The Confounding Role of GFP in the Repressilator Circuit

Mark Stevens
Mark.Stevens@mail.utoronto.ca
1009015192

JCP410 - Modelling of Biochemical Systems

Abstract

The Repressilator, a synthetic genetic oscillator, has been shown to have significant noise in-vivo. Previous researchers have found that replacing the original reporter protein GFP with the reporter mVenus causes the system to become more stable. In this experiment, we investigate possible mechanisms by which GFP may cause instability in our system. We used the possible explanation that GFP induced oxidative stress through hydrogen peroxide production to inform the model parameters and chemical pathways. We used a Stochastic Gillespie Simulation to first replicate the established repressilator model. We then extended our model by introducing GFP. We characterized the stability of the system when GFP interacts with each mRNA molecule individually, and all mRNA molecules combined. This experiment revealed GFP-induced instability when GFP caused increased expression of TetR and LacI. We also found GFP-induced instability when GFP created global mRNA degradation. However, we found that GFP-induced repression of TetR, LacI, and Lambda CI were not strongly associated with GFP-induced instability.

Introduction

The Represssilator is the first synthetic genetic oscillator. The system utilizes a series of negative feedback loops to create temporal oscillations in the concentration of each component. Figure 3 shows the protein pathway of the Repressilor. The Repressilator utilizes 3 proteins: the LacI protein from E. coli, tetR from the tetracycline-resistance transposon Tn10, and cI from λ phage (Elowitz, 2000). For clarity, the three proteins will be referred to as P_1, P_2, and P_3. We will also utilize some variable names throughout the paper: m_n is the RNA transcript, P_n is the protein, and ∅ represents the environment. To understand the mechanics of the system, it will be helpful to consider a single cycle. When P_1 is expressed, the protein binds to P_2's promoter sequence, causing the concentration of P_2 to decrease. P_3 is now free to express. Once P_3 begins expression, it triggers the degradation of P_1. And thus, the cycle continues at P_3.

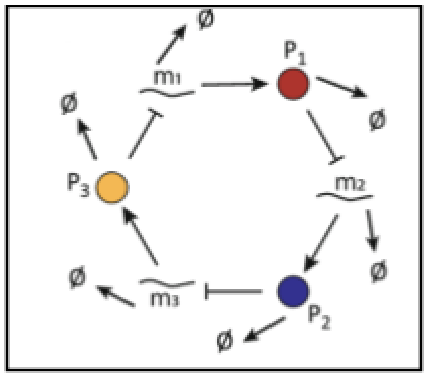


Figure 3. Repressilator Schematic (McCallum, 2021)

Figure 1 illustrates the DNA and promoter sequences of the original repressilator circuit. The original repressilator uses the TetR protein to inhibit expression of green fluorescent protein (GFP) adeno-associated virus (AAV). However, newer research suggests that gfp-aav interferes with the degradation of some repressors in the model, causing unexpected behavior and significant noise. To address these concerns, researchers use a new fluorescent protein, mVenus, and place the sequence on the same plasmid as the other proteins and promoters. Both the original gene network (Figure 1) and the new gene network (Figure 2) have been included below.

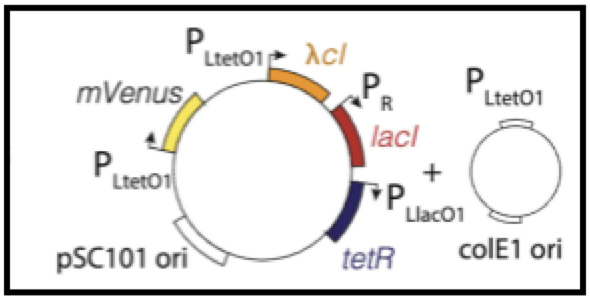


Figure 2. Updated Repressilator Gene Network (Potvin-Trottier 2016, McCallum 2021)

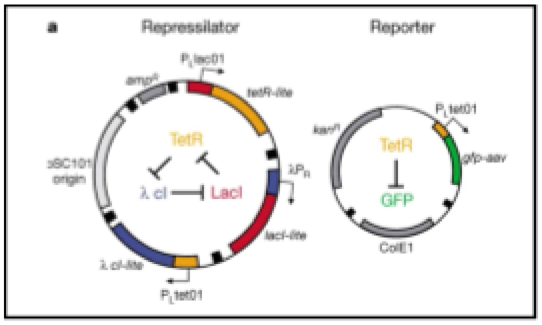


Figure 1. Original Repressilator Gene Network (Elowitz, 2000)

The exact mechanism for how GFP may cause influence protein degradation is not well understood. However, GFP is known to produce hydrogen peroxide during its synthesis while mVenus does not (Ansari 2016, Ganini 2017). Hydrogen peroxide is known to cause oxidative stress in E. Coli. This paper will investigate the potential dynamics of GFP, hydrogen peroxide, and the effects of oxidative stress.

Previous work has shown TetR is recruited in response to oxidative stress, as the gene codes for Superoxide Dismutases (SODs) which reduce superoxide ions like hydrogen peroxide (Liu, 2014). It has also been shown that Lambda CI increases in response to oxidative stress (Glinkowska, 2010). Both of these genes are involved in our repressilator model, and this potential up-regulation will be investigated. For completeness, we will also investigate LacI up-regulation in response to oxidative stress. It has also been shown that oxidative stress causes a global decrease in transcription levels (Zhu, 2019). This global transcription decrease will be investigated.

The schematics for the pathways being investigated are included in the figures below.

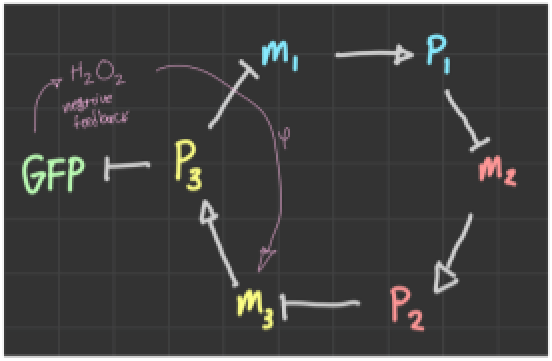


Figure 4. Schematic of Experiment 1, showing how Protein P_3 inhibits GFP, which creates H_2O_2, which uses the function φ as the rate function to affect mRNA$_3$

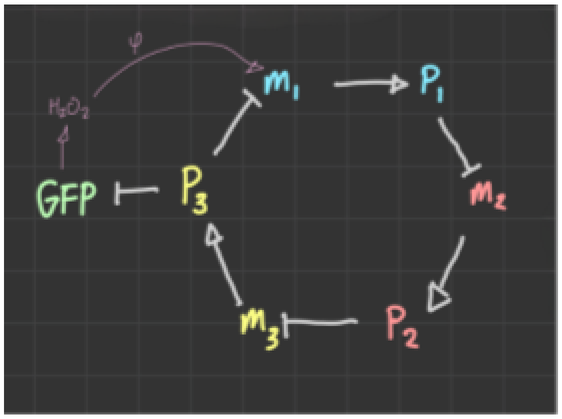


Figure 5. Schematic of Experiment 2, showing how Protein P_1 inhibits GFP, which creates H_2O_2, which uses the function φ as the rate function to affect mRNA$_1$

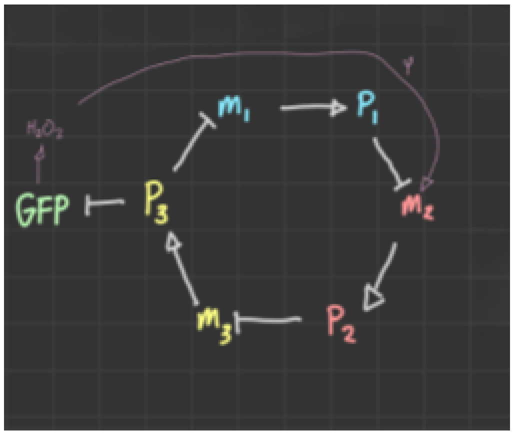


Figure 6. Schematic of Experiment 3, showing how Protein P_3 inhibits GFP, which creates H_2O_2, which uses the function φ as the rate function to modify $mRNA_2$

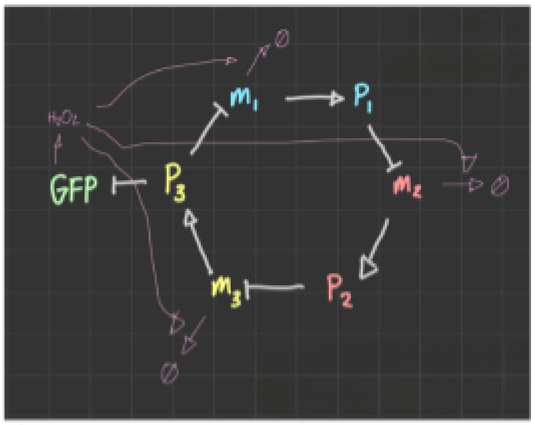


Figure 7. Schematic of Experiment 4, showing how Protein P_3 inhibits GFP, which creates H_2O_2, which increases the global degradation of mRNA

Existing Mathematical Model Description

To understand the behavior of the Repressilator circuit, we use two simulation techniques: ODE simulations and Stochastic Gillespie simulations. We use both simulations to determine the expected and ideal behavior of our system, and compare with the results we might see simply due to random variations in our system.

