Implementing LIQSS2

Anton de Villiers*

April 18, 2016

1 Introduction

The aim of this document is to provide a step-by-step numerical walk-through of the *second order* Linearly Implicit Quantised State Systems (LIQSS). Mogoni *et al.* [1] provides an algorithm for calculating the n-th order LIQSS algorithm in general. This algorithm does not provide all the numerical calculations and intricacies required when implementing the solver on an actual ODE model.

In this document we will consider the Lotka-Volterra model. The Lotka-Volterra equations, also known as the predator-prey equations, are a pair of first-order, non-linear, differential equations frequently used to describe the dynamics of biological systems in which two species interact, one as a predator and the other as prey. The populations change through time according to the pair of equations:

$$\frac{dx_1(t)}{dt} = \alpha x_1(t) - \beta x_1(t) x_2(t)$$

$$\frac{dx_2(t)}{dt} = \delta x_1(t) x_2(t) - \gamma x_2(t)$$

In this model $x_1(t)$ denotes the number of prey and $x_2(t)$ denotes the number of predators at time t. In this model we set the parameters as follows: $\alpha = \beta = \delta = 0.5$ and $\gamma = 0.3$.

QSS Solver[1] [5] is a modeling and simulation environment for continuous and hybrid systems. This simulation tool describe models using a subset of the Modelica language called MicroModelica [2]. QSS Solver supports the entire family of QSS related solvers, including a family of Linearly Implicit QSS (LIQSS) [4] solvers which were created to deal with stiff mathematical systems.

When solving the Lotka-Volterra model for 100 seconds we obtain the numerical results displayed graphically in Figure 1.1.

The rest of this document will consider the numerical implementation of the Lotka-Volterra model using the second order LIQSS solver in the QSS Solver application. Mogoni *et al.* [1] provides an 8 step recursive algorithm to numerically solve an ODE for an n-th order LIQSS solver in general.

This document will not consider event handling within QSS Solver at all, since the Lotka-Volterra model does not contain any events.

*HealthQ Technologies, Office 9, First Floor, The Woodmill Lifestyle, Vredenburg Road, Devon Valley, Stellenbosch, 7600, South Africa

[1] QSS Solver is an application that contains a family of QSS implemented solvers for addressing ODE systems.

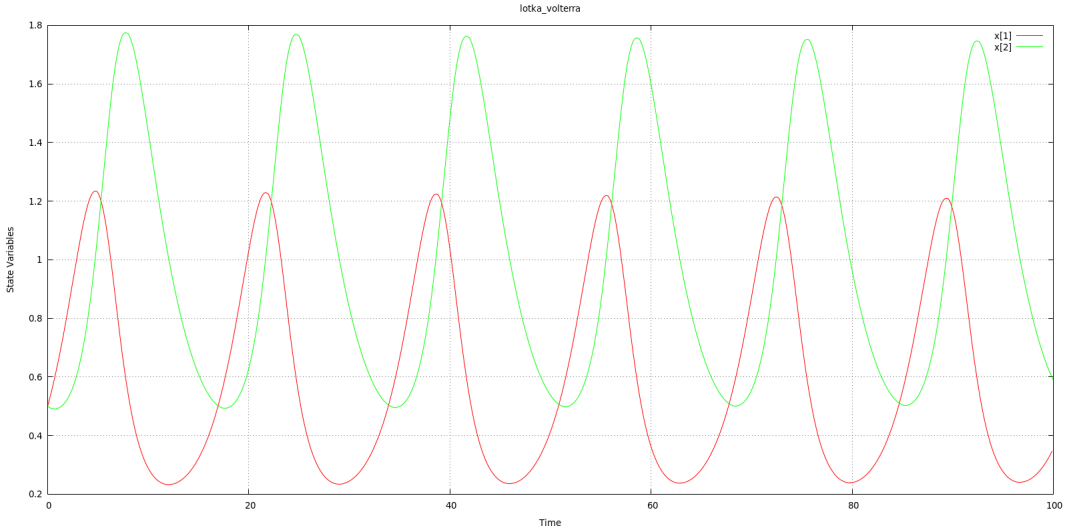

Figure 1.1: The Lotka-Volterra model simulated for 100 seconds.

