Systems Biology Final Project

Mathias D. Kallick

December 12, 2017

1 Kinetics and Motifs

For this project, we are looking at the feedback loop model from Becker-Weimann's 2004 paper "Modeling Feedback Loops of the Mammalian Circadian Oscillator". This model is essentially designed to create oscillations from a single large positive feedback loop - there is also a smaller negative feedback loop inside of the larger one ($y3$ negatively affects $y1$, which positively affects $y2$, which positively affects $y3$, thus completing the loop). The primary type of kinetics used in this model is simply first-order mass action. Most of the equations for the variable are various forms of activation by other variables, combined with various forms of degradation of the variable itself. The two notable exceptions are y_1 and y_4, the concentration of *Per2* or *Cry* mRNA, and the concentration of *Bmal1* mRNA respectively. For y_1, we see what can be described as a AND NOT gate. In computational biology, we often describe certain mathematical constructs as gates, making an analogy to binary logic (although these gates are very much non-binary). An OR gate represents a situation where a variable might be activated by either one of two other variables - if either X OR Y are present, then the variable will be activated - the truth table for this can be see in Table 1. OR gates are often thought of as sum gates - you add the two variables together. The second common gate used is the AND gate - this represents the situation where we need both X AND Y to be present for the activation to occur. AND gates are often thought of as product gates - you multiply the two variables together. The truth table for AND gates can be seen in Table 2. In this case, the authors are using a gate very similar to an AND gate, but instead of having it necessary for both variables to be there, the first variable needs to be there and the second variable needs to not be there - this is what I describe as an AND NOT gate - the truth table for this can be seen in Table 3. The equation they use for this is a Hill function with a Hill coefficient of 1 - the equation is shown in Equation 1. The other interesting equation here is the one used for y_4 - This is a Hill function with a Hill coefficient r, which they use a relatively high value of 3 for. This can be seen in Equation 2.

$$f(trans_{Per2/Cry}) = \frac{v_{1b} \cdot (y_7 + c)}{k_{1b} \cdot (1 + (\frac{y3}{k_{1i}})^p) + (y7 + c)} \tag{1}$$

X		Y		result
T	+	F	=	T
F	+	T	=	T
F	+	F	=	F
T	+	T	=	T

Table 1: Truth table for an OR gate.

X		Y		result
T	$\cdot$	F	=	F
F	$\cdot$	T	=	F
F	$\cdot$	F	=	F
T	$\cdot$	T	=	T

Table 2: Truth table for an AND gate.

$$f(trans_{Bmal1}) = \frac{v_{4b} \cdot y3^r}{k_{4b}^r + y3^r} \tag{2}$$

2 Numerical Solvers

2.1 Simulating the Model

In order to simulate this model, I coded up the ODE in both MatLab and Python - my main codebase is in Python, so it was important to have that code. However, I wanted to analyze the efficiency of various solvers for this model, and MatLab has a higher variety of available solvers than Python, so I also needed MatLab code for it. I was able to recreate Figure 3A from the paper (using Python), as can be seen in Figure 1.

2.2 Timing Various Solvers

For this project, we were tasked with looking at run-times of various computational ODE solvers - two stiff solvers and two non-stiff solvers (ode23, ode45, ode23s, ode15s). Stiffness is a property of ODEs that makes non-stiff solvers take much longer to solve them - essentially, they gain error very quickly, so non-stiff solvers need to take minuscule time-steps in order to get the error down to acceptable levels. Stiff solvers take a longer per time-step (they do more work so that error is kept down), but they can take longer time-steps because of that. A stiff equation will not take longer with a stiff solver, but it will take much longer with a non-stiff solver. We found, for a simulation over 10000 hours with $dt = .01$, ode23 took 12.05 seconds, ode45 took 10.55 seconds, ode23s took 12.88 seconds, and ode15s took 26.43 seconds. For ode23 and ode45,.this is a bit strange, since ode23 is a lower order solver than ode45 - higher order solvers

X		Y		result
T	$\cdot$	F	=	T
F	$\cdot$	T	=	F
F	$\cdot$	F	=	F
T	$\cdot$	T	=	F

Table 3: Truth table for an AND gate.

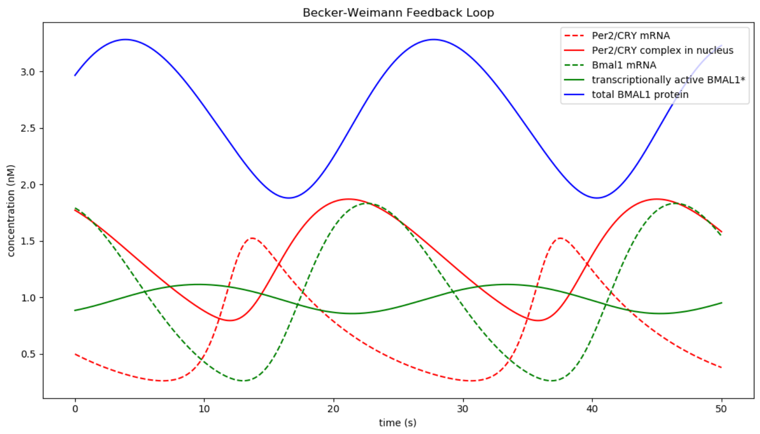

Figure 1: Recreation of Figure 3A from Becker-Weimann Paper. Recreated using ode15s in Python.

tend to take longer. However, the difference between 12.05 and 10.55 seconds is small enough that I could see this as operating system strangeness. Equally, the fact that ode23 and ode23s took essentially the same amount of time is a little strange - I did do a more in depth examination based on this (see Table 4) and found that ode23 and ode23s only take approximately the same amount of time for smaller time-steps. For $dt = .1$, ode23 takes much less time. Equally, ode15s only takes longer than ode23s when $dt = .01$, (this is also true for ode45 vs ode23) which suggests that higher order solvers can be faster for bigger time-steps, but take much longer with smaller time-steps. This might be due to the error restrictions on the system (Relative Tolerance $= 1e-6$, Absolute Tolerance $= 1e-8$). Since the stiff solvers take more time than the non-stiff solvers consistently (not always by much though), we can conclude that this system is not stiff.

2.2.1 Python vs. MatLab

Given that I needed to use MatLab for this part of the project, and Python for the rest of it, I decided to do a little speed-test between the two systems.

MatLab Timing	ode23	ode45	ode23s	ode15s
hours= 1000, timestep= .1	0.5657s	0.3491s	1.0390s	0.8864s
hours= 10000, timestep= .1	2.7256s	2.1562s	8.7303s	6.5089s
hours= 1000, timestep= .01	1.2880s	1.3271s	1.5278s	3.2722s
hours= 10000, timestep= .01	12.0546s	10.5547s	12.8800s	26.4383s

Table 4

I initially just wanted to see how efficient my code way (I assumed that the backends would be about the same). I ran my Python code under the same conditions as the MatLab code on the same computer right my MatLab test (see Table 5). As you can see, these times are all substantially faster than MatLab was able to run the same model with ode15s. Not only does this make me pretty proud of my frontend code, but it also speaks to the efficiency of SciPy. I did some research and I think this mostly comes from the fact that MatLab is coded in C, while SciPy has all of its integration code written in FORTRAN.

Python Timing	ode15s
hours= 1000, timestep= .1	0.1235s
hours= 10000, timestep= .1	1.1968s
hours= 1000, timestep= .01	0.6846s
hours= 10000, timestep= .01	6.4685s

Table 5

3 Sensitivity Analysis