Chapter 3 - Derivatives

Section 3.1: The derivative at a point

The derivative of a function $f(x)$ at a point $x = a$ is equal to the rate of change in $f(x)$ at that point, or equivalently the slope of the line tangent to the function at the point $x = a$.

On page 160 of S&D, the derivative of a function $f(x)$ at the point $x = a$ is defined to be equal to $f'(a) = \lim_{h \to 0} \frac{f(a+h)-f(a)}{h}$

- In words, explain why this expression gives the rate of change of the function $f(x)$ at the point $x = a$

Let's use sympy to confirm that the derivative of the function $f(x) = x^2 - 8x + 9$ at the point $x = a$ is $2a - 8$ as shown in Example 4 on page 161 of S&D.

In [1]:
```
import sympy as sp
```

In [3]:
```
f=sp.Function('f')
```

In [4]:
```
a,h,x=sp.symbols('a,h,x')
```

In [4]:
```
def f(x):
    return x**2-8*x+9
```

In [5]:
```
sp.limit((f(a+h)-f(a))/h,h,0)
```

Out[5]:
```
2*a - 8
```

Note in this example, that we defined "f" to be a function data type and then later gave the functional definition for f. Furthermore, before giving the functional definition for f, we also defined "x" to be a symbol data type.

With these definitions, if we typed $f(x)$ into Python, we would get the following result.

In [26]:
```
f(x)
```

Out[26]:
```
x**2 - 8*x + 9
```

Defining $f(x)$ in this way also helped when calculating the limit of the function at the point $x = a$. We could essentially type into Python directly the limit-based definition of the derivative.

An alternative way to evaluate the limit for this function at the point $x = a$ would be

```
In [28]: sp.limit(((a+h)**2-8*(a+h)+9-(a**2-8*a+9))/h,h,0)
```

```
Out[28]: 2*a - 8
```

But encoding the function and limit this way is prone to errors and looks un-necessarily complicated.

Let's now evaluate the derivative at the point $a = 3$, as done in Example 5, p. 161 of S&D. One approach to doing this is to use the substitution operation in sympy (http://docs.sympy.org/latest/tutorial/basic_operations.html (http://docs.sympy.org/latest/tutorial/basic_operations.html)).

```
In [6]: sp.limit((f(a+h)-f(a))/h,h,0).subs(a,3)
```

```
Out[6]: -2
```

Note how we did this, sp.limit((f(a+h)-f(a))/h,h,0) is a operation that returns $2 * a - 8$, and this is further operated on by substituting "3" for a.

An alternative approach is to directly substitute "3" for a in the code sp.limit((f(a+h)-f(a))/h,h,0), such as

```
In [7]: sp.limit((f(3+h)-f(3))/h,h,0)
```

```
Out[7]: -2
```

This is an okay approach, but prone to errors because you need to find each instance of a and then substitute. For the current expression, this is not a problem, but for more complicated expressions, it may be a problem.

Estimating derivatives from empirical data

There are fairly sophisticated methods to estimate the derivative of an empirical function. For example, the data in problem #33 on p. 166-7 of S&D can be thought of as an empirical function. S&D provide a simple approach on pages 162 - 163.

Here, we will work through problem #33,and also plot the empirical function. In doing so, we will extend our practical skills in either R or Python or both.

First, let's plot the empirical function.

R

We can think of the times and blood-alcohol values as vectors of numbers. In R, a vector of numbers $[1, 1.5, 2.0, 2.5, 3.0]$ is entered as

In [11]:
```
c(1,1.5,2,2.5,3)
```

```
1  1.5  2  2.5  3
```

Let's define the vector of numbers $[1, 1.5, 2.0, 2.5, 3.0]$ to be equal to the variable "t" sense these are times

