**May 11, 2022 – Hands-On Intro to the Basics of Scientific Programming**
**Prof. Joshua Weitz and QBioS Cohort**

Would you like to be able to build a computational model of a living system but don't know how? Then you are in the right spot.

This tutorial is meant to be interactive... that is *you* should be reading, typing, thinking, and asking questions. There are no source files to share, the point is for you to think about and enter the code and see the result. And then, to vary the code and see what changes. This is a more effective strategy than just *copying* code others have written and running it.

The materials are adapted from a semester long course entitled "Foundations of Quantitative Biosciences" developed by Prof. Joshua Weitz in Fall 2016/2017/2018 as the cornerstone class for the QBioS Ph.D. at Georgia Tech. The materials have been adapted to focus on stochastic models and to account for the greater variety of backgrounds of students in this week's workshop.

Today we will focus on the basics of coding that can help you build models... whether of gene expression, cellular dynamics, game theory, or some other problem linked to dynamics of living systems. A few notes before we get started:

- If you are experienced in coding, then please work on challenge problems identified in this tutorial.
- Please choose one language - MATLAB or Python - and stick with it for the two day workshop.
- Please work on your own computer, save files for future reference, but also - work with a partner so you can discuss and help each other as you go.

Let's get started!

# 1 Getting Started

Python is a dynamic programming language with a vast region of applications due its diverse library of packages. We will be using it in this class to simulate and model the dynamics of living systems from scales of molecules to ecosystems. Python is particularly good at:

- Building Multi Purpose Applications
- Web Development
- Rapid Prototyping

and is okay at:

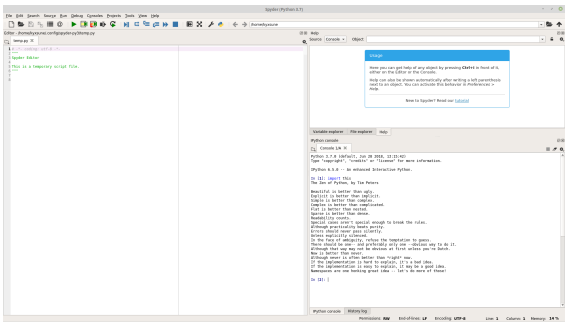
- Interfacing with other languages
- Speed of for loops (as compared to C)

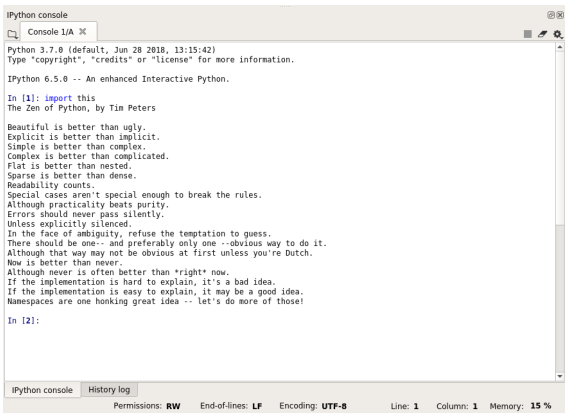
There are a variety of Integrated Development Environments(IDEs) for Python. For this course we will use Spyder. So, let's get started! In today's class, you will gain practical experiencing using Python. **If you are uncertain how to launch jupyter, try searching for "anaconda" using the search bar**. When you open the anaconda manager select Spyder, using the desktop link or application manager, then you should see a window that looks like this:

We will be doing most of our work in the Console, the region to the bottom right. The Console is where you enter commands, and you should see a prompt that looks like this:

```
In [ ]:
```

You can do basic math at this prompt (in order to run the cell press Shift+Enter), for example






```
In [ ]: 3+4
Out[ ]: 7
```

and (using the numpy package)

```
In [ ]: import numpy as np
        np.exp(1)
Out[ ]: 2.7182818284590451
```

Python's greatest strength is the diverse library of packages available to it. There are an incredible amount of packages that are useful for everything from 3D design to Genomics. Making use of this diverse set of packages requires that they be imported (loaded), into the current workspace. As long as the package is installed, Anaconda contains all the packages for this workshop, importing them is simple. Remember, that in every new session you must import the packages you wish to use.

```
In [ ]: import numpy as np
```

In addition to importing with the **import** command, we also use the **as** command to shorten the name. From numpy to np. Once a package is imported, it is made available for every cell downstream. Therefore it is good practice to have import statements in the first cell. Python has several functions that are built in (and you can use it like a calculator). Numpy (a python package), has **many** more functions for advanced calculation, for example, to learn more about the exponential function:

```
In [ ]: np.info(np.exp)
 exp(x[, out])

 Calculate the exponential of all elements in the input array.
```

```
Parameters
----------
x : array_like
Input values.

Returns
-------
out : ndarray
Output array, element-wise exponential of `x'.

See Also
--------
expm1 : Calculate ``exp(x) - 1'' for all elements in the array.
exp2  : Calculate ``2**x'' for all elements in the array.

Notes
-----
The irrational number ``e'' is also known as Euler's number.  It is
approximately 2.718281, and is the base of the natural logarithm,
``ln'' (this means that, if :math:`x = \ln y = \log_e y',
then :math:`e^x = y'. For real input, ``exp(x)'' is always positive.

For complex arguments, ``x = a + ib'', we can write
:math:`e^x = e^a e^{ib}'.  The first term, :math:`e^a', is already
known (it is the real argument, described above).  The second term,
:math:`e^{ib}', is :math:`\cos b + i \sin b', a function with
magnitude 1 and a periodic phase.

References
...
```

**For numpy functions use np.info() , for base python packages use help()**

Many functions have names that you expect (how do you think you should calculate cosine or sine of a value, for example)? Try it out! If you don't know the name of the function you can use the command np.lookfor or help.