2 The second order derivatives

QSS Solver makes use of GiNaC [3] to symbolically differentiate a system. Our first order derivatives look as follow:

$$\dot{x}_1(t) = 0.5x_1(t) - 0.5x_1(t)x_2(t) \tag{2.1}$$

$$\dot{x}_2(t) = 0.5x_1(t)x_2(t) - 0.3x_2(t) \tag{2.2}$$

Using the product rule, our second order derivatives are as follow:

$$\ddot{x}_1(t) = 0.5\dot{x}_1(t) - 0.5\left[\dot{x}_1(t)x_2(t) + x_1(t)\dot{x}_2(t)\right] \tag{2.3}$$

$$\ddot{x}_2(t) = 0.5\left[\dot{x}_1(t)x_2(t) + x_1(t)\dot{x}_2(t)\right] - 0.3\dot{x}_2(t) \tag{2.4}$$

3 The actual code of QSS Solver

QSS Solver has a very specific means of communicating with models that are created. Figure 3.1 displays the content of the root `qss-svoler` directory.

The most important (and relevant) subdirectories are:

- `src` — containing all the code for solving the ODE models
- `models` — contains the `.mo` files describing the model
- `build` — the converted model C code is found here with some scripts
- `output` — the outputs of the most recent simulation run with `.dat` data files.

```
anton@qwerty:~/qss-solver$ ls -lha
total 92K
drwxr-xr-x 11 anton anton 4,0K Mar  8 12:17 .
drwxr-xr-x 88 anton anton 4,0K Apr 11 11:49 ..
drwxr-xr-x  2 anton anton 4,0K Apr 11 10:31 bin
drwxr-xr-x 55 anton anton 4,0K Mar  4 09:32 build
-rw-rw-r--  1 anton anton  35K Dec 18 19:56 COPYING
drwxr-xr-x  8 anton anton 4,0K Jan 11 13:27 doc
-rw-rw-r--  1 anton anton 1,5K Dec 18 19:56 INSTALL
drwxr-xr-x 49 anton anton 4,0K Feb 11 09:21 models
drwxr-xr-x 46 anton anton 4,0K Mar  4 09:32 output
drwxr-xr-x  2 anton anton 4,0K Jan 18 12:38 packages
-rw-rw-r--  1 anton anton  247 Mar  8 12:17 .project
-rw-rw-r--  1 anton anton 2,2K Dec 18 19:56 README.txt
drwxr-xr-x  7 anton anton 4,0K Apr  8 10:35 src
drwxrwxr-x 50 anton anton 4,0K Feb  9 10:39 testsuite
drwxr-xr-x  5 anton anton 4,0K Jan 11 13:27 usr
```

Figure 3.1: The root of `qss-solver`.

To understand the workings of the solver we will investigate `src` in much detail. To start we will provide a detailed flow of how the solver iterates over the various functions and methods during each iteration (time step) when solving a model.

4 The notation related to the variables

Some of these variable alignments are specifically due for the Lotka-Volterra model.

5 Numerical interpretation of the model

Firstly, the initialization steps are discussed to get the simulation off and running. Thereafter an iterative procedure is employed to run over all required time periods to simulate the model in its entirety.

5.1 The initialization process

The model starts with $x_1(0) = x_2(0) = 0.5$. The `QSS_SEQ_initialize` function in qss_integrator.c calculates the initial values in the model. This step basically calls `LIQSS2_recomputeNextTime` in liqss2.c to find the next time that each state variable must be calculated.