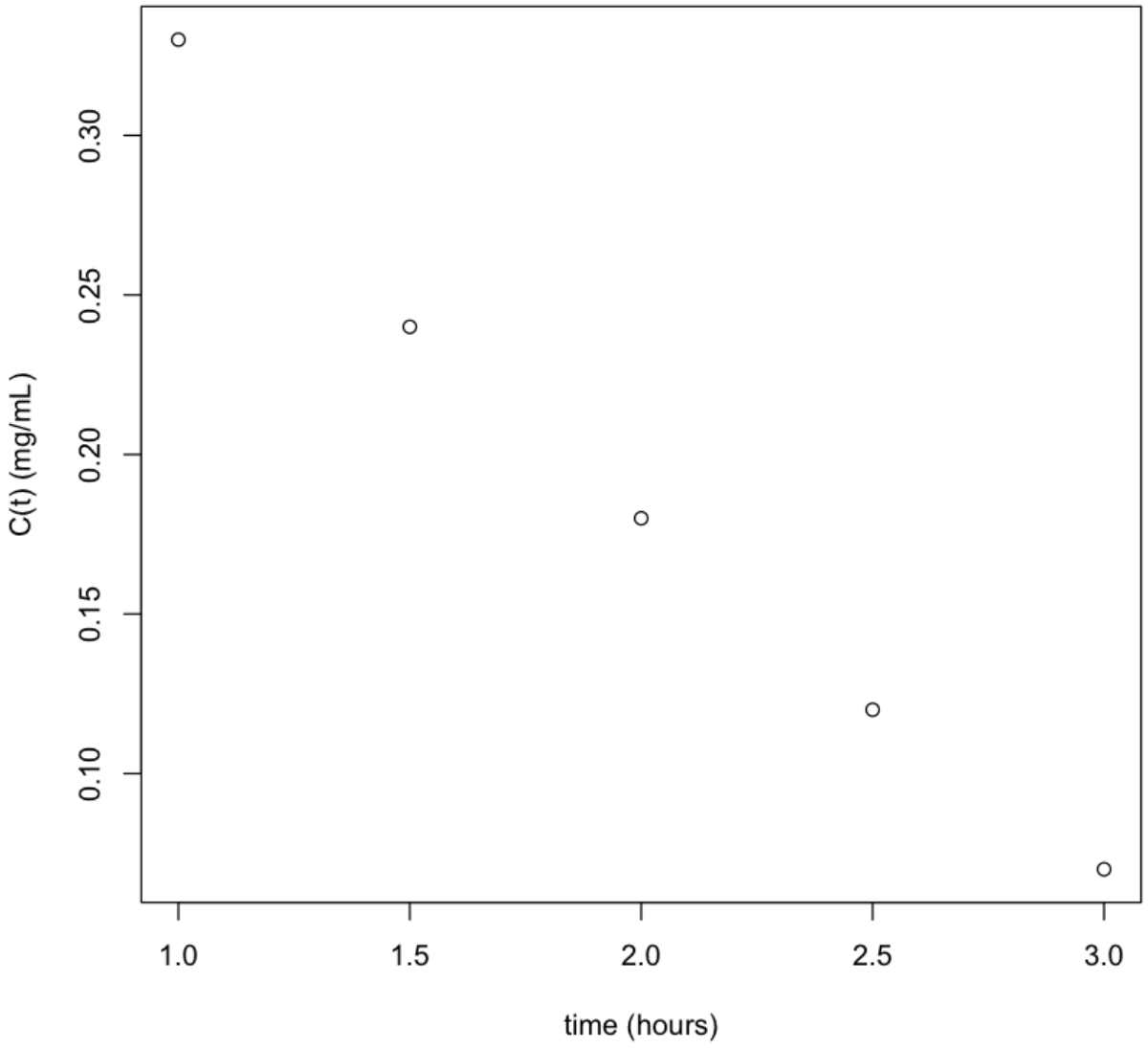
In [12]:
```
t=c(1,1.5,2,2.5,3)
```

Similarly, we will define the vector of blood alcohol concentrations to be equal to the variable "C"

In [13]:
```
C=c(0.33,0.24,0.18,0.12,0.07)
```

Next, we'll plot the empirical function.

In [14]:
```
plot(t,C,xlab='time (hours)',ylab='C(t) (mg/mL)')
```



The average rate of change of an empirical function over a time interval from t_1 to t_2 is

$\frac{f(t_2)-f(t_1)}{t_2-t_1}$ (S&D p. 162)

The interval [1.0,2.0] corresponds to the first and third time points, so the average rate of change is

In [15]:
```
(C[3]-C[1])/(t[3]-t[1])
```

-0.15

The units for the change are $\frac{(mg/mL)}{hr}$, or equivalently $\frac{mg}{mLhr}$.

- Calculate the average rate of change for at least one other time interval and verify

For part b of problem #33, we can use the approach given in example 7 (page 166), whereby an estimate of the derivative is the average of the rate of change to the left of the point $x = 2$ and the rate of change to the right of the point $x = 2$.