Python is not just a calculator. It is also a programming language that can store values in memory. For example, the command

```
In [ ]: x=3
        x
Out [ ]: 3
```

tells Python that the variable x has the value 3 and now every time you use "x", Python will substitute the value 3, for example:

```
In [ ]: y=x+1
        print(y)
Out [ ]: 4
```

Note that if you don't want Python to report back the answer/value you can either call the variable or use a print() function. It is very important to realize the "=" sign does not mean that Python checks to see if the two sides are equal to each other, but rather states that whatever is on the left should be assigned the value of that on the right. If you want to check the truth of a particular statement, for example is x equal to 3, or alternatively, is y equal to 3 then you would type:

```
In [ ]: x==3
Out [ ]: True

In [ ]: y==3
Out [ ]: False
```

The double "==" sign tells Python to logically compare what is on the left with that on the right and return 1 if true and 0 if false.

Python can also handle arrays of values (for example a vector or a matrix). The simplest way is to use the numpy arange function, which defines a sequential vector, for example

```
In [ ]:v = np.arange(1,5)
       print(v)
Out [ ]: [1 2 3 4]
```

you can also modify the increments by adding a step parameter at the end

```
In [ ]: w = np.arange(1,9,2)
        print(w)
Out [ ]: [1 3 5 7]
```

Any entry can be examined using the brackets

```
In [ ]: w[3]
Out [ ]: 7
```

and basic math can be performed automatically on vectors (and matrices), for example

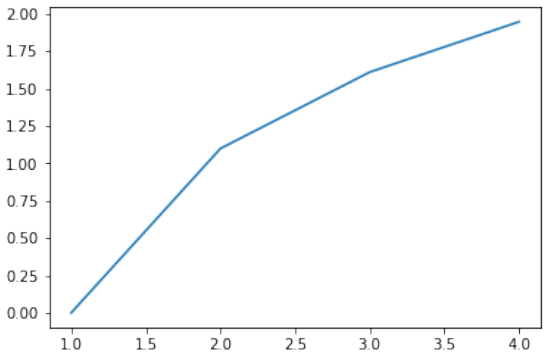
```
In [ ]: np.log(w)
out [ ]: array([ 0.        ,  1.09861229,  1.60943791,  1.94591015])
```

**Python lists/arrays are 0 indexed, meaning the first element's index is 0 not 1**

Python using the Matplotlib library can also plot graphs and surfaces and all sorts of things. To create a simple plot, use the plot command.

```
In [ ]: import matplotlib.pyplot as plt
        plt.plot(v,np.log(w))
```

which leads to:



# 2 Building "Programs" from "Scripts" and "Functions"

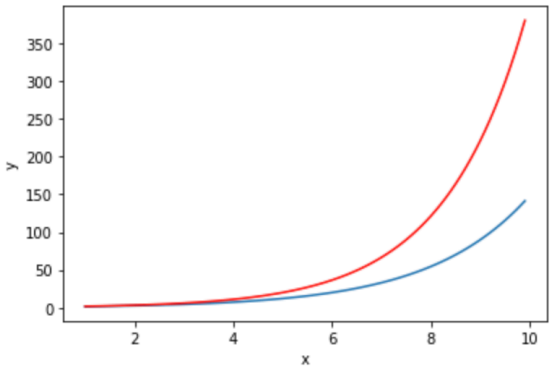
However, once you have a lot of commands, it will get exhausting typing them (especially when you make mistakes). So, Python has functions, which is a list of commands, that execute in order. To create a function create a new cell and define a function using def.

```
# My first function
def func():
    x = np.arange(1,10)
    y1 = np.exp(x*.5)
    y2 = np.exp(x*.6)
    plt.plot(x,y1)
    plt.plot(x,y2,color='red')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.show()
```

Run the cell this is placed in, and then in the next cell put.

```
In [ ]: func()
```

and it should do all the commands in your script and give you your figure.




The problem with this function is that if you wanted to change the exponents in these files, you would have to edit the function and then re-run it, instead of designating the change in the current cell. Moreover, this function does not return a variable or accept a variable as input.

```
def logGrowth(t,N):
    '''
    logGrowth gives the growth rate of a population of size N at time t
    '''
    r = .5
    K = 100
    dNdt = r*N*(1-N/K)
    return(dNdt)
```

Now you can use your function like one of Python's, for example:

```
In [ ]: j = np.arange(110)
        plt.plot(j,logGrowth(0,j))
        plt.xlabel('N')
        plt.ylabel('dN/dT')
        plt.show()
```

gives an upside down parabola, denoting that growth rate is positive between 0 and 100 and negative when N is greater than 100.

# 3 Getting Started with Core Techniques

## 3.1 Create loops

The two types of loops we will discuss are "for" and "while" loops. Both loop start with a keyword such as for or while and they end with the word end. The "for" loop allows us to repeat certain commands many times with a "counter" variable. Here is one example:

```
In [ ]: for i in np.arange(1,4):
            print(1-i)
```

You can also increment your counter variable by any real number like

```
In [ ]: for i in np.arange(1,0,-.1):
            print(1-i)
```

Or you can even use a set of arbitrary values:

```
In [ ]: for i in [1,3,5,7]:
        print(1-i)
```

**Challenge problem: Exercise on matrices**

Define a random matrix A of size 3-by-3. Use a double "for loop" to calculate the square of the entries in A and store the values in another matrix B. (Hint: type np.lookfor('random') if you don't know how to define a random matrix)