ODE:

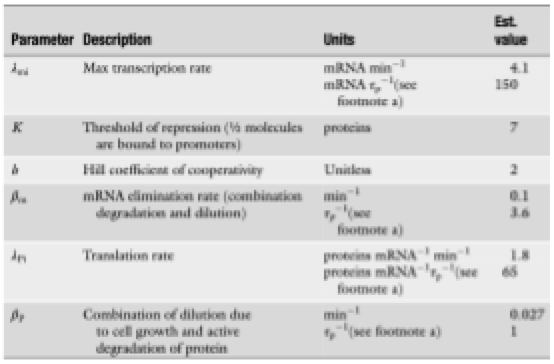
The ODE simulation relies upon two mass-action assumptions. These are *spatial homogeneity*, that the reaction rates are constant regardless of location, and *the continuum hypothesis*, that there are many molecules in the system. These assumptions allow us to utilize continuous differential equations to model our system. The equations and constants used are described below. Note that λ represents creation and β represents degradation. Also note that our simulation's parameters come from the public database BioNumbers (Milo, 2010).

$$\emptyset \xrightarrow{f(P_{i-1})} m_i$$

$$m_i \xrightarrow{\beta_m} \emptyset$$

$$m_i \xrightarrow{\lambda_P} P_i + m_i$$

$$P_i \xrightarrow{\beta_P} \emptyset$$



Parameter	Description	Units	Est. value
λ_{m1}	Max transcription rate	mRNA min^{-1} mRNA τ_p^{-1}(see footnote a)	4.1 150
K	Threshold of repression (½ molecules are bound to promoters)	proteins	7
h	Hill coefficient of cooperativity	Unitless	2
β_m	mRNA elimination rate (combination degradation and dilution)	min^{-1} τ_p^{-1}(see footnote a)	0.1 3.6
λ_{P1}	Translation rate	proteins mRNA^{-1} min^{-1} proteins mRNA^{-1}τ_p^{-1}(see footnote a)	1.8 65
β_P	Combination of dilution due to cell growth and active degradation of protein	min^{-1} τ_p^{-1}(see footnote a)	0.027 1

Table 1. Parameter values utilized in both our deterministic and stochastic simulations.

We use the set of functions above to model our system, where i=1, 2, 3 for the 3 different proteins. β_m, β_i represent the rates of degradation of the mRNA and protein respectively. λ_p is the rate of translation of the repressor. Our Hill Equation is $f(P_{i-1}) = \frac{\lambda_m \cdot K^h}{K^h + P_{i-1}^h}$. This is used to help represent "turning on or off" of our specific genes. The proteins bind cooperatively to the promoters, thus turning the gene "off". When no proteins are bound to the promoters, the gene is assumed to be "on". h is the hill coefficient and K is the threshold for half of the repressors to binding to the binding sites and represents the affinity the proteins have for the binding sites. λ_m is the translation rate of the protein.

Gillespie Stochastic Simulation

The equations we use for a stochastic simulation are markedly similar to that in our ODE simulation. However, as opposed to modeling the concentration of each species directly, we instead model the probability of each reaction occurring at each time step, and randomly sample from this probability distribution. The set of changes to our system is described to the right. Note that the numbers of proteins and mRNA transcripts are discrete, while the rates of the reactions are continuous.

$$\begin{gathered}
m_i \xrightarrow{\frac{\lambda_m \cdot K^h}{K^h + {P_{i-1}}^h}} m_i + 1 \\
m_i \xrightarrow{\beta_m \cdot m_i} m_i - 1 \\
P_i \xrightarrow{\lambda_P \cdot m_i} P_i + 1 \\
P_i \xrightarrow{\beta_P \cdot P_i} P_i - 1
\end{gathered}$$

At each time step, we must find both the type of reaction occurring and the time until this reaction occurs. We repeat this over our entire simulation. We solve for time with the equation $\tau = \frac{-\ln(r_1)}{\lambda_{tot}}$ where λ_{tot} is the sum of all rates of reactions and r is some random number between 0 and 1. This equation comes from the exponential probability distribution, which was chosen for its property of being "memoryless". That is to say, the distribution is not dependent on the amount of time which has passed so far, which is the behavior we would expect from our biological system.

We determine the type of reaction by taking a weighted sample from our rate vector function (RVP). The RVP determines the rate of each possible reaction. We then assume that reactions with a higher weight are more likely to occur. We sample from this probability distribution by sorting all rates on a number line, normaling to be between 0 and 1, and then take a random sample from this distribution. Our

$$\begin{aligned}
rvf(m,P) = \Big[&\frac{\lambda_m \cdot K^h}{K^h + P_3^h},\ \beta_m \cdot m_1, \lambda_P \cdot m_1, \beta_P \cdot P_1, \frac{\lambda_m \cdot K^h}{K^h + P_1^h},\ \beta_m \cdot m_2,\ \lambda_P \cdot m_2, \\
&\beta_P \cdot P_2, \frac{\lambda_m \cdot K^h}{K^h + P_2^h},\ \beta_m \cdot m_3,\ \lambda_P \cdot m_3,\ \beta_P \cdot P_3\Big]
\end{aligned}$$

sample will correspond to the reaction which we simulate as having occurred.

The original simulations in matlab are available at this GitHub repo, while the simulations rewritten in Python are available at this GitHub repo.

Augmenting existing model

Similar to the equations we've established for the proteins in our repressilator, we will utilize the equations above to simulate both the number of GFP mRNAs and GFP proteins. It has been shown that GFP synthesis produces 19 Hydrogen Peroxide molecules per GFP (Seaver, 2001). This stoichiometric production will be called ALPHA and will be one of our simulation parameters. Similarly, it has been shown that Hydrogen Peroxide inside an E. Coli cell undergoing exponential growth, like we see in our repressilator model, follows the equation $\frac{d[H_2O_2]}{dt} = 14 + \alpha[GFP] - 82.6 \cdot [H_2O_2] - [H_2O_2]\frac{V_{max}}{[H_2O_2]+K}$. This yields the foundation of our augmented repressilator model.

Experiments 1, 2, and 3

For each gene in the repressilator, we will simulate two equations: linear production of the mRNA and a Hill-Equation repression of the mRNA. Although linear production of mRNA by hydrogen peroxide is not realistic, this was investigated to provide a base-level understanding of how hydrogen-peroxide-induced production may influence the system. We also investigate a Hill-Equation repression, which could correspond to Hydrogen Peroxide directly binding to the promoter sequence (which is unlikely) or Hydrogen Peroxide activating some secondary protein to act as the inhibitor. We will modify the target mRNA's production rate function to be one of the following in the linear and Hill systems, respectively:

$$\frac{\lambda_m \cdot K^h}{K^h + P_{i-1}^h} + \beta \cdot [H_2O_2] \qquad \frac{\lambda_m \cdot K^h}{K^h + P_{i-1}^h} + \frac{\beta \cdot K^h}{K^h + [H2O2]^h}$$

Beta will be our second parameter we investigate in our experiments.