In [17]:
```
1/2*((C[3]-C[2])/(t[3]-t[2])+(C[4]-C[3])/(t[4]-t[3]))
```

-0.12

- Why is it more accurate to take the average of the rate of change above and below the point, as opposed to choosing to take as the derivate the rate either above or below the point?

Scientific Python

Vectors of numbers are entered as shown below.

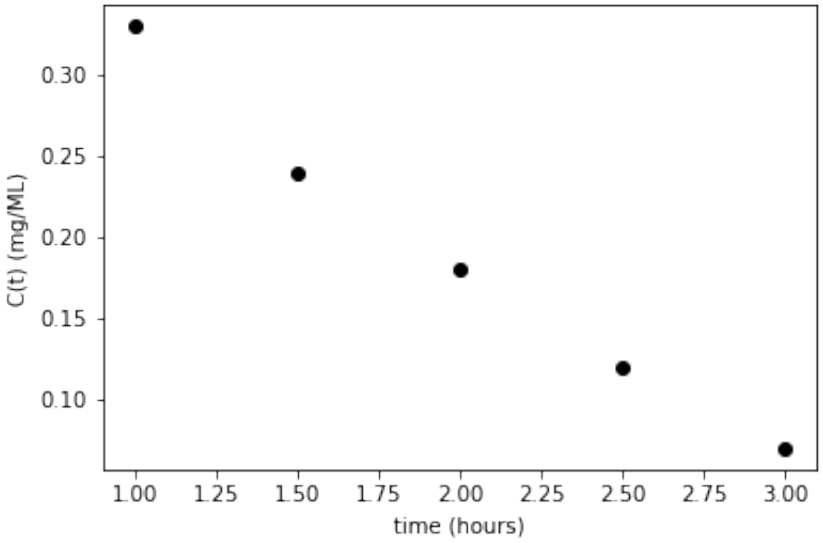
In [14]:
```
t=[1.,1.5,2,2.5,3.]
```

In [15]:
```
C=[0.33,0.24,0.18,0.12,0.07]
```

Plotting is the same as before, except we want to plot data points only and not lines between points. To do so we add the option 'ko' to the plot command to indicate plotting "black" "points" (see the tutorial https://matplotlib.org/users/pyplot_tutorial.html (https://matplotlib.org/users/pyplot_tutorial.html)).

In [3]:
```
import matplotlib.pyplot as plt
```

In [16]:
```
plt.xlabel('time (hours)')
plt.ylabel('C(t) (mg/ML)')
plt.show(plt.plot(t,C,'ko'))
```



Note that indexing of vectors in Python starts at "0", as opposed to "1", so rates are calculated as for part (a-i) and part (b):

In [35]:
```
(C[3-1]-C[1-1])/(t[3-1]-t[1-1])
```

Out[35]:
```
-0.15000000000000002
```

In [36]:
```
1/2*((C[3-1]-C[2-1])/(t[3-1]-t[2-1])+(C[4-1]-C[3-1])/(t[4-1]-t[3-1]))
```

Out[36]:
```
-0.12
```

Section 3.2: Derivatives as functions and higher derivatives

The textbook in Section 3.2 makes the technical connection between the derivative of a function at a point, which is a scalar (number-valued) versus the derivative of a function across an interval, which is function-valued. It then presents the technical basis of higher derivatives. For example, the second derivative of a function is the derivative of the derivative of a function. This is likely a review for you and we will now move toward using evaluating derivatives in R and Scientific Python using the derivative function, as opposed to evaluating it as a limit.

Section 3.3 - 3.4: Basic Differentiation formulas

Using Scientific Python, it is not necessary to memorize or be able to derive various cases for the derivatives of functions. In practice, theoretical biologists let a computer algebra system perform the calculation.

For example, consider taking the derivative of the $\csc(x)$ function. One approach to evaluating the derivative is to recognize that $\csc(x) = \frac{1}{\sin(x)}$, and then use the quotient rule to calculate the derivative. This approach requires remembering that the derivative of $\sin(x)$ is $-\cos(x)$, or being able to derive this equivalence.

Alternatively, we can use the derivative function in sympy to calculate the derivative (http://docs.sympy.org/latest/tutorial/calculus.html (http://docs.sympy.org/latest/tutorial/calculus.html)).