The second type of loop is the "while" loop. The "while" loop repeats a sequence of commands as long as some condition is met. For example, given a number $n$, the following code will return the smallest non-negative integer $a$ such that $2^a \geq n$.

```
In [ ]: def smallexp(n):
            a = 0
            while 2**a < n:
                a = a+1
            return(a)
In [ ]: a = smallexp(4)
        print(a)
Out [ ]: 2
```

Note that in the above example we used the conditional statement, $2^a < n$ to decide whether the statement within the while loop should be proceeded. Such conditional statements are also used in "if" statements that are discussed in the next section. The rational operator used here is "<", which means "less than." Other rational operators that are available in Python include:

```
>  greater than
>= greater than or equal
<= less than or equal
== equal
!= not equal
```

Simple conditional statements can be combined by logical operators 'and' and 'or' into compound expressing such as the following:

```
(5 > 1) and (5 == 6)
(5 > 1) or (5 == 6)
```

## 3.2 Make decisions

Now, let's suppose you want your code to make a decision, and the "if" statement is what you need. The general form in Python is as follows:

```
if expression1:
  statements1
elif expression2:
  statements2
else:
  statements3
```

**Challenge problem: Recursive factorials**

Exercise: Finish the following pseudo-code that gives the factorial of a positive number $n$, using the recursive formula $n! = n(n-1)$.

```
def factorial_recur(n):
    if #:
        N = 1
    else:
        N = #
    return(N)
```

## 3.3 Go fast, i.e., "vectorize"

Remember: `for` loops are SLOW. One way to make your Python code run faster is to "vectorize" the algorithm you use in the code. Vectorization can be done by converting for and while loops to equivalent vector or matrix operations. A simple example would be the following timeit is a magic command that captures the runtime of the cell in jupyter. Commands on the same line as timeit are not included in the timer. The double percent sign is for jupyter notebook.

```
In [ ]: %%timeit x = 1 ; y = []
        for i in np.arange(1001):
            y.append(np.log10(x))
            x = x + .01
In [ ]: %%timeit
        x = np.arange(.01,10,.01)
        y = np.log10(x)
```

For more complicated code, vectorization is not always so obvious. Nonetheless, there is a simple procedure that will help: elementwise operators.

## 3.4 Elementwise operators

In Python, you can do calculations element by element. For example, if you have a variable `X` and want to square it. In Python squaring is denoted by ** instead.

```
In [ ]: x = np.array([5,5,5,5])
        x^2
```

Did it work? Yes! In numpy elementwise multiplication it is the default behavior to interpret the subsequent operator element by element. This is also helpful for a whole matrix. If you would like to do dot multiplication, simply use @ operator

```
In [ ]: x = 5 * np.ones((4,4))
        x @ x
```

Contrast the two answers. In the first, you squared all the 5-s. In the second you multiplied two square matrices by each other.

**Challenge problem: Element-wise operations: square root**

Take the square root of the numbers between 1 to 20 using an elementwise operator.

## 3.5 Find values you want to know about

Given an array X, the command

```
In [ ]: x[x!=0]
```

returns the indices of all nonzero elements of X. You can also use a logical expression to define X. For example,

```
In [ ]: x[x>2]
```

returns the indices corresponding to the entries of X that are greater than 2. Notice that X could also be a matrix. In this case, using the np.argwhere() returns a list of the elements that satisfy the condition.

```
In [ ]: np.argwhere(x>2)
```

However, when you want to locate the entries that satisfy more than one logical expressions, the 'element-wise' logical operators (& and |) are used in place of 'short-circuit' logical operators (&& and ||).

**Challenge problem: Finding entries in matrices**

Exercise: Build a 5-by-5 random matrix A using the command

```
In [ ]: A = np.random.rand(5,5)
```

Find the linear indices of entries whose value is smaller than 1/4 and bigger than 1/6. Check the answer with your neighbors.

### 3.6 Save and load data

Saving and loading objects (variables,arrays, or other forms of data) can be done in a multitude of ways in Python, however; for this course we will use the native methods for numpy.

```
In [ ]: import numpy as np
        numpy_array = np.array([10,20,30,40])
        np.save('numpy_store.npy',numpy_array)
```

and numpy will save the specified array in the file name. When you want to load all the variables from the file specified by filename, just type

```
In [ ]: numpy_loaded = np.load('numpy_store.npy')
```

## 4 Dynamics

The stochastic model of gene expression that we will study during this workshop is an extension of a deterministic model that consists of ordinary differential equations (ODEs). Some basic concepts from the theory of ODEs will be helpful in understanding how the averages of your stochastic simulations evolve in time, so we cover them here. The term "system" is used here to describe a mathematical model consisting of any number of differential equations.

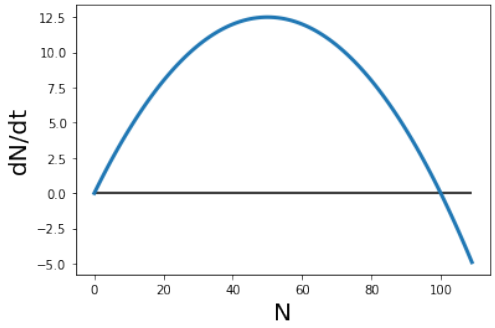
### 4.1 One-dimensional systems & bistability

A nice example of a one-dimensional ODE is that of logistic growth:

$$\frac{\mathrm{d}N}{\mathrm{d}t} = rN\left(1 - \frac{N}{K}\right)$$

where $N = N(t)$ is a variable describing a population size as a function of time, $r > 0$ is a constant parameter that describes the population growth rate, and $K > 0$ is a constant parameter describing the carrying capacity of the population. It is instructive to plot this equation directly along with the horizontal line representing $\mathrm{d}N/\mathrm{d}t = 0$:

```
In []:  vN = np.arange(0,110)
        plt.plot(vN,logGrowth(vN,0),linewidth=3)
        plt.hlines(0,np.min(vN),np.max(vN))
        plt.xlabel('N',fontsize=20)
        plt.ylabel('dN/dt',fontsize=20)
        print(np.where(logGrowth(vN,0)==0)) # print the fixed points
Out []: (array([  0, 100]),)
```



