* and *all* which are analogous to Python's and *numpy any* and *all* functions. If you use the *all* function when comparing a vector and scalar, the function will return *True* if all of the elements in the vector meet the condition set by the operator and the other clause. The *any* function on the other hand will return True if any of the elements in the vector meet the conditions set by the operator and the other clause.

### 1.4.2 Type1 Function Examples

All of *A* in the *SActive* timeseries between *0* and *50* are *greater than* the amount of *A* in the *SInactive* timeseries at time 25.

```
all(A[SActive]@t=(0, 50) >  A[SInactive]@t=25)
```

If *A* in the *SActive* timeseries at time *0* are *greater then* any of *B* between the bounaries of *13* and *19*, return *True* else *False*

```
any(A[SActive]@t=0 >  B[SActive]@t=(13, 19))
```

### 1.4.3 Type2 functions

Type 2 functions currently include:

- mean
- min
- max

Which are self explainatory in what they do.

### 1.4.4 Type 2 function examples

The **mean**, **maximum** or **minimum** (respectively) of *A* in the *SActive* time series between time *0* and *50* is *greater than* the amount of *A* in the *SInactive* time series at time *0*

```
mean(A[SActive]@t=(0, 50)) > A[SInactive]@t=0
max(A[SActive]@t=(0, 50)) > A[SInactive]@t=0
min(A[SActive]@t=(0, 50)) > A[SInactive]@t=0
```

# CHAPTER 2

# Runner

**class** qualitative_model_fitting.**Runner**(*ant_str*, *obs_str*)

The manual interface into model valiation

This interface is intended for iteratively checking whether your model reproduces your observations. The `manual_interface` is ideal for iteratively modifying a model and checking whether the required observations are met by your model.

This contrasts with the `automatic_interface` which will modify parameters automatically until it finds a set that complies with all observations.

Usage:

First get the antimony string for the model you want to test.

```
antimony_string = '''
model SimpleFeedback()
    compartment Cell = 1;
    var A in Cell;
    var B in Cell;
    var C in Cell;
    const S;
    const I;

    A = 0;
    B = 0;
    C = 0;
    S = 0;
    I = 0;
    BI = 0;

    k1 = 0.1;
    k2 = 0.1;
    k3 = 0.1;
    k4 = 0.1;
    k5 = 10;
```

(continues on next page)

(continued from previous page)

```
    k6 = 0.1;
    k7 = 0.1;
    k8 = 0.1;

    R1: => A             ; Cell * k1*S;
    R2: A =>             ; Cell * k2*A*C;
    R3: => B             ; Cell * k3*A;
    R4: B =>             ; Cell * k4*B;
    R5: B + I => BI      ; Cell * k5*B*I;
    R6: BI => B + I      ; Cell * k6*BI;
    R7: => C             ; Cell * k7*B;
    R8: C =>             ; Cell * k8*C;
end
'''
```

And then create an input string that defines your simulations and comparisons. There are described in more detail below.

```
input_string='''
timeseries None { S=0, I=0 } 0, 100, 101
timeseries S { S=1, I=0 } 0, 100, 101
timeseries I { S=0, I=1 } 0, 100, 101
timeseries SI { S=1, I=1 } 0, 100, 101
observation
    Obs_basics1:    A[None]@t=0            > A[None]@t=10
    Obs_basics2:    A[S]@t=10              > A[S]@t=0
    Obs_basics3:    A[S]@t=25              > A[SI]@t=25
    Obs_mean:       mean(B[S]@t=(0, 100))  > mean(B[SI]@t=(0, 100))
    Obs_max:        max(B[SI]@t=(0, 100))  > max(B[S]@t=(0, 100))
    Obs_min:        min(B[SI]@t=(0, 100))  == 0
    Obs_any:        any(B[SI]@t=(0, 100)   > 3)
    Obs_all:        all(B[S]@t=(0, 100)    < 1)'''
```

Now we have a model and an input string we can use `Runner.run` to automatically check the validity of the statements in the input string.

```
>>> Runner(antimony_string, input_string).run()
          name              observation  evaluation
0  Obs_basics1                    0 > 0       False
1  Obs_basics2               0.9779 > 0        True
2  Obs_basics3          1.5713 > 2.4536       False
3     Obs_mean          0.9376 > 0.1644        True
4      Obs_max          0.3675 > 1.3467       False
5      Obs_min                   0 == 0       False
6      Obs_any  any(TimeInterval > 3)       False
7      Obs_all  all(TimeInterval < 1)       False
```

This is the first version of *qmf* and there are a number of planned features that are not yet supported. In no particular order, these are:

**Todo:**

- Build in full profile type analysis using a machine learning classification model. This would allow for profiles to be compared agaist (e.g.) a hyperbolic, transient or sigmoidal curve.
- Implement a cache system for performance improvements

- Implement the 'between' operator for implementing a rule that a component should be between x and y.
- Implement the 'almost' operator for floating point comparisons
- Implement the 'start' and 'end' operators for time intervals to abstract the need to always remember the end point of a simulation
- Allow for assigning variables to collections so we can list species that have the same rules
- Build in loops so we can do bulk validations
- Build the steady state block
- Build a dose response block
- Build the sensitivity block
- Build a plot block

# Index

## R