In [8]:
```
sp.diff(sp.csc(x),x)
```

Out[8]:
```
-cot(x)*csc(x)
```

Note that having imported sympy as "sp", the derivative operator is "sp.diff", with the function as the first argument and the variable as the second argument. Note further, that I am assuming that "x" has retained its status as a "symbol" in Scientific Python.

A lovely first-year calculus exam question might be

Evaluate the following expression $\frac{d}{dx}(\frac{d}{dx}\csc(x))$.

I.e., the second derivative of the function $\csc(x)$.

Not a problem:

In [9]:
```
sp.diff(sp.diff(sp.csc(x),x),x)
```

Out[9]:
```
-(-cot(x)**2 - 1)*csc(x) + cot(x)**2*csc(x)
```

Or an alternate encoding of this second derivative is

In [10]:
```
sp.diff(sp.csc(x),x,x)
```

Out[10]:
```
(2*cot(x)**2 + 1)*csc(x)
```

Note we seemlingly get different answers, but let's simplify the output of sp.diff(sp.diff(sp.csc(x),x),x), (http://docs.sympy.org/latest/tutorial/simplification.html (http://docs.sympy.org/latest/tutorial/simplification.html)).

In [13]:
```
sp.simplify(sp.diff(sp.diff(sp.csc(x),x),x))
```

Out[13]:
```
(2*cot(x)**2 + 1)*csc(x)
```

If we had directly pasted the output -(-cot(x)*2 - 1)\csc(x) + cot(x)*2\csc(x) into the simplify function, we would get an error because functions cot and csc need to have an "sp." before them.

In [14]:
```
sp.simplify(-(-cot(x)**2 - 1)*csc(x) + cot(x)**2*csc(x))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call
last)
<ipython-input-14-13dea8103983> in <module>()
----> 1 sp.simplify(-(-cot(x)**2 - 1)*csc(x) + cot(x)**2*csc(x))

NameError: name 'cot' is not defined
```

In [15]:
```
sp.simplify(-(-sp.cot(x)**2 - 1)*sp.csc(x) + sp.cot(x)**2*sp.csc(x))
```

Out[15]:
```
(2*cot(x)**2 + 1)*csc(x)
```

What is an interpretation of the second derivative of a function?

On your own, try problem #55 on p. 201 of S&D.

Section 3.5: The chain rule

Although sympy can take care of the mechanics of calculating derivatives, it's important to have a good conceptual understanding of what a derivative is, its various interpretations and how it can be used to help us understand biology. The chain rul provides a good example of how a principle from mathematics can be used to help us think about an analyze biological problems.

For example, in ecology, consider the growth rate of a population at time t, $r(t)$. This growth rate can be a function of many things, one of which is the average body size of individuals in the population at time t, $\bar{z}(t)$. Such that the growth rate is a composite function $r(t) = f(\bar{z}(t))$.

Ecologist are often interested in factors that determine how $r(t)$ changes with time, or the derivative of $r(t)$ with respect to time $\frac{d}{dt}r(t) = \frac{d}{dt}f(\bar{z}(t))$. Using the chain rule, this derivative evaluates to

$$\frac{d}{dt}f(\bar{z}(t)) = \frac{df(\bar{z}(t))}{d\bar{z}(t)}\frac{d\bar{z}(t)}{dt}$$

Conceptually and empirically, this is very useful: The formula tells use that the change in growth rate is a product of the change in growth rate due to changes in mean body size times the change in mean body size with respect to time.

Ecologists can directly measure the first term in this product, i.e. they can regress growth rate on mean body size. The second term suggests that to understand how population growth changes through time requires consideration of how mean body size changes through time, which is a

function of both plastic responses to the environment and evolutionary changes in mean breeding values.

In fact, given a model for $\bar{z}(t)$, the chain rule can further tease apart the various factors that affect changes in this quantity.

The chain rule forms the basis of the classic paper by Hairston et al. (2005).

On your own, using sympy verify the formula given by equation 5 on p. 207 of S&D.

In [16]:
```
b=sp.symbols('b')
```

In [18]:
```
sp.diff(b**x,x)
```

Out[18]:
```
b**x*log(b)
```

#56, p. 213

Lets use this program to see how we can use sympy both abstractly and concretely in the context of a problem involving the chain rule.