The above plot is called the **phase portrait**. From the phase portrait, it is clear graphically that the derivative peaks when the population is equal to half of the carrying capacity $N = K/2$, with a value of $\mathrm{d}N/\mathrm{d}t = rK/4$. The population sizes where the derivative is equal to zero $\mathrm{d}N/\mathrm{d}t = 0$ are of special interest. They are called **fixed points**, let us denote them $N^*$; they can be found algebraically by setting the above equation to zero and solving for $N^*$:

$$0 = rN^*\left(1 - \frac{N^*}{K}\right)$$

$$N^* = 0 \text{ or } K$$

The fixed point $N^* = 0$ is **unstable**, since small deviations from it result in a population size that tends away from the fixed point. On the other hand, $N^* = K$ is a **stable** fixed point because small deviations from it result in population sizes that tend back to $N^* = K$. This model is **monostable**, since there is only one stable fixed point. In this model of population growth, the stable fixed point is the carrying capacity $K$ for any choice of parameter values for $r$ and $K$.

**In the following, the words "gene" and "protein" are used interchangeably. Don't get caught up in the biological details. The emphasis here is how the notion of genetic bistability can be understood in terms of mathematical models.**

Example: Stability Analysis

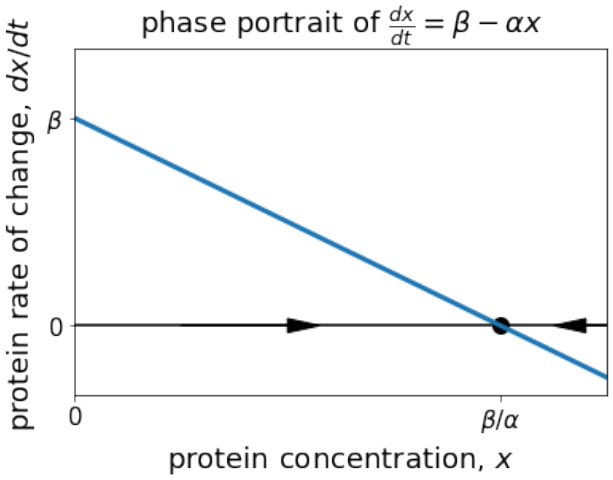
Find the fixed point(s) and their stability for the following 1-dimensional system. Sketch the phase portrait. Is the system monostable?

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \beta - \alpha x$$

where $\beta > 0$ and $\alpha > 0$ are constant parameters representing production and degradation, respectively; $x > 0$ is a variable representing the concentration of protein $X$. The phase portrait is depicted below, along with the python code that produced it. Don't worry too much about understanding the code; it is there only for your reference. The important part to understand is why the phase portrait looks the way it does for this particular system.

**Example: Stability Analysis**

```
In []:  import numpy as np
        import matplotlib.pyplot as plt

        # define what will be plotted
        xVec = np.linspace(0,5,6) # defines vector of x values to plot derivative over
        dxdt = lambda x, alpha, beta : beta-alpha*x # defines the derivative
        # set parameter values
        alpha = 3./4.
        beta = 3.
        # make the plot
        xlocs, xlabels = plt.xticks()  # Get the current locations and labels of xticks
        plt.xticks([0,beta/alpha], ['0', r'$\beta/\alpha$'],fontsize=15)  # set labels for xticks
        plt.xlim([0,beta/alpha+1]) # set limits of x
        plt.xlabel(r'protein concentration, $x$',fontsize=18)
        ylocs, ylabels = plt.yticks()  # Get the current locations and labels of yticks
        plt.yticks([0,beta], ['0', r'$\beta$'],fontsize=15)  # set labels for yticks
        plt.ylim([-1,beta+1]) # set the limits of y
        plt.ylabel(r'protein rate of change, $dx/dt$',fontsize=18)
        plt.hlines(0,0,xVec[-1]) # plot the horizontal line dx/dt=0
        FP_x=np.where(dxdt(xVec, alpha, beta)==0)[0][0] # find the x-coordinate of the fixed point
        FP_y=dxdt(FP_x, alpha, beta) # y-coordinate of fixed point (0)
        FP=[FP_x, FP_y] # assemble the fixed point as an ordered pair
        #print(FP) # double check that it is correct
        # plot the stable fixed point with a filled circle
        plt.plot(FP_x, FP_y, linestyle='', marker='o', color='black', markersize=10)
        plt.plot(dxdt(xVec, alpha, beta), linewidth=3) # plot the line representing the derivative
        plt.arrow(1,0,1,0,head_width=0.2,color='black') # derivative > 0 for x<beta/alpha
        plt.arrow(xVec[-1],0,-0.2,0,head_width=0.2,color='black') # derivative < 0 for x>beta/alpha
        plt.title(r'phase portrait of $\frac{dx}{dt}=\beta-\alpha x$',fontsize=18)
        plt.show()
Out []:
```




The problem above demonstrates a simple model of gene expression where protein is produced at a constant rate $\beta$ and degrades at a rate $\alpha x$ proportional to the instantaneous amount of protein present.

The phase portrait for this system is a downward-sloping line that intersects the $\mathrm{d}x/\mathrm{d}t = 0$ axis once at the protein concentration $x = \beta/\alpha$, which is a stable fixed point. Since this is the only stable fixed point for any choice of parameter values for $\alpha$ and $\beta$, this system is monostable.