Experiment 4

We will also simulate Hydrogen Peroxide causing transcription to decrease globally. We will simulate this by adding the following term to each mRNA's degradation rate:

$$\beta_m \cdot m_i + \beta \cdot [H2O2]$$

It should be noted that our simulation corresponds to Hydrogen Peroxide directly causing the degradation of mRNA which is not necessarily realistic. However, this simplified model will provide a foundation from which to expand upon further.

We will perform two simulations for this experiment: one including the degradation of GFP's mRNA and one without. The system including the GFP mRNA degradation is a negative feedback loop, and it was believed that this may prevent the hydrogen peroxide from causing a substantial effect in the stability of the repressilator system. Thus, an experiment was included which did not have this self-regulation.

Results and Discussion

We first began by performing the deterministic simulation of the original repressilator. We compared this result with known deterministic simulations of the model, and found our simulation was consistent with other scientific works (McCallum, 2021). These simulations both illustrate a very consistent period of approximately 10 minutes.

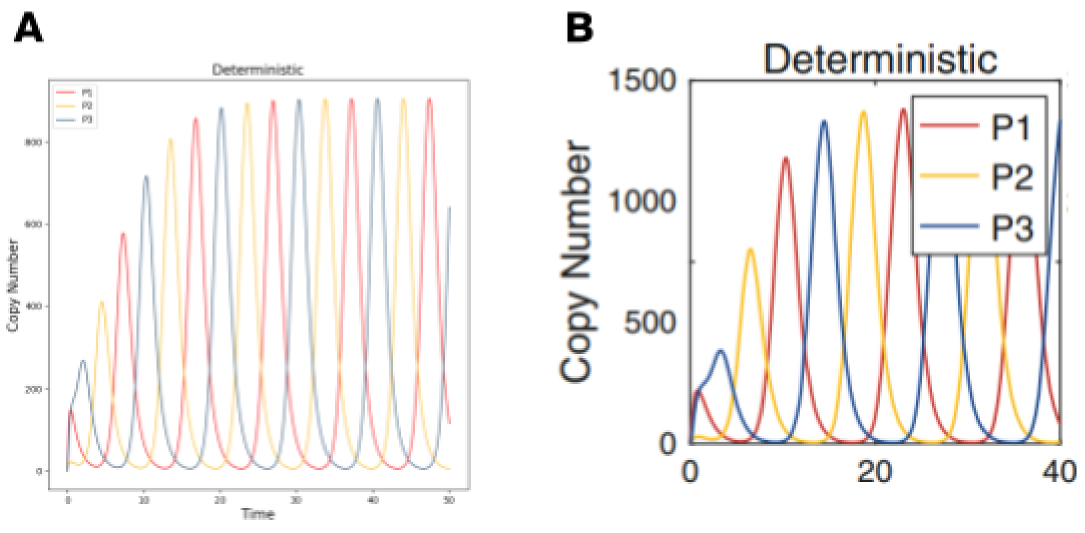


Figure 8. A time-concentration graph of the deterministic simulation. The concentrations of proteins P_1, P_2, and P_3 are plotted against the time of the simulation in minutes. The parameters for both simulations have been included in Table 1. Our experimental simulation **(A)** is compared against previously found experimental results **(B)**.

While Figure 8 demonstrates the ideal behavior of the Repressilator circuit, the deterministic model fails to capture the system's behavior under stochastic conditions. While a stability analysis of the system would help to understand the sets of parameters which yield a stable system, this analysis fails to encompass the sensitivity of the system due to random fluctuations in the numbers of molecules while the system is evolving.

To address these concerns, a stochastic simulation has also been performed and included above in Figure 9.

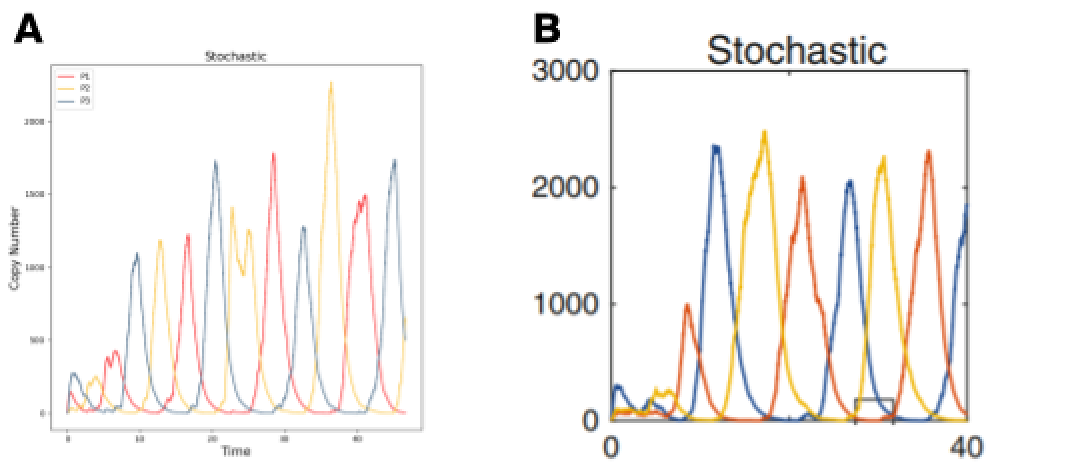


Figure 9. A time-concentration graph of the stochastic simulation. The numbers of proteins P_1, P_2, and P_3 are plotted against the time of the simulation in minutes. The parameters for both simulations have been included in Table 1. A single time-trace of our simulation **(A)** has been compared against a single time trace simulated by previous researchers **(B)**.

Visualizations of a single pass of the simulation can be helpful in developing an intuition for how the system might evolve over time. However, generalizations about the parameters and stability of the system should not be made from a single pass, as results are likely to be highly influenced by random variance. In the visualization included in Figure 9, we can see how the maximum copy number of each protein varies in each cycle. Similarly, we can see that at 25 minutes, the concentration of P_2 initially decreases, increases again, and finally decreases. This premature decrease was clearly not caused by the concentration of P_1 like we would expect, as the concentration of P_1 had not yet increased. Instead, the concentration of P_2 was highly influenced by our random sampling method. Purely by chance, the number of copies of P_2 decreased instead of increased. This is just a single example of the types of variations we might expect to see if we were to perform this experiment in vivo.

To analyze the typical stability of our system, we performed this simulation using a common technique called a “grid scan”. In this technique, we test our experiment over every possible combination of two sets of parameters. In this way, we can understand our system’s dependence on both parameters. Of interest are our systems degradation of protein β_p as a ratio of the system’s degradation of mRNA transcript β_m. We will call this parameter β. We will also study our system's overall production $\lambda_p \cdot \lambda_m$ as a ratio of our system's overall degradation $\beta_p \cdot \beta_m \cdot K$, and call this parameter α. To find these ratios, the parameter β_p was held to a constant value of 1, and values of other entries were varied.

This was performed by plotting α against β, where $\alpha = \frac{\lambda_p \cdot \lambda_m}{\beta_p \cdot \beta_m \cdot K}$ and $\beta = \frac{\beta_p}{\beta_m}$. For each time trace, we determined the autocorrelation of the system by finding the autocorrelation of each time trace over time, and using the convolution theorem to combine these time traces and find a general autocorrelation of our entire system for a single time trace. The result was our autocorrelation, which represents the signal's similarity from its first peak to its second peak. This correlation allows us to understand how "predictable" our system is, and how stable we expect the peaks of our concentrations to be over time. We expect the correlation to be 1 in a perfect oscillator.

Note that 1000 trials were performed for each of the 121 bins using the UWU cluster (a small, 4-node, 112 thread cluster I built) using massive parallelization. In total, the cluster used one node and 28 cores, and took 10 hours to compute all trials (as opposed to the estimated 70 hours on a desktop computer). For numpy to produce random samples for each trial, a random seed was input into each thread.