5.1.1 The first state variable

Let us first consider all the values related to the first state variable (*i.e.* `var = 0`):

- `x[0] = 0.5`
- `x[1] = 0.125`
- `x[2] = 0`
- `q[0] = 0.5`
- `q[1] = 0`

Function	Location	Description
QSS_initSimulator	/engine/qss/qss-simulator.c	Initiates the simulation model.
QSS_Simulator	/engine/qss/qss-simulator.c	Allocates memory, default values, *etc.*
SIM_simulate	/engine/common/qss-simulator.c	Initializes the simulator.
QSS_simulate	/engine/qss/qss-simulator.c	Simulate the model.
↪Random	/engine/common/random.c	Not sure what happen here.
↪getTime	/engine/common/utils.c	Retrieves CPU time.
↪QSS_initializeDataStructs	/build/lotka_volterra/lotka_vol.c	Assigns values to the solver.
↪subTime	/engine/common/utils.c	Computes difference between two CPU times.
↪INT_initialize	/engine/common/integrator.c	Calls QSS_SEC_initialize.
↪↪QSS_SEC_initialize	/engine/qss/qss-integrator.c	Performs the initial values of derivatives and calculation related to initial variable settings.
↪↪FRW_recomputeDerivative	/engine/qss/qss_frw.c	Calls SYM_recomputeDerivative.
↪↪SYM_recomputeDerivative	/engine/qss/qss_frw_imp.c	Determines the derivatives.
↪↪advanceTime	/engine/common/utils.c	Determines the next time.
↪↪FRW_nextInputTime	/engine/qss/qss_integrator.c	Calls SO_nextInputTime.
↪↪SO_nextInputTime	/engine/qss/qss_frw_imp.c	Calculates next time based on §4.3 in [1]
↪↪FRW_nextEventTime	/engine/qss/qss-integrator.c	Ignored for now...
↪↪QA_recomputeNextTime	/engine/qss/qss_quantizer.c	Redirects call
↪↪LIQSS2_recomputeNextTime	/engine/qss/liqss2.c	Computes the next time step.
↪↪↪minRootPos	/engine/common/utils.c	Find smallest positive root in polynomial.
↪↪SC_Scheduler	/engine/qss/qss_scheduler.c	Not entirely sure what happens here.
↪↪getTimeValue	/engine/qss/utils.c	Retrieves time from CPU.
↪QSS_SEQC_integrator.c	/engine/qss/qss_seqc_integrator.c	Iteratively simulates all the time steps.
↪↪advanceTime	/engine/common/utils.c	Determines the next time.
↪↪QA_updateQuantizedState	/engine/qss/liqss2.c	Updates the q's.
↪↪QA_nextTime	/engine/qss/liqss2.c	Recalculates the derivatives
↪↪evaluatePoly	/engine/common/utils.c	Finds value of a polynomial with input values.
↪↪FRW_recomputeDerivatives	/engine/qss/qss_frw.c	Calls SYM_recomputeDerivative
↪↪↪SYM_recomputeDerivative	/engine/qss/qss_frw_imp.c	Calculates the ODEs
↪↪QA_recomputeNextTime	/engine/qss/liqss2.c	Computes next time

Table 3.1: A basic layout of the function calls to simulate a model (in this case the Lotka-Volterra model) in QSS Solver with the LIQSS2 solver.