We now study a 1-dimensional model of **bistability**. This bistability requires that protein production become a function of some variable, instead of being a constant rate $\beta$, as in exercise 5.1. Suppose that gene $X$ is regulated by its own concentration. We study the system:

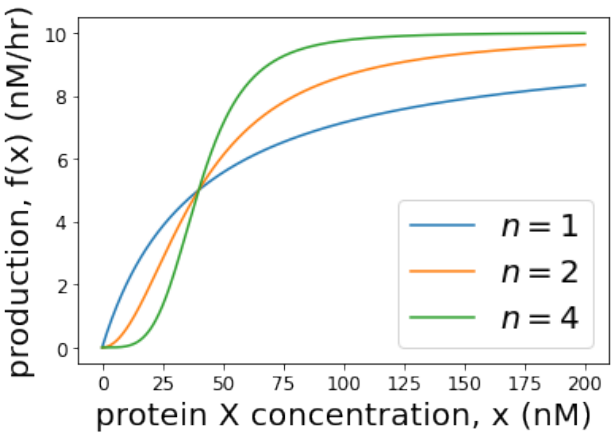
$$\frac{\mathrm{d}x}{\mathrm{d}t} = f(x) - \alpha x$$

where $x$ is the concentration of protein $X$ and $\alpha > 0$ is the rate of degradation of $x$. If we constrain $f(x)$ to be monotonic, it will either be a strictly increasing or a strictly decreasing function of $x$. To start, let's focus on the increasing case. Consider a production function of the form:

$$f(x) = \frac{\beta x^n}{K^n + x^n}$$

To get an intuition for this function, let us define it in python and make some plots for various parameter values. For simplicity, let us fix the maximum production rate at $\beta = 10$ nM/hr and the half-activation concentration at $K = 40$ nM, while varying the cooperativity parameter $n$. Note that all of these parameters are taken to be positive numbers.

```
In []:  import numpy as np
        import matplotlib.pyplot as plt

        def activator(x, beta, K, n):
            """
            function f(x) = activator(x, beta, K, n)
            gives the production rate as a function of the concentration of protein X
            x is the concentration of protein X
            beta is the maximum production rate of protein X
            K is the half-activation concentration of protein X
            n is the cooperativity parameter, which modulates the sharpness of the increase of f(x)
            """
            f = (beta*x**n)/(K**n+x**n)
            return f

        # study the effect of parameter n
        xVec = np.linspace(0,200,400) # defines vector of x values to plot f over
        fVec_nEq1 = activator(xVec, beta=10, K=40, n=1) # define f(x) with n=1
        fVec_nEq2 = activator(xVec, beta=10, K=40, n=2) # define f(x) with n=2
        fVec_nEq4 = activator(xVec, beta=10, K=40, n=4) # define f(x) with n=4
        plt.plot(xVec, fVec_nEq1, label=r'$n=1$') # plot for n=1
        plt.plot(xVec, fVec_nEq2, label=r'$n=2$') # plot for n=2
        plt.plot(xVec, fVec_nEq4, label=r'$n=4$') # plot for n=4
        plt.legend(fontsize=20) # add a figure legend
        plt.xlabel('protein X concentration, x (nM)', fontsize=20) # x-axis label
        plt.ylabel('production, f(x) (nM/hr)', fontsize=20) # y-axis label
        plt.xticks(fontsize=12) # bigger ticks on horizontal axis
        plt.yticks(fontsize=12) # bigger ticks on vertical axis
Out []: (array([-2.,  0.,  2.,  4.,  6.,  8., 10., 12.]),
        <a list of 8 Text major ticklabel objects>)
```




From the plot above, it is clear that increasing the cooperativity parameter $n$ yields a production curve that increases at a faster rate with respect to $x$. This curve is bounded between $f(x) = 0$ and $f(x) = \beta = 10$. The parameter $\beta$ therefore determines the maximum production rate. At $x = K = 40$, the curve reaches the half-activation point $f(x) = \beta/2 = 10/2 = 5$. So we may call $K$ the half-activation concentration.

Just as we did for the logistic model, let us now analyze $\mathrm{d}x/\mathrm{d}t = f(x) - \alpha x$ with the activator function defined above. The fixed points $x^*$ result when the derivative is zero $\mathrm{d}x/\mathrm{d}t = 0$, which occurs when