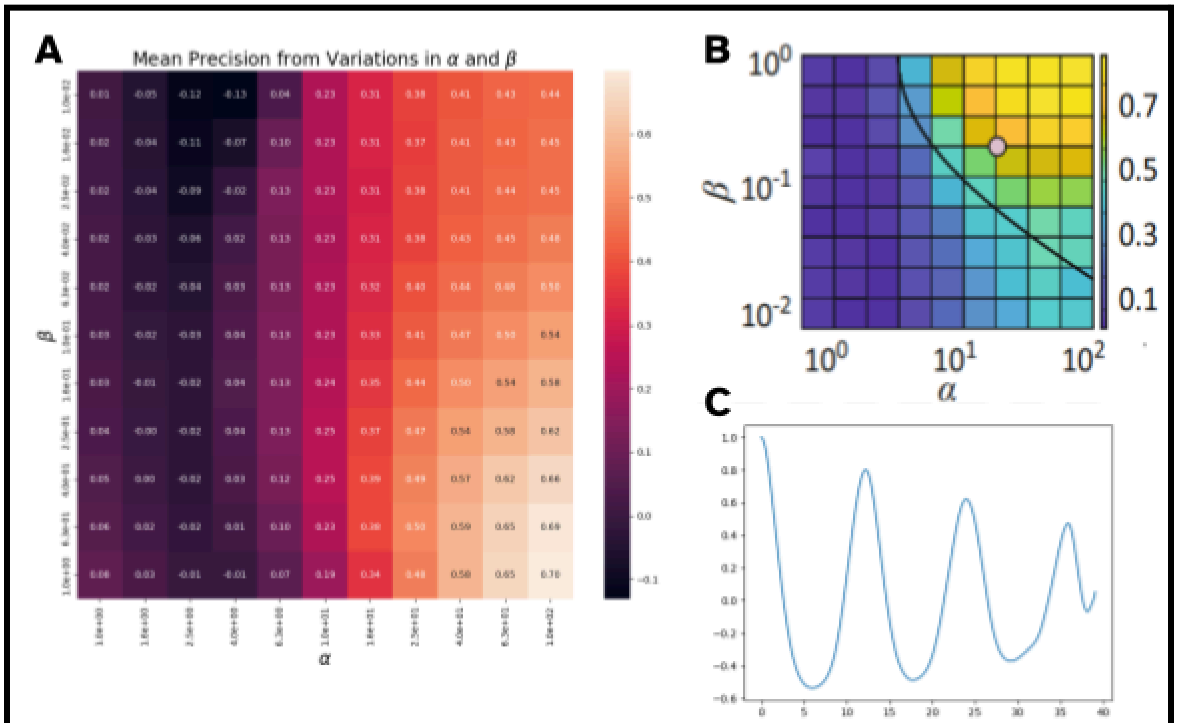


Figure 10. Average correlation of our system as a function of $\alpha = \frac{\lambda_p \cdot \lambda_m}{\beta_p \cdot \beta_m \cdot K}$ and $\beta = \frac{\beta_p}{\beta_m}$ for n=1000 trials. Our generated precision **(A)** is compared against known precisions **(B)** for the Repressilator circuit. Note that graph **(A)** has an inverted y-axis compared to graph **(B)**. The precision (also called the autocorrelation) is generated by comparing the correlation of a signal with a delayed copy of itself. This generates a precision graph **(C)**. This precision graph displays the correlation between the signal and later copies of itself. On the vertical axis the correlation, and on the horizontal axis is time. The distance

Because our produced heatmap in Figure 10 is similar to the heatmaps of similar works, we conclude that our experiments are similar, and that our model has the same predictive power as previously established models. To estimate the period of a single cycle of the repressilator, the period (the distance from peak to peak of our autocorrelation) was computed for each trial, and the average was taken.

From Figure 10, we find that our system's period is much more dependent on α than on β. However, once our system is in a stable regime ($\alpha \geq 10$), the period is also dependent on values of β. For future work, a broader parameter scan will be performed to attempt to maximize the stability of our system.

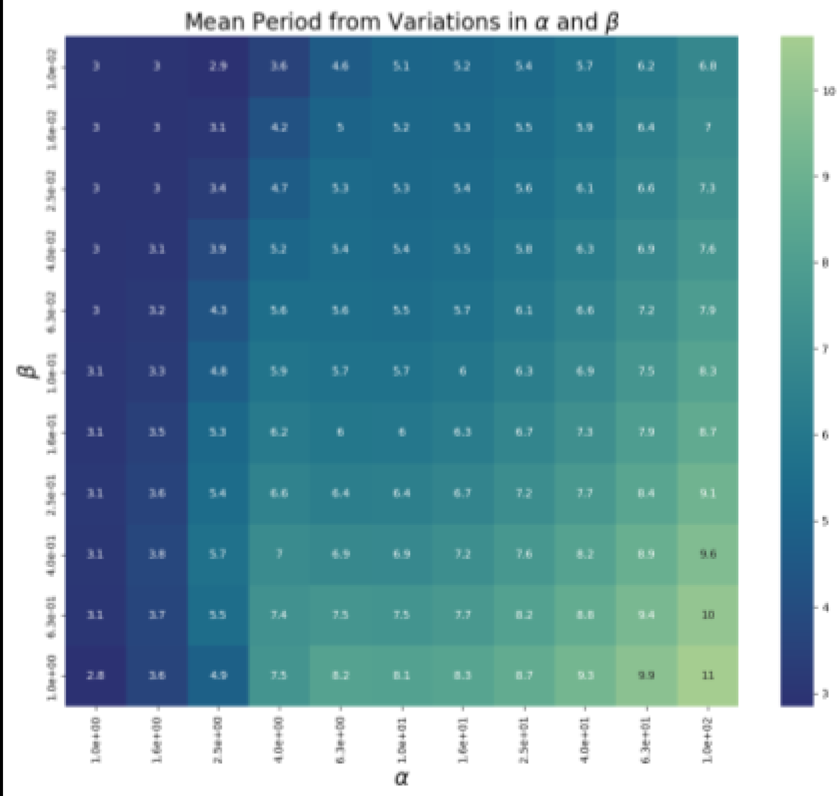


Figure 11. Period of Repressilator oscillation as a function of $\alpha = \frac{\lambda_p \cdot \lambda_m}{\beta_p \cdot \beta_m \cdot K}$ and $\beta = \frac{\beta_p}{\beta_m}$ for n=1000 trials. Note that the time is computed in minutes.

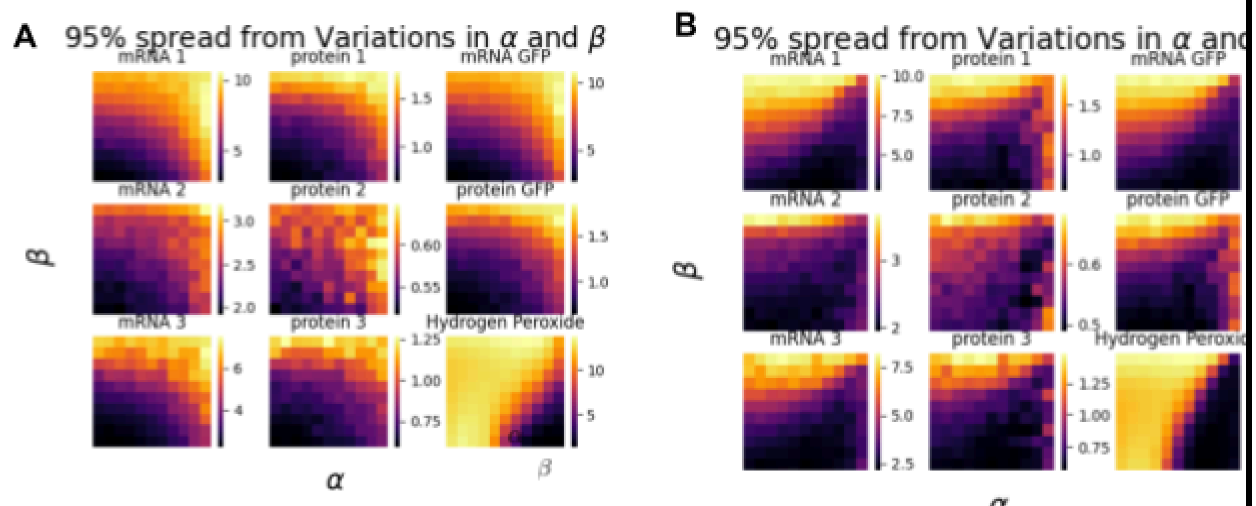


Figure 12. A) Experiment 1 Linear 95% spread in power-spectrum. **B)** Experiment 1 Hill function. α is the number of H_2O_2 produced per GFP. α begins at 0 and ends at 100 on a log scale with the units of Hz. β starts at 130 and ends at 1000. β is further described in the methods section for bot the linear and Hill functions. Brighter values correspond to a higher value, and hence a greater spread in the power-spectrum. A more detailed explanation on the interpretation of these power spectra is included in **Appendix A.**

Experiment 1 Findings:

In both our Linear and Hill simulations, we find that protein 2 exhibits the least change in response to this modified system. This makes sense, as protein 2 is the most "distant" from protein 3 in our repressilator loop. Perturbations which happen at protein 3 are slightly stabilized in protein 1, and are virtually entirely stabilized by protein 2. We see that mRNA 3 exhibits the greatest spread in oscillations, which is also consistent with our expectations. We are essentially creating a second, smaller loop involving protein 3 and GFP, which causes significant noise in our system.