Variable	Notation	Location	Description
der[1]	$\dot{x}_1(t)$	lotka_volterra.c	Calculates of $\dot{x}_1$ at t.
der[2]	$\ddot{x}_1(t)$	lotka_volterra.c	Calculates of $\ddot{x}_1$ at t.
der[4]	$\dot{x}_2(t)$	lotka_volterra.c	Calculates of $\dot{x}_2$ at t.
der[5]	$\ddot{x}_2(t)$	lotka_volterra.c	Calculates of $\ddot{x}_2$ at t.
x[0]	$x_1(t)$	qss_integrator.c	Value of x_1 at t.
x[1]	$\dot{x}_1(t)$	qss_integrator.c	Value of $\dot{x}_1$ at t.
x[2]	$\ddot{x}_1(t)$	qss_integrator.c	Value of $\ddot{x}_1$ at t.
x[3]	$x_2(t)$	qss_integrator.c	Value of x_2 at t.
x[4]	$\dot{x}_2(t)$	qss_integrator.c	Value of $\dot{x}_2$ at t.
x[5]	$\ddot{x}_2(t)$	qss_integrator.c	Value of $\ddot{x}_2$ at t.
q[0]	$q_1(t)$	qss_integrator.c	Value of q_1 at t.
q[1]	$\dot{q}_1(t)$	qss_integrator.c	Value of $\dot{q}_1$ at t.
q[2]	$\ddot{q}_1(t)$	qss_integrator.c	Value of $\ddot{q}_1$ at t.
q[3]	$q_2(t)$	qss_integrator.c	Value of q_2 at t.
q[4]	$\dot{q}_2(t)$	qss_integrator.c	Value of $\dot{q}_2$ at t.
q[5]	$\ddot{q}_2(t)$	qss_integrator.c	Value of $\ddot{q}_2$ at t.

Table 4.1: Some of the important variables in for the Lotka-Volterra model using the LIQSS2 solver.

- q[2] = 0
- a[var] = 0
- u0[var] = 0.125
- u1[var] = 0
- dq[var] = 0

The variable u0 for var = 0 is calculated as: u0[var] = x[1] - q[0] * a[var].

In a similar vein u1 for var = 0 is calculated as u1[var] = 2 * x[2] - q[1] * a[var].

Two new variables are introduced and are calculated as follows: diffxq[1] = q[1] - x[1] = -0.125 diffxq[2] = -x[2] = 0

After this another variable is used, which is initially set at follows: lqu[var] = x[0] * tolerance = 0.0005. In this case the relative tolerance of 1e-3 is used.

Now we calculate diffxq[0]. This is done in two manners. The first manner being as follows:

diffxq[0] = q[0] - dq[var] + lqu[var] - x[0] = 0.0005

After this calculation has been done, a root finding algorithm is employed called minRootPos to find the first positive root for the polynomial: ax^2+bx+c, where $a =$ diffxq[2], $b =$ diffxq[1] and $c =$ diffxq[0]. So we consider the second (actually it is a first-order) order polynomial $0x^2 - 0.125x + 0.0005$. Only one positive root exist, therefore the *minimum positive root* is MPR = 0.004. From this information, we conclude that we would want to update the time of the first state variable at 0.004 seconds in time.

Now we consider a different calculation of diffxq[0].

diffxq[0] = q[0] - dq[var] - lqu[var] - x[0] = -0.0005

Notice that this corresponds to a positive increase in the quantum, whereas the first calculation corresponds to a negative decrease in the quantum. Therefore MPR = 100, where 100 is the simulation time.

Now we do root finding on $0x^2 - 0.125x - 0.0005$. No positive roots exist in this case. Therefore we conclude that next time we would like to calculate the next time of the first state variable at 0.004 seconds from now.

5.1.2 The second state variable

Let us now consider all the values related to the second state variable (*i.e.* var = 1):

- x[3] = 0.5
- x[4] = -0.025
- x[5] = 0
- q[3] = 0.5
- q[4] = 0
- q[5] = 0
- a[var] = 0

- u0[var] = -0.025
- u1[var] = 0
- dq[var] = 0

The following calculations are performed in LIQSS2_recomputeNextTime and is located in liqss2.c.

The variable u0 for var = 1 is calculated as: u0[var] = x[4] - q[3] * a[var].

In a similar vein u1 for var = 1 is calculated as u1[var] = 2 * x[5] - q[4] * a[var].

Two new variables are introduced and are calculated as follows: diffxq[1] = q[4] - x[4] = 0.025 diffxq[2] = -x[5] = 0

After this another variable is used, which is initially set at follows: lqu[var] = x[3] * tolerance = 0.0005. In this case the relative tolerance of 1e-3 is used.