$$0 = f(x^*) - \alpha x^*$$

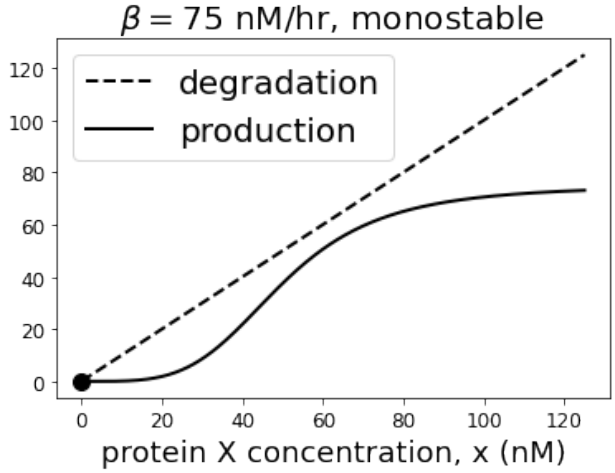
$$\alpha x^* = f(x^*)$$

$$\alpha x^* = \frac{\beta (x^*)^n}{K^n + (x^*)^n}$$

Although this equation can be solved algebraically for $x^*$, let us take a graphical approach. We plot the left-hand side of the above equation and the right-hand side separately. The location(s) where these curves intersect yield the solution(s) $x^*$ of the fixed point equation above.

```
In []:  ## set fixed parameter values
        alpha=1 #1/hr
        n=4 # dimensionless
        K=50 # nM

        ## case 1: monostable regime
        beta=75 # nM/hr
        xVec=np.linspace(0,125,400) # define vector of x values to plot over
        degr=alpha*xVec # straight line of slope 'alpha' going
        through the origin)
        prod=activator(xVec, beta, K, n) # define the production curve (given by f)
        plt.plot(xVec, degr, linestyle='--', linewidth=2, color='black', label='degradation')
        plt.plot(xVec, prod, linestyle='-',  linewidth=2, color='black', label='production')
        plt.legend(fontsize=20) # add a figure legend
        plt.xlabel('protein X concentration, x (nM)', fontsize=18) # x-axis label
        plt.title(r'$\beta=75$ nM/hr, monostable', fontsize=20) # slap a title on there
        plt.xticks(fontsize=12) # bigger ticks on horizontal axis
        plt.yticks(fontsize=12) # bigger ticks on vertical axis
        FPs_x=np.where(prod==degr) # find the fixed points (x-coordinates)
```

```
        FPs_y=alpha*FPs_x # y-coordinates of the fixed points
        plt.plot(FPs_x, FPs_y, linestyle='', marker='o', color='black', markersize=10)
Out []: [<matplotlib.lines.Line2D at 0x7f863520b750>]
```

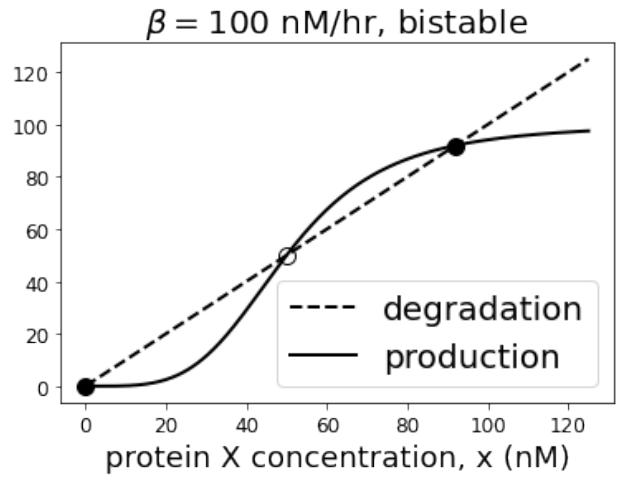



There is a single fixed point at $x = 0$. What is its stability? This can be answered by inspecting the sign of the derivative in a small neighborhood around the fixed point. Since the derivative is given by $\mathrm{d}x/\mathrm{d}t = \text{production} - \text{degradation}$ and $\text{production} - \text{degradation} < 0$ in a small neighborhood to the right of the fixed point in the plot above, the protein concentration $x$ decreases toward this fixed point. Therefore, this fixed point is stable and the system is monostable in this parameter regime with $\beta = 75$.

As the name implies, a **bistable** system is one with two coexisting stable states. How might we render this system to be bistable? Remember that the parameter $\beta$ sets the maximum of the production curve. So if we increase $\beta$ from 75, then then production curve shown above will have a higher maximum. If increased enough, that curve will intersect the dashed degradation line while maintaining its previous intersection at the origin. This will create two new fixed points, rendering the system bistable. Let's see this in action, through some python code:

```
In []:  ## case 2: bistable regime
        beta=100 # nM/hr
        xVec=np.linspace(0,125,400) # define vector of x values to plot over
        degr=alpha*xVec # define the degradation curve
        prod=activator(xVec, beta, K, n) # define the production curve (given by f)
        plt.plot(xVec, degr, linestyle='--', linewidth=2, color='black', label='degradation')
        plt.plot(xVec, prod, linestyle='-',  linewidth=2, color='black', label='production')
        plt.legend(fontsize=20) # add a figure legend
        plt.xlabel('protein X concentration, x (nM)', fontsize=18) # x-axis label
        plt.title(r'$\beta=100$ nM/hr, bistable', fontsize=20) # slap a title on there
        plt.xticks(fontsize=12) # bigger ticks on horizontal axis
        plt.yticks(fontsize=12) # bigger ticks on vertical axis
        FPs_x=xVec[np.where(np.abs(prod-degr)<0.15)[-1]] # find the fixed points (x-coordinates)
        print(FPs_x) # inspect it by printing
        FPs_x=FPs_x[:-1] # chop off the last one, it is repetitive
        print(FPs_x) # check that the redundant one got removed
        FPs_y=alpha*FPs_x # y-coordinates of the fixed points
        stableFPs_x=[FPs_x[0], FPs_x[2]] # x coordinates of the stable fixed points
```

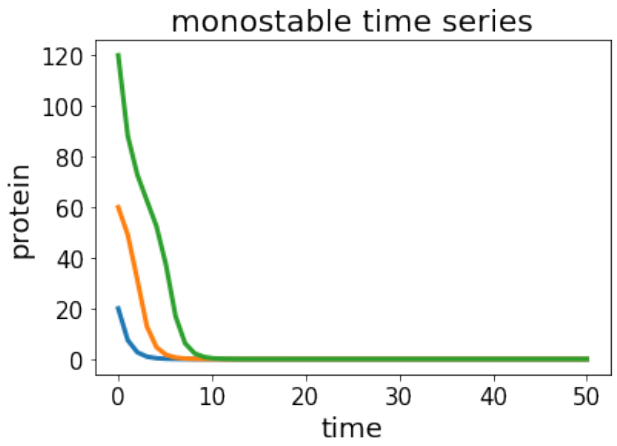
```
        stableFPs_y=[FPs_y[0], FPs_y[2]] # y coordinates of the stable fixed points
        unstableFP_x=FPs_x[1] # x coordinate of the unstable fixed point
        unstableFP_y=FPs_y[1] # y coordinate of the unstable fixed point
        plt.plot(stableFPs_x, stableFPs_y, linestyle='', marker='o', color='black', markersize=10)
        plt.plot(unstableFP_x, unstableFP_y, linestyle='', marker='o', color='black',
            markersize=10, mfc='none')
Out []: [ 0.         50.12531328 91.79197995 92.10526316]
        [ 0.         50.12531328 91.79197995]
        [<matplotlib.lines.Line2D at 0x7f863520b050>]
```