While both systems experience increased instability in response to higher values of BETA, we note that the linear system's instability is exacerbated by increases to alpha while the Hill system is somewhat protected by increases to alpha. This result is not as easily explained, and further research is necessary. These results point to a possible explanation that general increases to the numbers of mRNA 3 cause instability.

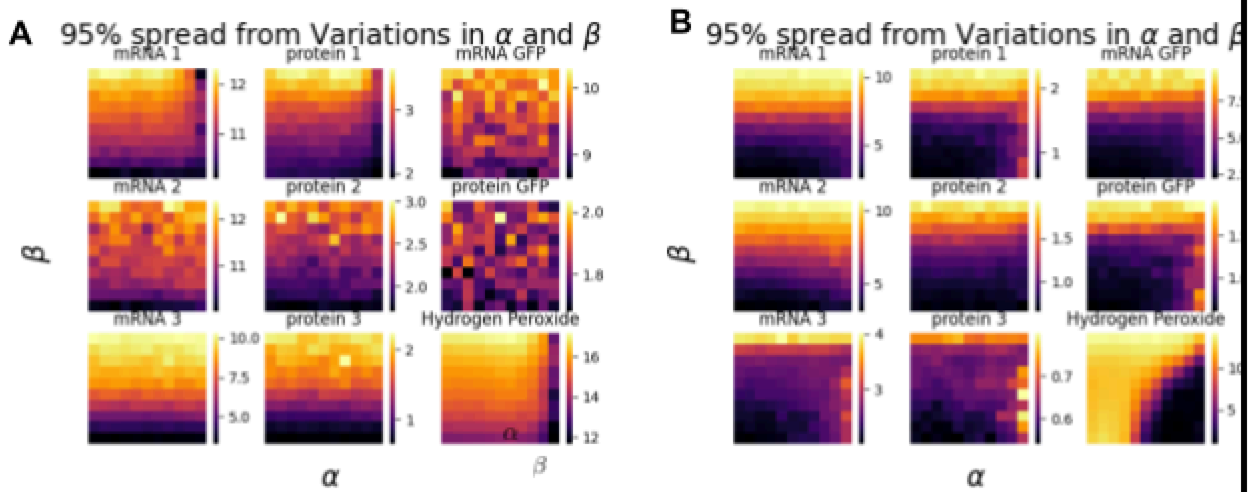


Figure 13. A) Experiment 2 Linear 95% spread in power-spectrum. **B)** Experiment 2 Hill function. α is the number of H_2O_2 produced per GFP. α begins at 0 and ends at 100 on a log scale with the units of Hz. β starts at 130 and ends at 1000. β is further described in the methods section for bot the linear and Hill functions. Brighter values correspond to a higher value, and hence a greater spread in the power-spectrum. A more detailed explanation on the interpretation of these power spectra is included in **Appendix A**.

Experiment 2 Findings:

Both our Linear and Hill simulations show very little dependence on alpha, and are mostly dependent on beta. This indicates that the system is mostly dependent on the presence of hydrogen peroxide and is largely independent of the timing of that hydrogen peroxide. It should be noted that GFP “tracks” m1. That is to say, the mRNA for both protein1 and GFP are inhibited by protein 3, and we see the concentrations of protein 1 and GFP stay relatively similar.

Further experimentation is required to fully understand this result. It should be noted that the small increases in instability which we see in our Hill simulation are likely a failure of our Gillespie algorithm and are further explained in APPENDIX X.

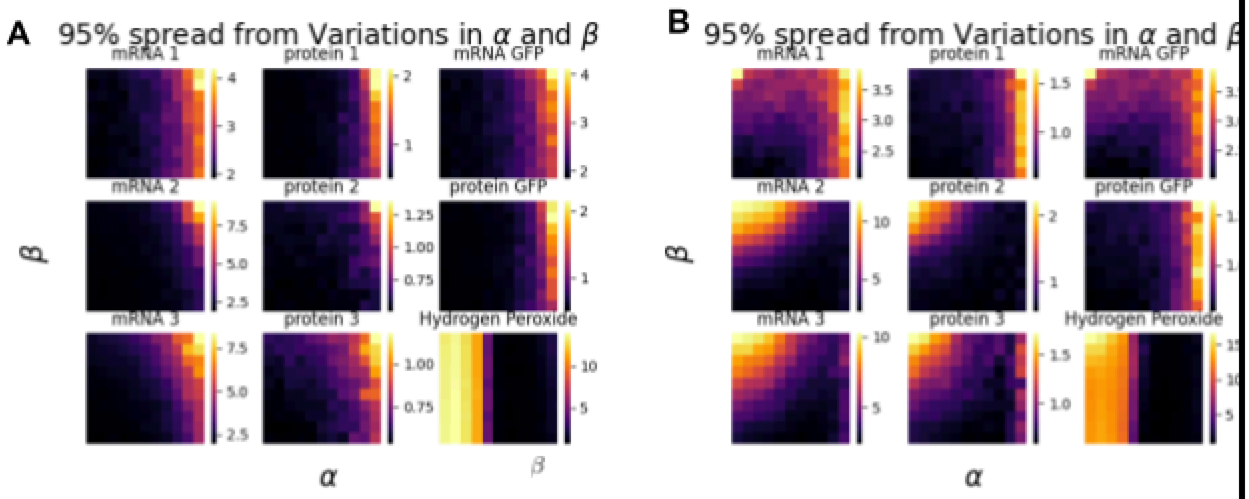


Figure 14. A) Experiment 3 Linear 95% spread in power-spectrum. **B)** Experiment 3 Hill function. α is the number of H_2O_2 produced per GFP. α begins at 0 and ends at 100 on a log scale with the units of Hz. β starts at 130 and ends at 1000. β is further described in the methods section for both the linear and Hill functions. Brighter values correspond to a higher value, and hence a greater spread in the power-spectrum. A more detailed explanation on the interpretation of these power spectra is included in **Appendix A**.

Experiment 3 Findings:

The Linear and Hill simulations for this experiment show very drastic differences. The Linear system requires a threshold level of alpha before any instability is shown. After this threshold is met, the system seems to be somewhat dependent on beta. In contrast, the Hill system shows a threshold value for beta in order to show instability. Higher levels of alpha seem to be a protecting factor in the Hill system, and cause the instability to decrease.

Recall that GFP "tracks" protein 1, and both molecules exhibit increased concentration at roughly the same time. This is because the mRNA for both protein 1 and GFP is inhibited by protein 3. We can see that hydrogen peroxide's mRNA stimulation acts against the inhibition protein 1 performs. Thus, for the linear system, hydrogen peroxide is required to "overcome" the effects produced by P1 before it is able to have any effect on m2. This is why the linear system has such a large dependence on alpha, as it requires a large number of molecules to have a substantial effect. In the Hill system, the hydrogen peroxide shares the same inhibition activity as protein 1. Consequently, it triggers instability when it is not dependent on GFP expression (low alpha) and when it constantly stimulates the mRNA (high beta).