For this problem, the function $A(x)$ can be entered using sympy as

In [46]:
```
f=sp.Function('f')
g=sp.Function('g')
h=sp.Function('h')
A=sp.Function('A')

def A(x):
    return f(g(h(x)))
```

In [49]:
```
A(x)
```

Out[49]:
```
f(g(h(x)))
```

The problem gives various values of the functions and their derivatives and wants us to evaluate $A'(x)$ at the point $x = 500$. How can we combine a conceptual understanding of the derivative and the chain rule with specific numerical instances?

First, let's try to take the derivative of $A(x)$ using sympy.

In [53]:
```
sp.diff(A(x),x)
```

Out[53]:
```
Derivative(h(x), x)*Subs(Derivative(f(_xi_1), _xi_1), (_xi_1,), (g(h(x
)),))*Subs(Derivative(g(_xi_1), _xi_1), (_xi_1,), (h(x),))
```

The result involves some complicated looking stuff involving a new operator "Subs", with a capital "S" and "Derivative" with a capital "D". These are unevaluated versions of substitution (subs) and differentiation (diff).

(subs) and differentiation (diff).

To have Python evaluate these expressions, we can use the operator "doit()".

In [54]:
```
sp.diff(A(x),x).doit()
```

Out[54]:
```
Derivative(f(g(h(x))), g(h(x)))*Derivative(g(h(x)), h(x))*Derivative(h
(x), x)
```

The result is simpler, with the substitutions having been made. The Derivative is still left because given that we have not specified specific equations for f, g, and h, their derivatives cannot be simplified further.

Nevertheless, the resulting expression is useful and is equal to

$$\frac{df(g(h(x)))}{dg(h(x))}\frac{dg(h(x))}{dh(x)}\frac{dh(x)}{dx}$$

or equivalently

$$f'(g(h(x)))g'(h(x))h'(x)$$

Using the numerical instances:

$$h(500) = 8$$

$$g(h(500)) = g(8) = 2$$

$$f'(g(h(500))) = f'(g(8)) = f'(2) = 1$$

$$g'(h(500)) = g'(8) = 1/4$$

$$h'(500) = 2.5 = 5/2$$

Together,

$$f'(g(h(x)))g'(h(x))h'(x) = 1 \times 1/4 \times 5/2 = 5/8$$

The result indicates that the instantaneous rate of increase of antibiotic reaching the sinus cavity at a 500mg dose of antibiotic is 5/8 mg per mg dose orally.

Note that there is actually an error in the value of the derivative given for $h'(500)$. Based on problem 1.3.7, the value should be about 2.5×10^{-4}, such that the instantaneous rate of increase is $1 \times 1/4 \times 2.5 \times 10^{-4} = 0.0000625$.

Section 3.8: Approximations to functions and Newton's method

The derivative forms the basis of two fundamental approaches in theoretical biology - approximations to functions and Newton's method for solving roots of equations.

For example, often we seek to calculate the equilibrium of something - be it population size, allele frequency or the concentration of a hormone. An equilibrium occurs when $x_{t+1} = x_t$. If we let $\widetilde{x}$ be the equilibrium point and $x_{t+1} = f(x_t)$, then an equilibrium is the root of the equation $\widetilde{x} = f(\widetilde{x})$.

Furthermore, we often want to know if an equilibrium is stable or not. To test this, we often use perturbation techniques that assume small changes in a variable. These small changes can be accurately approximated with linear or quadratic approximations, which can often simplify analysis and the interpretation of results.

Solving for roots

The sympy library in Scientific Python has functions for finding roots of equations. The default approach used by sympy is the "secant method", which is an extension of Newton's method. It is also possible to force sympy to use Newton's method.

Let's solve for the roots of the equation in Example 5 on p. 234 of S&D using sympy. To do this we define the function for which we seek to find the roots for.

In [2]:
```
x=sp.symbols('x')
```

In [188]:
```
def f(x):
    return x**3-2*x-5
```

In [4]:
```
f(x)
```

Out[4]:
```
x**3 - 2*x - 5
```

Next we use the root solver within the "mpmath" sublibrary of sympy to numerically solve for a root (http://docs.sympy.org/0.7.6/modules/mpmath/calculus/optimization.html (http://docs.sympy.org/0.7.6/modules/mpmath/calculus/optimization.html)).

In [9]:
```
sp.mpmath.findroot(f,2)
```

Out[9]:
```
mpf('2.0945514815423266')
```

The answer above agrees with the example from the textbook.