Now we calculate diffxq[0]. This is done in two manners. The first manner being as follows:

diffxq[0] = q[3] - dq[var] + lqu[var] - x[3] = 0.0005

After this calculation has been done, a root finding algorithm is employed called minRootPos to find the first positive root for the polynomial: ax^2+bx+c, where $a =$ diffxq[2], $b =$ diffxq[1] and $c =$ diffxq[0]. So we consider the second (actually it is a first-order) order polynomial $0x^2+0.025x+0.0005$. No positive root exist, therefore the MPR = 100.

Now we consider a different calculation of diffxq[0].

diffxq[0] = q[3] - dq[var] - lqu[var] - x[3] = -0.0005

Now we find that MPR = 0.02.

5.1.3 The following steps

The next time is now set at 0.004, as this the smallest positive time period for all state variables calculated. We have now concluded the initial part of the simulation.

5.2 The integration step

The following procedures are iteratively performed until the simulation is terminated. The following calculations are mainly performed in QSS_SEQC_integrate and is located in qss_seqc_integrator.c.

5.2.1 Update state variable associated with next time

The simulation begins by advancing to the next time step. The next time is dt = 0.004 for variable index = 0. The advanceTime uses this information to update the x[i] values as follows:

x[0] = x[0] + dt*x[1] + dt*dt*x[2] = 0.5005

x[1] = x[1] + 2*dt*x[2] = 0.125

Then we have to remember that dQRel is the relative tolerance of 1e-3 = 0.001.

Then lqu[index] = dQRel[index] * fabs (x[0]) = 0.0005005.

With this information we call LIQSS2_updateQuantizedState in liqss2.c.

We start off with the following that we have from the initialization process, where `var = 0`, `u0[var] = 0.125` and `u1[var] = a[var] = dq[var] = 0`.

Now for some calculations. Firstly, `elapsed = 0.004 - 0 = 0.004`.

A new variable in introduced: `qAux[var] = q[0] + elapsed * q[1] = 0.5005 - 0.004*0.125 = 0.5`.

Then we consider the old gradient: `oldDx[var] = x[1] = 0.125`.

Finally, `u0[var] = u0[var] + elapsed * u1[var] = 0.125`

Not much has changed. But now some "magic" calculations are performed some calculations. The importance or relevance of these calculations are not currently known. The following two variables are calculated as well as part of this "magic" calculations:

`dx = 0`

The variable `dq[var]` is calculated as follows (and will be done in this manner in most cases):

if `x[2] < 0` then `dq[var] = -lqu[var]`

else `dq[var] = lqu[var]`

Therefore, in this case, `dq[var] = 0.0005005 = lqu[var]`

At some point `q[0] = x[0] = 0.5005`.

Thereafter, some more calculations are performed.

`q[0] = q[0] + dq[var] = 0.5010005`

`q[1] = x[1] = 0.125`

The actual logic behind these alterations must still be clarified. The code is found at the end of `LIQSS2_updateQuantizedState` in liqss2.c. This may be a crucial set of calculations.

Alright, so we have updated the `q` values and the `x` values of some the first state variable.

5.2.2 Perform `evaluatePoly` for the other state variables

In this case we know that if the one state variable is updated, then the other state variable has to be updated as well, due to interconnectedness of the ODE system. The first state variable has already been updated. Now we have to consider the second state variable.

Firstly, we check how much time has passed for a state variable since its last update. For the second state variable that is `elapsed = 0.004`.

Now we perform `evaluatePoly`, which does the following:

`x[3] = x[3] + elapsed*x[4] + elapsed*elapsed*x[5] = 0.5 + 0.004*(-0.025) + 0 =0.4999`.

Alright, now we know that $x_2(0.004) \approx 0.4999$.