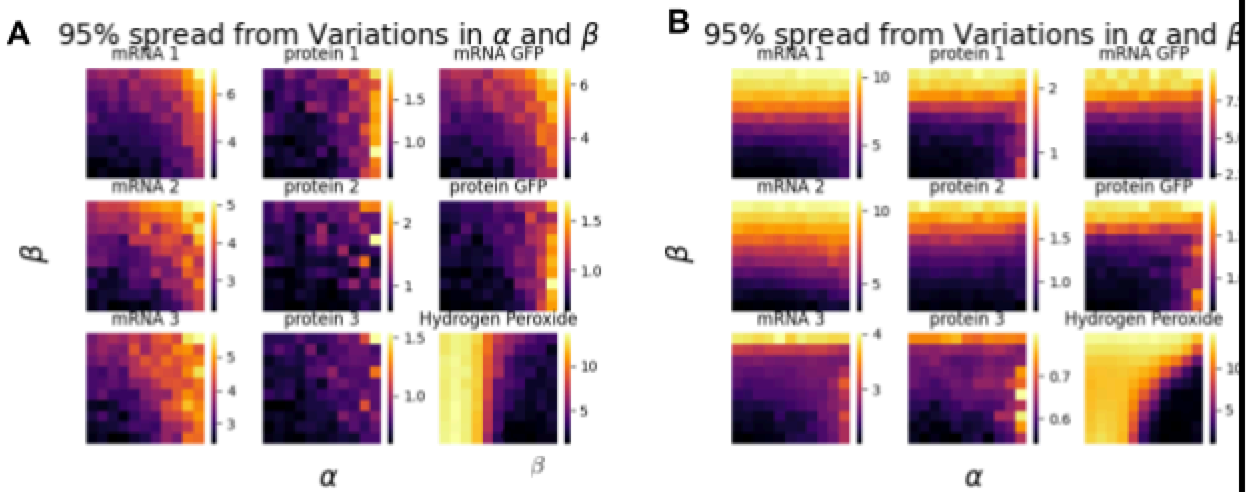


Figure 15. A) Experiment 4 Linear 95% spread in power-spectrum. **B)** Experiment 4 Hill function.α is the number of H_2O_2 produced per GFP. α begins at 0 and ends at 100 on a log scale with the units of Hz. β starts at 130 and ends at 1000. β is further described in the methods section for this mRNA degradation. Brighter values correspond to a higher value, and hence a greater spread in the power-spectrum. A more detailed explanation on the interpretation of these power spectra is included in **Appendix A**.

Experiment 4 Findings

In both experiments performed, the system exhibits increased instability with increases to alpha and beta. This indicates that our system is sensitive to increases in both the number of hydrogen peroxide produced per GFP (alpha) and to the global increase of mRNA degradation (beta).

This is consistent with our initial experimental findings, that a decrease to the protein degradation to mRNA degradation ratio causes instability in the system. By increasing our mRNA degradation, we throw our system back into an unstable regime, and is a possible explanation for how GFP may cause instability in the repressilator system.

Conclusion

While our experiment doesn't isolate oxidative stress as the only means by which GFP may cause instability in the repressilator system, it does help to implicate possible means by which GFP may induce instability in the repressilator. Systems which exhibit increased noise in response to increases in alpha (hydrogen peroxide produced per GFP) are known to be pathways in which GFP may increase noise in the repressilator system.

We found the Linear experiment 1, Linear experiment 3, and both versions of experiment 2 exhibited GFP-induced instability. All hill experiments and the linear experiment 2 failed to show GFP-induced instability.

Although we specifically studied oxidative stress, the results from this experiment can be generalized to other GFP-induced instability mechanisms. We are able to conclude that the most likely mechanisms by which GFP could induce instability are by increasing expression of protein 3 or protein 2, or by triggering global mRNA degradation.

This experiment also uncovered the fact that when simulating systems with varying magnitudes of protein copy numbers, it's critical to use a constant-window time. That is to say, when running a Gillespie simulation, it's critical to keep in mind that the average ΔT is dependent on the overall rates of the systems. When reactants are introduced with high production and degradation rates (like the hydrogen peroxide), the average ΔT decreases, as it reactions happen more quickly. This doesn't affect the distribution of our other reactants, but it does mean that if we continue to use a fixed number of iterations, we will end up simulating a smaller total time. Instead of using a fixed number of iterations, we should use a fixed length of time to simulate. The effects of this were especially apparent because we studied the fast-fourier transform of our concentrations, which tends to yield more precise peaks when a greater volume of data is provided.

For future work, we could model the reactions as dependent as opposed to independent. For example, we simply added our hydrogen peroxide to the existing rate equations. This would simulate the processes as being independent. For future work, it may be beneficial to instead multiply the rates together, which would correspond to the rates being dependent. We could also use deterministic simulations to characterize the exact domains of stability we would expect to see in this simulation.

References

(1) McCallum, G.; Potvin-Trottier, L. Using Models to (Re-)Design Synthetic Circuits. In *Synthetic Gene Circuits : Methods and Protocols*; Menolascina, F., Ed.; Springer US: New York, NY, 2021; pp 91–118. https://doi.org/10.1007/978-1-0716-1032-9_3.

(2) Potvin-Trottier, L.; Lord, N. D.; Vinnicombe, G.; Paulsson, J. Synchronous Long-Term Oscillations in a Synthetic Gene Circuit. *Nature* **2016**, *538* (7626), 514–517. https://doi.org/10.1038/nature19841.

(3) Elowitz, M. B.; Leibler, S. A Synthetic Oscillatory Network of Transcriptional Regulators. *Nature* **2000**, *403* (6767), 335–338. https://doi.org/10.1038/35002125.

(4) Milo et al. Nucl. Acids Res. (2010) 38 (suppl 1): D750-D753.

(5) Ansari, A. M.; Ahmed, A. K.; Matsangos, A. E.; Lay, F.; Born, L. J.; Marti, G.; Harmon, J. W.; Sun, Z. Cellular GFP Toxicity and Immunogenicity: Potential Confounders in in Vivo Cell Tracking Experiments. *Stem Cell Rev and Rep* **2016**, *12* (5), 553–559. https://doi.org/10.1007/s12015-016-9670-8.

(6) Ganini, D.; Leinisch, F.; Kumar, A.; Jiang, J.; Tokar, E. J.; Malone, C. C.; Petrovich, R. M.; Mason, R. P. Fluorescent Proteins Such as eGFP Lead to Catalytic Oxidative Stress in Cells. *Redox Biology* **2017**, *12*, 462–468. https://doi.org/10.1016/j.redox.2017.03.002.

(7) Glinkowska, M.; Łoś, J. M.; Szambowska, A.; Czyż, A.; Całkiewicz, J.; Herman-Antosiewicz, A.; Wróbel, B.; Węgrzyn, G.; Węgrzyn, A.; Łoś, M. Influence of the Escherichia Coli oxyR Gene Function on λ Prophage Maintenance. *Arch Microbiol* **2010**, *192* (8), 673–683. https://doi.org/10.1007/s00203-010-0596-2.

(8) Zhu, M.; Dai, X. Maintenance of Translational Elongation Rate Underlies the Survival of Escherichia Coli during Oxidative Stress. *Nucleic Acids Research* **2019**, *47* (14), 7592–7604. https://doi.org/10.1093/nar/gkz467.

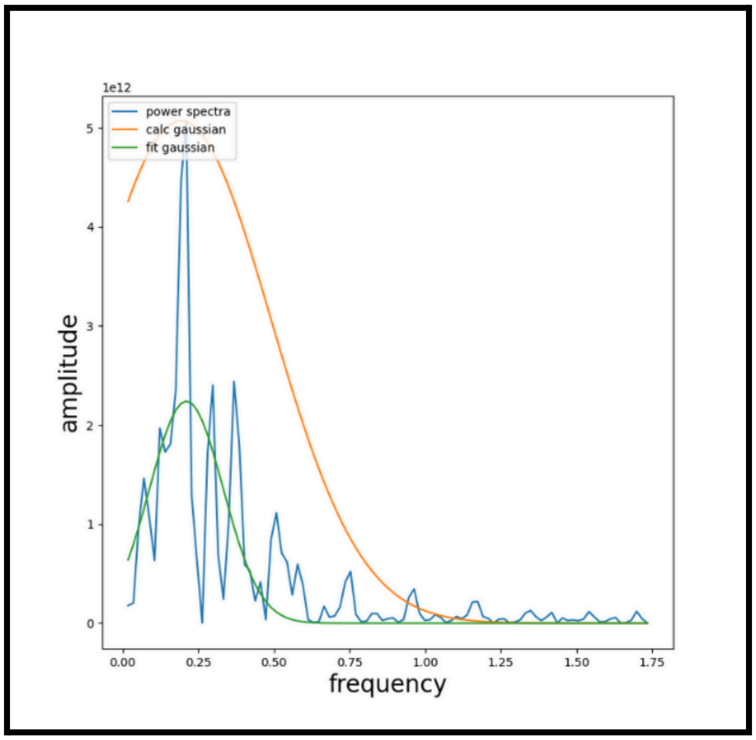
(9) Seaver, L. C.; Imlay, J. A. Hydrogen Peroxide Fluxes and Compartmentalization inside Growing *Escherichia Coli*. *J Bacteriol* **2001**, *183* (24), 7182–7189. https://doi.org/10.1128/JB.183.24.7182-7189.2001.