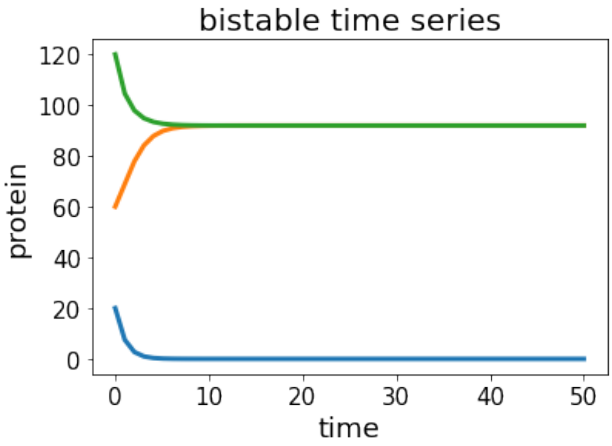



We can use Python to numerically solve differential equations, like the logistic growth equation or the protein production equation defined above. We will use a function from the **scipy** package called 'odeint'.

```
In []:  ## define the protein production equation dx/dt=f(x)-alpha*x discussed above

        def proteinproduction(x,t,pars):
          """
          Returns the instantaneous rate of change given the current time t,
          state variable x, and parameters (pars)
          """
          beta = pars['beta']
          K = pars['K']
          n = pars['n']
          alpha = pars['alpha']

          dxdt = activator(x, beta, K, n) - alpha*x
          return dxdt
Out []:

In []:  ## solve the protein production equation numerically for two different values of
        ## beta, showcasing monostability versus bistability.

        import numpy as np
        import matplotlib.pyplot as plt
        from scipy import integrate
```

```
        # set parameter values
        pars = {}
        pars['alpha'] = 1   #1/hr
        pars['n'] = 4       # dimensionless
        pars['K'] = 50      # nM

        # Numerical solutions of the protein production equation
        t0 = 0 # Initial time
        tf = 50 # Final time
        T = np.linspace(t0,tf) # time steps to report

        x0 = [20,60,120] # Initial protein concentrations

        ####   CHANGE TO 100 TO SEE CASE 2 (BISTABLE) #####
        pars['beta'] = 75 # original value is 75
        ######################################################

        for j in range(len(x0)): # loop through each initial condition x0
          vNint = integrate.odeint(proteinproduction,x0[j],T,args=(pars,)) # this line solves the diffe
          # Plot results for monostable case
          plt.title('monostable time series',fontsize=20)
          if pars['beta']==100:
            plt.title('bistable time series',fontsize=20) # change the title for bistable case
          plt.plot(T,vNint,linewidth=3) # Plot numerically integrated solution
          plt.xlabel('time',fontsize=18)
          plt.ylabel('protein',fontsize=18)
        plt.xticks(fontsize=15)
        plt.yticks(fontsize=15)
Out []: (array([-20.,   0.,  20.,  40.,  60.,  80., 100., 120., 140.]),
        <a list of 9 Text major ticklabel objects>)
```

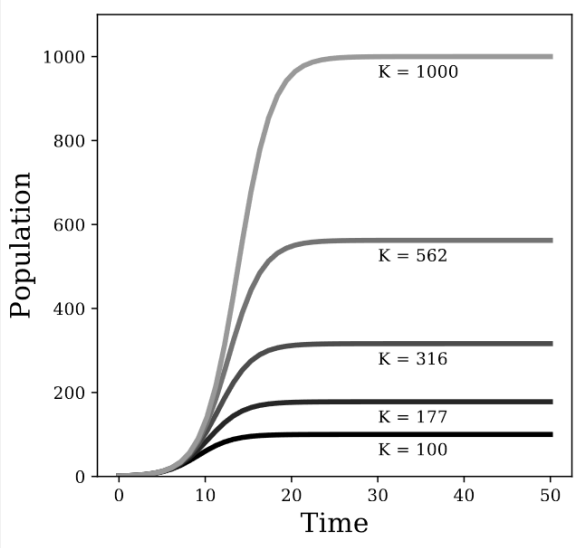





Google scipy odeint for more information.