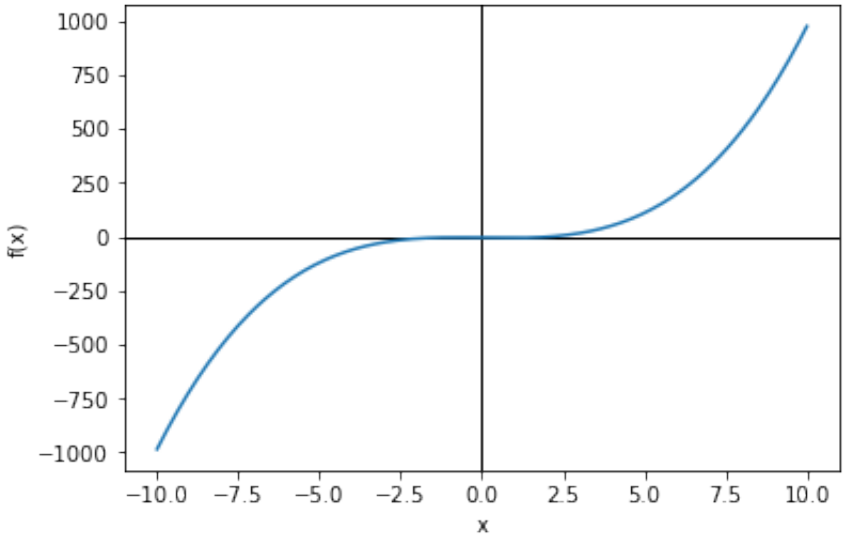
The basic information that is required in the "findroot" function is the name of the function and an initial starting point for finding a root; in this case we chose "2" as the initial starting point. Note the the initial starting point can sometimes be critical for accurately finding a root and/or the particular root of interest when there are multiple roots to an equation.

Are there other roots to this equation? Let's plot the equation to check.

In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [50]:
```
x = np.linspace(-10,10, 256, endpoint=True)
y=f(x)

plt.xlabel('x')
plt.ylabel('f(x)')
plt.axhline(linewidth=1,color = 'black')
plt.axvline(linewidth=1,color = 'black')
plt.show(plt.plot(x,y))
```

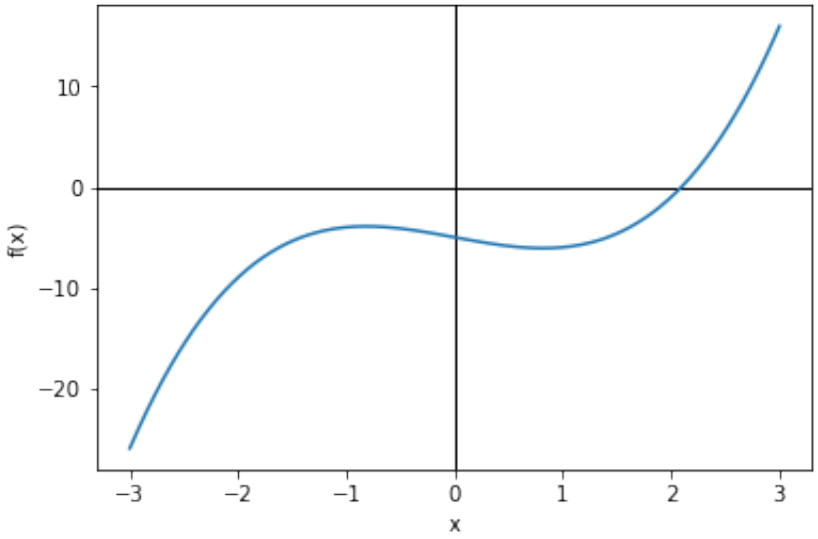


Note that x and y axes were added to the plot at the origin to help us see when the function may be equal to $y = 0$. x and y axes were added using the commands plt.axhline(linewidth=1,color = 'black') and plt.axvline(linewidth=1,color = 'black').

The plot above suggests that if there is another root it is in the interval $-3 < x < 3$, so we will focus on this interval in the plot.

In [51]:
```
x = np.linspace(-3,3, 256, endpoint=True)
y=f(x)

plt.xlabel('x')
plt.ylabel('f(x)')
plt.axhline(linewidth=1,color = 'black')
plt.axvline(linewidth=1,color = 'black')
plt.show(plt.plot(x,y))
```



The function does not cross $y = 0$ at any point other than $x = 2.0946$, so this is the only root.