5.3 Updating the first and second derivatives

Now we call `FRW_recomputeDerivatives` in qss_seqc_integrator.c. This effectively calls `SYM_recomputeDerivatives` in qss_frw_imp.c. There are some calculations that happen here but the most important one is a pointer function to the first and second order derivatives. This is done by the call `simModel->deps()`. This calls the `MOD_dependencies` function in lotka_volterra.c.

We make use of our q's in this case to calculate the x's. Before we go into the calculations, let us just consider the current values of our q's.

q[0]	q[1]	q[2]	q[3]	q[4]	q[5]
0.5010005	0.125	0	0.5	0	0

Calculate x[1] as you would in (2.1) by making use of the q's. The following calculations only makes use of the values above in the table.

```
x[1] = der[1] = 0.5*(0.5010005) - 0.5*(0.5010005)*(0.5)
     = 0.25050025 - 0.125250125 = 0.125250125
```

Calculate x[2] as you would in (2.2) by making use of the q's in the table. We multiply (2.2) by 0.5 as well. This is an implementation consideration for all second derivative calculations.

```
x[2] = der[2] = 0.5*(0.5*(0.125) - 0.5*(0.125*0.5)) = 0.5*(0.0625 - 0.03125) = 0.015625
```

The same procedure is done for the second state variable using (2.3) and (2.4).

```
x[4] = der[4] = 0.5*(0.5010005)*(0.5) - 0.3*(0.5) = 0.12525025 - 0.15 = -0.024749875
```

```
x[5] = der[5] = 0.5*(0.5*(0.5*0.125)) = 0.015625
```

Now we have the following:

x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0.5005	0.125250125	0.015625	0.4999	−0.024749875	0.015625

5.4 Computing the next time

Here we again call LIQSS2_recomputeNextTime in liqss2.c.

5.4.1 The first state variable

Let us first consider all the values related to the first state variable (*i.e.* var = 0):

- x[0] = 0.5005
- x[1] = 0.125250125
- x[2] = 0.015625
- q[0] = 0.5010005
- q[1] = 0.125
- q[2] = 0
- a[var] = 0
- u0[var] = 0.125250125
- u1[var] = 0.03125

- `dq[var] = 0.0005005`
- `lquOld[var] = 0.5`

The variable `u0` for `var = 0` is calculated as: `u0[var] = x[1] - q[0] * a[var]`.

In a similar vein `u1` for `var = 0` is calculated as `u1[var] = 2 * x[2] - q[1] * a[var]`.

Two new variables are introduced and are calculated as follows: `diffxq[1] = q[1] - x[1] = -0.000250125` `diffxq[2] = -x[2] = -0.015625`

After this another variable is used, which is initially set at follows: `lqu[var] = lquOld[var] * tolerance = 0.0005005`. In this case the relative tolerance of `1e-3` is used.

Now we calculate `diffxq[0]`. This is done in two manners. The first manner being as follows:

`diffxq[0] = q[0] - dq[var] + lqu[var] - x[0] = 0.0005005`

After this calculation has been done, a root finding algorithm is employed called `minRootPos` to find the first positive root for the polynomial: ax^2+bx+c, where $a =$ `diffxq`[2], $b =$ `diffxq`[1] and $c =$ `diffxq`[0]. So we consider the second (actually it is a first-order) order polynomial $0.0005005x^2 - 0.000250125x - 0.015625$. The minimum positive root exist, therefore the *minimum positive root* is `MPR = 0.171149`. From this information, we conclude that we would want to update the time of the first state variable at $0.004 + 0.171149 = 0.175149$ seconds in time.

Now we consider a different calculation of `diffxq[0]`.

`diffxq[0] = q[0] - dq[var] - lqu[var] - x[0] = -0.0005005`

Notice that this corresponds to a positive increase in the quantum, whereas the first calculation corresponds to a negative decrease in the quantum. Therefore `MPR = 100`, where 100 is the simulation time. No positive roots exist in this case. Therefore we conclude that next time we would like to calculate the next time of the first state variable at 0.175149 seconds from now.