**Challenge problem: Variables and Differential Equations**

Read the 'odeint' documentation and modify your logistic growth model to accept $r$ and $K$ as parameters. Vary the value of $K$ over 1 order of magnitude and compare the results. If your code works, it should look something like this:

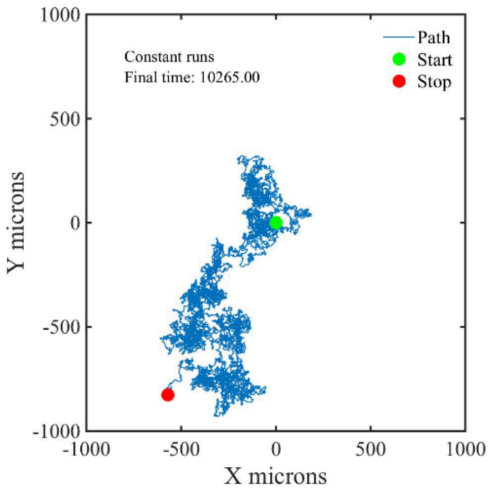


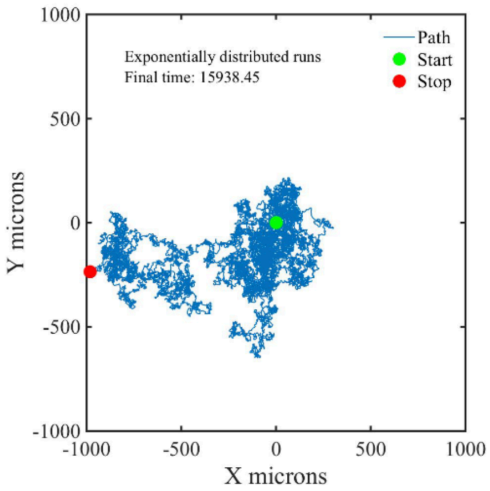

# 5 Advanced - Individual-Based Simulations

For those more experienced in Python, try and develop a simulation of "diffusion" in which a bacteria swims at a speed of 10 $\mu$m/s with each "run" lasting 1 second. In this case the direction of the next run is random. How long will it take, on average, for the bacteria to travel 1mm away from its source? According to the rules of diffusive motion, the average time to travel a distance $x$ should be:

$$T = \frac{x^2}{D} \tag{1}$$

What do we mean by *average* here? If you develop code, can you visualize the results? If you change the length in equation, how does $T$ change? Can you confirm the scaling and estimate $D$? What happens if the durations of each run is exponentially distributed, so that runs last on average 1 second, but can vary in duration? Did the answer with respect to travel time to reach 1mm change in a quantitative or qualitative way? We get much further into these ideas in the Foundations of Quantitative Biosciences course. Here are examples of two runs, one with constant duration and one with exponentially distributed durations:



# 6 Solutions to Challenge Problems

This section includes solutions... try to figure this out on your own before you look! Really! You can do it!

**Solution to Challenge Problem: Exercise on Matrices.**

```
In []:  #Define a random matrix A of size 3-by-3
         A = np.random.rand((3,3))
         #Initialize an empty matrix B
         B = np.zeros((3,3))
         #Use a double "for loop"
         for i in np.arange(3):
             for j in np.arange(3):
                 #calculate the square of the entries in A
                 sq = A[i][j]**2
                 #and store the values in another matrix B
                 B[i][j]=sq
```

**Solution to Challenge Problem: Recursive Factorials.** The solution to this recursive problem involves thinking about what happens in a sequence. For example, if 3! = 321 that is equivalent to 3! = 32! = 3 2 1! = 3 2 1. In essence, by breaking the big problem into little problems of the same kind it is possible to get to the point where the answer is known and then reconstruct the solution – in the formula above it would be moving from left to right and back again. Here is one such solution. Note that one would have to add error correction to make sure there are not issues if the user calls the functions with a real number or a non-positive integer.

```
In []: def factorial_recur(n):
        """
        function N = factorial_recur(n)
        Returns the factorial of n
        """
        if n==1:
           N=1
        else:
           N = n*factorial_recur(n-1)
        return N
```

**Solution to Challenge Problem: Element-wise operations: square root.**

```
In []: import numpy as np
        a = np.arange(1,21,1)
        print(a)
        b = np.sqrt(a)
        print(b)
```

**Solution to Challenge Problem: Finding Elements in Matrices.** Given that `A=np.random.rand(5,5)` generates a $5 \times 5$ matrix, finding indices can be done in a few ways. The most direct way is to use the find command, as follows:

```
In []:  ind = np.argwhere((A>(1/6)) & (A<(1/4)))
```

Note that this answer can be empty if there are no elements that satisfy the condition.

**Solution to Challenge Problem: Variables and Differential Equations.** Although some variables may be fixed in a given system of equations, there are many circumstances where it is necessary to probe the response of a system to variation in such conditions. In this case, it is possible to pass variables to the odeint command, which will, in turn, pass these variables on the function specified to actually compute the rate of change in the system of equations. It's best seen by example. First, here is a modified code that generates a structure of parameters called `pars` and sets up a loop to change the value of $K$:

```
In []: def logGrowth_withpars(N,t,pars):
             """
             function dNdt=logGrowth(N,t,pars)
             logGrowth gives the growth rate of a population of size N at time t
             pars is a dictionary that contains parameters
             """
             r = pars['r']
             K = pars['K']
             dNdt = r*N*(1-N/K)
             return dNdt
```

Next, here is a script that calls this new function multiple times, each with a new value of $K$ and then plots the dynamics on the same axes.

```
In []: import numpy as np
        import matplotlib.pyplot as plt
        from scipy import integrate
        from Lab1_Functions import logGrowth_withpars

        #Parameters
        t0=0 #Initial Time
        tf=50 #Final time
        tVec = np.linspace(t0,tf)
        N0 = 1 #Initial population size
        pars={}
        pars['r']=0.5;
        Krange = np.logspace(2,3,num=5)

        for i in range(len(Krange)):
             pars['K']=Krange[i] #Set the value of K
             vNint = integrate.odeint(logGrowth_withpars,
                         N0,
                         tVec,
                         args=(pars,))
             plt.plot(tVec,vNint,color=np.array([0.75,0.75,0.75])*i/len(Krange),linewidth=3)
             plt.text(30,pars['K']-50,'K = {K}'.format(K=int(pars['K'])),fontsize=10)
plt.ylim([0,1100])
```