(10) Liu, H.; Yang, C.-L.; Ge, M.-Y.; Ibrahim, M.; Li, B.; Zhao, W.-J.; Chen, G.-Y.; Zhu, B.; Xie, G.-L. Regulatory Role of tetR Gene in a Novel Gene Cluster of Acidovorax Avenae Subsp. Avenae RS-1 under Oxidative Stress. *Front. Microbiol.* **2014**, *5*. https://doi.org/10.3389/fmicb.2014.00547.

Appendix A - Fourier Transform

Simply analyzing the time trace for each experiment does not typically yield meaningful insight. While these graphs are helpful for qualitatively understanding the behavior of a system, they are insufficient for characterizing the regimes under which the system is stable. As such, more sophisticated methods are required.

For each trial of our experiments, we extracted each individual time trace. We then performed a Fast Fourier Transform (FFT) to extract the power spectrum. This power spectrum represents the strength that each sine wave contributes to the overall function. This power spectrum was analyzed and is how our results were obtained. Many attributes about this power spectrum were analyzed, including the 50%, 68%, 95%, and 99% spread for each species, the mean period, the standard deviation, and a bell curve was fit to these distributions. While these results were interesting, they were ultimately omitted for clarity.



Appendix B - Code Snippet

The code for this project is extensive, and consequently, only a snippet will be included in this paper. However, all code can be found at the github linked here.
https://github.com/MarkAStevens04/JCP410

Paper_extension.py holds our gillespie simulation functions, and is the actual "experiment". We use main.py to run our experiment and perform our data analysis. WE use single_pass() as a harness to tell handle the experiment and gillespie simulation. The function full_gillespie() in main.py is what iterates through our RVF. Autocorrelate performs our data analysis (fast fourier transform). We then return the attributes we have received for that single trial.