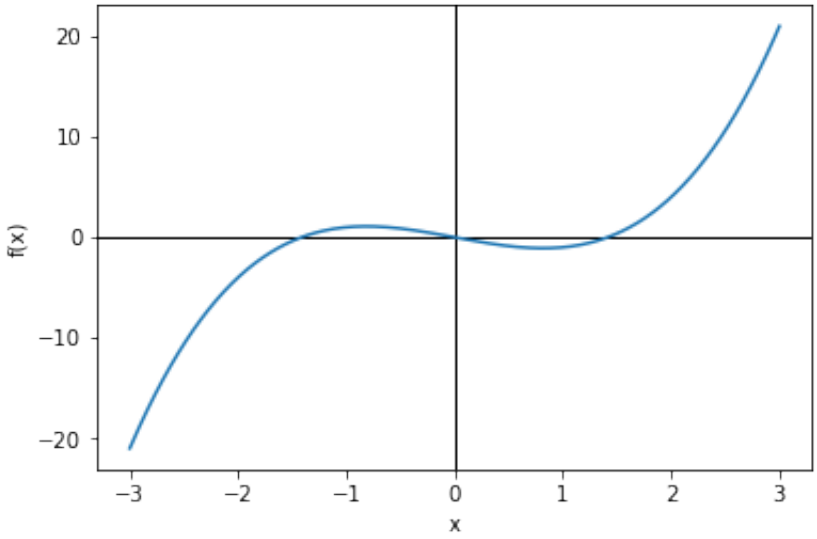
Let's consider modifying the function to be $f(x) = x^3 - 2x$ and find the roots for this equation.

Plotting the equation indicates that there are three roots:

In [68]:
```
def f(x):
    return x**3-2*x
```

In [71]:
```
x = np.linspace(-3,3, 256, endpoint=True)
y=f(x)

plt.xlabel('x')
plt.ylabel('f(x)')
plt.axhline(linewidth=1,color = 'black')
plt.axvline(linewidth=1,color = 'black')
plt.show(plt.plot(x,y))
```



Using the "findroot" function and different starting points reveals the three roots:

In [72]:
```
sp.mpmath.findroot(f,2)
```

Out[72]:
```
mpf('1.414213562373095')
```

In [73]:
```
sp.mpmath.findroot(f,1)
```

Out[73]:
```
mpf('1.414213562373095')
```

In [74]:
```
sp.mpmath.findroot(f,0.1)
```

Out[74]:
```
mpf('0.0')
```

In [75]:
```
sp.mpmath.findroot(f,-0.1)
```

Out[75]:
```
mpf('0.0')
```

In [76]:
```
sp.mpmath.findroot(f,-1)
```

Out[76]:
```
mpf('-1.414213562373095')
```

Approximations to functions

Taylor's polynomials (series)

A very commonly used technique in theoretical biology is to approximate a function using a Taylor series. In Scientific Python the Taylor series approximation can be implemented in the following way (http://docs.sympy.org/0.7.6/modules/mpmath/calculus/approximation.html (http://docs.sympy.org/0.7.6/modules/mpmath/calculus/approximation.html)). BUT, I have experienced some problems using the internal Taylor's function and therefore define a function to perform the Taylor's series approximation manually.

Example 8, p. 236 - 237 S&D

First, we define the base function that we want to approximate.

In [3]:
```
f=sp.Function('f')
x=sp.symbols('x')
```

In [4]:
```
def f(x):
    return sp.ln(x)
```

Next, we define a general function that will give the Taylor's series approximation to a base function.

We will define the function using the definition for the Taylor's series. As function arguments, we will include the function we seek the Taylor's series for, the point at which we are centering the approximation (x0) and the degree of the approximation.

The function definition uses the "factorial()" function from the mpmath library. Furthermore, it makes use of the notation "sp.diff(function,x, i)", which takes the ith derivative of the function with respect to x. Note that the "0th" derivative is defined to be the original function. After symbolically evaluating the derivative, "x0" is substituted for "x" using the subs() function. The "while" statement adds to the apprx variable the order of the Taylor's approximation from zero up to the "degree" specified.