5.4.2 The second state variable

Let us now consider all the values related to the second state variable (*i.e.* `var = 1`):

- `x[3] = 0.4999`
- `x[4] = -0.024749875`
- `x[5] = 0.015625`
- `q[3] = 0.5`
- `q[4] = 0`
- `q[5] = 0`
- `a[var] = 0`
- `u0[var] = -0.025`
- `u1[var] = 0`
- `dq[var] = 0`
- `lquOld[var] = 0.5`

The following calculations are performed in `LIQSS2_recomputeNextTime` and is located in liqss2.c.

The variable `u0` for `var = 1` is calculated as: `u0[var] = x[4] - q[3] * a[var]`.

In a similar vein `u1` for `var = 1` is calculated as `u1[var] = 2 * x[5] - q[4] * a[var]`.

Two new variables are introduced and are calculated as follows:

`diffxq[1] = q[4] - x[4] = 0.024749885`

`diffxq[2] = -x[5] = -0.015625`

After this another variable is used, which is initially set at follows: `lqu[var] = lquOld[var] * tolerance = 0.0005`. In this case the relative tolerance of `1e-3` is used.

Now we calculate `diffxq[0]`. This is done in two manners. The first manner being as follows:

`diffxq[0] = q[3] - dq[var] + lqu[var] - x[3] = 0.0006`

After this calculation has been done, a root finding algorithm is employed called `minRootPos` to find the first positive root for the polynomial: ax^2+bx+c, where $a =$ `diffxq`$[2]$, $b =$ `diffxq`$[1]$ and $c =$ `diffxq`$[0]$. So we consider the second (actually it is a first-order) order polynomial $-0.015625x^2 + 0.024749885x + 0.0006$. A positive root exists, therefore the `MPR = 1.607874`.

Now we consider a different calculation of `diffxq[0]`.

`diffxq[0] = q[3] - dq[var] - lqu[var] - x[3] = -0.0004`

In this case `MPR = 0.01633`.

From this information, we conclude that we would want to update the time of the first state variable at $0.004 + 0.01633 = 0.02033$ seconds in time.

Now we have the next time as 0.02033 for updating the second state variable. We repeat the steps in this section iteratively until the model terminates.

5.5 The times recorded in the data output files

So, only the variable with the smallest `MPR` value is added to the data points that are plotted (the actual result output of the model):

For `x[0]`:

Time	0	0.004	0.1635979317188350	0.3214248996021490
`q[0]`	0.5	0.5010005	0.5214711917063192	0.5427062325916104

For `x[3]`:

Time	0	0.0203300512720699	0.1952141687993008	0.3665194624381802
`q[0]`	0.5	0.4999995	0.4952646253973667	0.4926047569404617

This is for the first 0.4 seconds of simulation time.

6 Pseudo code of LIQSS2

TODO

References

[1] Migoni G, Bortolotto M, Kofman E & Cellier FE, 2013, *Linearly implicit quantized-based integration methods for stiff ordinary differential equations*, Simulation Modelling Practice and Theory, **35**, pp. 118–136.

[2] Fernández J & Kofman E, *μ-Modelica Language Specification*, [Online], Cited 5th April 2016, Available from `http://www.fceia.unr.edu.ar/control/modelica/micromodelicaspec.pdf`

[3] GiNaC, 2016, *GiNaC is not a CAS*, [Online], Cited 15th March 2016, Available from `http://www.ginac.de/`

[4] Migoni G & Fofman E, 2009, *Linearly implicit discrete event methods for stiff ODE's*, Latin American Applied Research, **39(3)**, pp. 245–254.

[5] QSS Solver, 2016, *Modeling and simulation tool for continuous and hybrid systems*, [Online], Cited 15th March 2016, Available from `https://sourceforge.net/projects/qssengine/`