multi.py is what runs our multithreaded application. Running a single time trace iteratively for all experiments would have taken over 250 hours. Instead, we ran experiments in parallel using over 56 CPU cores to bring the time down to only 10 hours. multi.py handled this multithreading and saved our results with h5py. Data_analysis takes the data imported from the cluster and displays the data in a readable graph.

```
import h5py
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import main

# = General Layout of Experiments =
# RVFs are for stochastic simulations, dYdt are for deterministic.
# We define wrappers to help quickly set up parameters.




#
---------------------------------------------------------------------
------------------------------------------------
# Get GFP up and running
def dYdt2(t, Y, p):
    # p: (lambda_m, lambda_p, beta_m, beta_p, h, K)
    # Y: [m1, P1, m2, P2, m3, P3, m4, P4]
```

```python
    m1 = (p[0] * (p[5] ** p[4])) / (p[5] ** p[4] + Y[5] ** p[4]) -
Y[0] * (p[2] + p[3])
    p1 = Y[0] * p[1] - Y[1] * p[3]
    m2 = (p[0] * (p[5] ** p[4])) / (p[5] ** p[4] + Y[1] ** p[4]) -
Y[2] * (p[2] + p[3])
    p2 = Y[2] * p[1] - Y[3] * p[3]
    m3 = (p[0] * (p[5] ** p[4])) / (p[5] ** p[4] + Y[3] ** p[4]) -
Y[4] * (p[2] + p[3])
    p3 = Y[4] * p[1] - Y[5] * p[3]
    m4 = (p[0] * (p[5] ** p[4])) / (p[5] ** p[4] + Y[5] ** p[4]) -
Y[6] * (p[2] + p[3])
    p4 = Y[6] * p[1] - Y[7] * p[3]
    return [m1, p1, m2, p2, m3, p3, m4, p4]

def rvf_gfp(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K] Parameter
vector for ODE solver
    :return:
    """
    # This is where you put in your RVF!
    # return a 1x12 matrix
    # print(f'running rvf {x}')
    # print(f'x at 0: {x[0]}')
    # POSSIBLE ERROR:
    # deg_m might be (beta_m * x[]) NOT (beta_m + beta_p) * x[]
    # print(f'p: {p}')
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]
    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]
    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2, prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4]


#
---------------------------------------------------------------------
-----------------------------------------------
# Test degradation interference


def rvf_gfp(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K] Parameter
vector for ODE solver
    :return:
    """
    # This is where you put in your RVF!
    # return a 1x12 matrix
    # print(f'running rvf {x}')
    # print(f'x at 0: {x[0]}')
    # POSSIBLE ERROR:
    # deg_m might be (beta_m * x[]) NOT (beta_m + beta_p) * x[]
    # print(f'p: {p}')
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]
    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2, prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4]




def gfp_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((8, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 100000,
K=7, stoich_mat=stoich_mat, rvf=rvf_gfp, x0_g=x0_g)
```

```
    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]

    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")

    return plt



#
----------------------------------------------------------------------
------------------------------------------------
# Adding Hydrogen Peroxide

def rvf_h2o2(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K] Parameter
vector for ODE solver
    :return:
    """
    # This is where you put in your RVF!
    # return a 1x12 matrix
    # print(f'running rvf {x}')
    # print(f'x at 0: {x[0]}')
    # POSSIBLE ERROR:
    # deg_m might be (beta_m * x[]) NOT (beta_m + beta_p) * x[]
    # print(f'p: {p}')
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = 10
    prod_h = (5835 * 60) + stoich * x[7]
    V_max = 1.2 * (10 ** 6) * 60
    K = 7.2*(10**17) * 60
    deg_h = 82.6 * x[8] * 60 + (x[8] * V_max) / (x[8] + K)
    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]





def h2o2_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 300000,
K=7, stoich_mat=stoich_mat, rvf=rvf_h2o2, x0_g=x0_g, p=["cat"])
```

```
    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    # plt.plot(rt, p1_r, label="P1", color="#f94144")
    # plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    # plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")
    plt.plot(rt, p5_r, label="P5", color="#c1121f")

    return plt


#
---------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 1A:
# TetR increases in concentration in response to Oxidative stress.
# Use regular rate constant to increase concentration of m3

def rvf_exp1a(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
    :return:
    """
    # alpha is the parameter we should change!
    # alpha = 0.01 semi stable, 0.1 unstable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]
    alpha = p[6]
    beta = p[7]
    # Example parameter values:
    # known: [160.02406557883805, 60.00902459206427,
3.6101083032490973, 1, 2, 7, 10, 0.01]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h) + beta *
x[8]
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]




def exp1a_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 100000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp1a, x0_g=x0_g, p=[10, 0.01])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt



#
---------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 1B:
# TetR increases in concentration in response to Oxidative stress.
# Use Hill Equation to increase concentration of m3

def rvf_exp1b(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
    :return:
    """
    # beta is the parameter we should change! Corresponds to
production of mRNA-3
    # beta = 1000 semi stable, 100 stable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]
    alpha = p[6]
    beta = p[7]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h) + (beta * K
** h) / (K ** h + x[8] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]
    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]




def exp1b_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 600000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp1b, x0_g=x0_g, p=[10, 100])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt
```

```
#
----------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 2A:
# E Coli decreases global translation under oxidative stress.
# Simulate this by increasing degradation of mRNA in response to
H2O2.
# DOES NOT include degradation of GFP mRNA

def rvf_exp2a(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
    :return:
    """
    # beta is the parameter we should change! Corresponds to mRNA
degradation rate
    # beta = 0.01 semi unstable, 0.1 unstable, 0.001 semi unstable,
0.0001 semi unstable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]

    alpha = p[6]
    beta = p[7]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0] + x[8] * beta

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]
    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2] + x[8] * beta

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h) + p[6] *
x[8]
    deg_m3 = (beta_m + beta_p) * x[4] + x[8] * beta

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]




def exp2a_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 1000000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp2a, x0_g=x0_g, p=[10, 0.0001])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt



#
---------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 2B:
# E Coli decreases global translation under oxidative stress.
# Simulate this by increasing degradation of mRNA in response to
H2O2.
# DOES include degradation of GFP mRNA

def rvf_exp2b(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
    :return:
    """
    # beta is the parameter we should change! Corresponds to mRNA
degradation rate
    # beta = 0.01 semi stable, 0.1 unstable, 0.001 mostly stable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]

    alpha = p[6]
    beta = p[7]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0] + x[8] * beta

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2] + x[8] * beta

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h) + p[6] *
x[8]
    deg_m3 = (beta_m + beta_p) * x[4] + x[8] * beta

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6] + x[8] * beta

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]




def exp2b_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]
    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 1000000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp2b, x0_g=x0_g, p=[10, 0.0001])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt

#
---------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 3A:
# Lambda cI increases in response to Oxidative stress.
# Use regular rate constant to increase concentration of m1

def rvf_exp3a(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
    :return:
    """
    # beta is the parameter we should change! Corresponds to mRNA
production in relation to H2O2
    # beta = 0.1 unstable, 0.001 pretty much stable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]

    alpha = p[6]
    beta = p[7]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h) + beta * x[8]
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]
    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]




def exp3a_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 1000000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp3a, x0_g=x0_g, p=[10, 0.01])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt




#
```

```
----------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 3B:
# Lambda cI increases in concentration in response to Oxidative
stress.
# Use Hill Equation to increase concentration of m1

def rvf_exp3b(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
    :return:
    """
    # beta is the parameter we should change! Corresponds to mRNA-1
transcription in response to H2O2
    # beta = 1000 unstable, = 100 stable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]

    alpha = p[6]
    beta = p[7]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h) + (beta * K **
h) / (K ** h + x[8] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]
```

```
def exp3b_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 1000000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp3b, x0_g=x0_g, p=[10, 1000])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]
    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt




#
---------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 4A:
# Most vulnerable to a feedback loop
# Use regular rate constant to increase concentration of m2

def rvf_exp4a(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
    :return:
    """
    # beta is the parameter we should change! Corresponds to mRNA-2
transcript rate in response to H2O2
    # beta = 0.1 unstable, 0.001 pretty much stable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]
    alpha = p[6]
    beta = p[7]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h) + beta * 
x[8]
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in 
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]




def exp4a_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
```

```
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 10000000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp4a, x0_g=x0_g, p=[10, 0.1])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt



#
---------------------------------------------------------------------
------------------------------------------------
# Idea of Experiment 4B:
# Most vulnerable to a feedback loop
# Use Hill Equation to increase concentration of m2

def rvf_exp4b(x, p):
    """
    Calculates the rates of rxns using current quantities of each
species
    :param x: [m1, P1, m2, P2, m3, P3, m4, P4, h] (array of molecule
numbers)
    :param p: [lambda_m, lambda_p, beta_m, beta_p, h, K, alpha, beta]
Parameter vector for ODE solver
```

```
    :return:
    """
    # beta is the parameter we should change! Corresponds to mRNA-2
transcript rate in response to H2O2
    # beta = 1000 unstable, = 100 stable
    lambda_m = p[0]
    lambda_p = p[1]
    beta_m = p[2]
    beta_p = p[3]
    h = p[4]
    K = p[5]

    alpha = p[6]
    beta = p[7]


    prod_m1 = (lambda_m * K**h) / (K**h + x[5] ** h)
    deg_m1 = (beta_m + beta_p) * x[0]

    prod_p1 = lambda_p * x[0]
    deg_p1 = beta_p * x[1]

    prod_m2 = (lambda_m * K ** h) / (K ** h + x[1] ** h) + (beta * K
** h) / (K ** h + x[8] ** h)
    deg_m2 = (beta_m + beta_p) * x[2]

    prod_p2 = lambda_p * x[2]
    deg_p2 = beta_p * x[3]

    prod_m3 = (lambda_m * K ** h) / (K ** h + x[3] ** h)
    deg_m3 = (beta_m + beta_p) * x[4]

    prod_p3 = lambda_p * x[4]
    deg_p3 = beta_p * x[5]

    prod_m4 = (lambda_m * K ** h) / (K ** h + x[5] ** h)
    deg_m4 = (beta_m + beta_p) * x[6]

    prod_p4 = lambda_p * x[6]
    deg_p4 = beta_p * x[7]

    # * 60 b/c we're in molecules / sec and we need to be in
molecules/min
    # stoich is number of hydrogen peroxide generated per GFP.
    stoich = alpha
    # h_generation = 5835 * 60
    # V_max = 1.2 * (10 ** 6) * 60
    # K = 7.2 * (10 ** 17) * 60
    # h_degrade = 82.6 * 60
    h_generation = 5835
    V_max = 1.2 * (10 ** 6)
    K = 7.2 * (10 ** 17)
    h_degrade = 82.6

    prod_h = h_generation + stoich * x[7]
    deg_h = h_degrade * x[8] + (x[8] * V_max) / (x[8] + K)

    return [prod_m1, deg_m1, prod_p1, deg_p1, prod_m2, deg_m2,
prod_p2, deg_p2,
            prod_m3, deg_m3, prod_p3, deg_p3, prod_m4, deg_m4,
prod_p4, deg_p4,
            prod_h, deg_h]




def exp4b_wrapper_stoch(plt):
    """
    Performs a run including gfp.
    Uses rvf_gfp
    :return: Updated plt
    """
    # reactions: prod_m1, deg_m1, prod_P1, deg_P1, prod_m2, deg_m2,
prod_P2, deg_P2, prod_m3, deg_m3, prod_P3, deg_P3, prod_m4, deg_m4,
prod_P4, deg_P4, prod_h2o2, deg_h2o2
    stoich_mat = [
        [1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1]]

    x0_g = np.zeros((9, 1))
    x0_g[0, 0] = 10

    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    # *** NOTE: H should be set to 2, but is set to 1 by default!!
***
    rt, rx, peaks, autoc = main.single_pass(0.277, 380, 2, 1000000,
K=7, stoich_mat=stoich_mat, rvf=rvf_exp4b, x0_g=x0_g, p=[10, 1000])

    # - Regular Graph -
    p1_r = rx[1, :]
    p2_r = rx[3, :]
    p3_r = rx[5, :]
    p4_r = rx[7, :]
    p5_r = rx[8, :]

    plt.plot(rt, p5_r, label="P5", color="#c1121f")
    plt.plot(rt, p1_r, label="P1", color="#f94144")
    plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    plt.plot(rt, p3_r, label="P3", color="#577590")
    plt.plot(rt, p4_r, label="P4", color="#06d6a0")


    return plt



#
---------------------------------------------------------------------
```

```python
def plot_gfp_stoch():
    """
    Create time trace graph from algorithm run.
    :return:
    """



    plt.figure(figsize=(8, 8))
    # h2o2_wrapper_stoch(plt)
    # gfp_wrapper_stoch(plt)
    # exp1a_wrapper_stoch(plt)
    exp1b_wrapper_stoch(plt)
    # exp2a_wrapper_stoch(plt)
    # exp2b_wrapper_stoch(plt)
    # exp3a_wrapper_stoch(plt)
    # exp3b_wrapper_stoch(plt)
    # exp4a_wrapper_stoch(plt)
    # exp4b_wrapper_stoch(plt)

    # plt.plot(rt, p1_r, label="P1", color="#f94144")
    # plt.plot(rt, p2_r, label="P2", color="#f9c74f")
    # plt.plot(rt, p3_r, label="P3", color="#577590")
    # plt.plot(rt, p4_r, label="P4", color="#06d6a0")

    plt.xlabel("Time", fontsize=15)
    plt.ylabel("Copy Number", fontsize=15)
    plt.title("GFP Stochastic", fontsize=15)

    plt.legend(loc="upper left")
    plt.subplots_adjust(bottom=0.1, left=0.1)

    # - Autocorrelation graph -
    # plt.plot(rt, autoc)

    plt.show()
```

```
# *** NOTE: H should be set to 2, but is set to 1 by default!! ***



if __name__ == "__main__":
    # plot_gfp_det()
    plot_gfp_stoch()
```