In [5]:
```
def taylor(function, x0, degree):
    i = 0
    apprx = 0
    while i <= degree:
        apprx = apprx + (sp.diff(function,x, i).subs(x, x0))/(sp.mpmath.f
        i = i + 1
    return apprx
```

Using the general formula to calculate the Taylor's series approximation to a function, we then define a function to equal this approximation in the following way.

In [6]:
```
f_taylor=sp.Function('f_taylor')
x0,degree=sp.symbols('x0 degree')
```

In [16]:
```
def f_taylor(x,x0,degree):
    return taylor(f(x),x0,degree)
```

f_taylor simplify returns the taylor approximation to the function $f(x)$, specifying the centering point "x0" and the "degree".

Let's see what this function outputs. We'll try a third-degree Taylor series approximation about the point $x = 1$.

In [11]:
```
f_taylor(x,1,3)
```

Out[11]:
```
1.0*x + 0.333333333333333*(x - 1)**3 - 0.5*(x - 1)**2 - 1.0
```

Next, let's plot the base function and its Taylor's approximations as in figure 12 of S&D, p. 237.

To do so requires us to translate between a sympy function, in this case sp.ln(x) and f_taylor() and numpy. This is done using the operator lambdify in sympy (http://docs.sympy.org/latest/modules/utilities/lambdify.html (http://docs.sympy.org/latest/modules/utilities/lambdify.html)). We need to lambdify the f_taylor() function because it uses sympy functions internally, i.e. sp.mpmath.factorial().

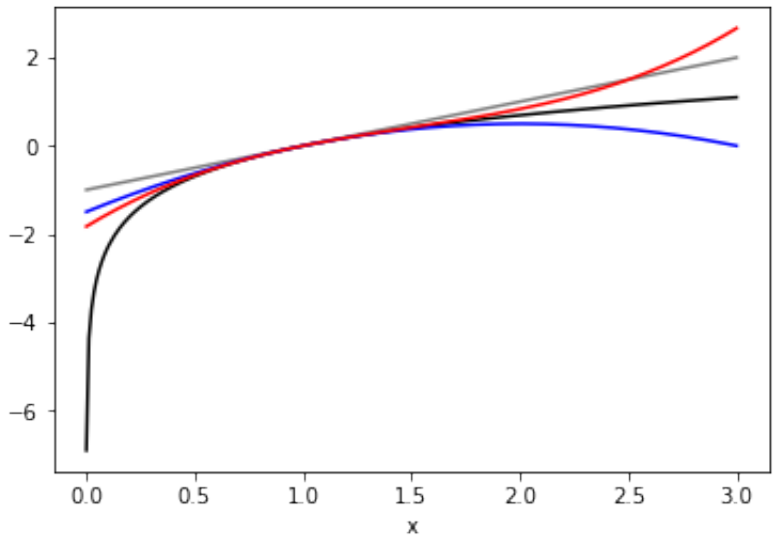
Note further in the code below that we define a lambdified function of f_taylor() for each degree of the Taylor's approximation because the stopping point of the "while" loop within the f_taylor() function needs to be defined for the function to be lambdified. Note that by setting "degree", f_taylor() becomes a function of two variables x and $x0$ and this is indicated with the first argument "(x,x0)" in the sp.lambdify function.

In [26]:
```
x_val = np.linspace(0.001,3, 256, endpoint=True)
y=sp.lambdify(x,f(x),"numpy")
y_val=y(x_val)

f1_taylor=sp.lambdify((x,x0),f_taylor(x,x0,1),"numpy")
f2_taylor=sp.lambdify((x,x0),f_taylor(x,x0,2),"numpy")
f3_taylor=sp.lambdify((x,x0),f_taylor(x,x0,3),"numpy")

f1_val=f1_taylor(x_val,1)
f2_val=f2_taylor(x_val,1)
f3_val=f3_taylor(x_val,1)

plt.xlabel('x')
plt.plot(x_val,y_val,color='black')
plt.plot(x_val,f1_val,color='gray')
plt.plot(x_val,f2_val,color='blue')
plt.plot(x_val,f3_val,color='red')
plt.show()
```



Note further that we defined each plot in a separate row and then invoked plt.show() as the last line to show all of the define plots.

To better understand why we needed to "lambdify" the function f(x), let's try directly evaluating y=f(x_val), which seemed the logical thing to due given our previous experience plotting in Scientific Python.

In [30]:
```
y=f(x_val)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-30-8d174a004a53> in <module>()
----> 1 y=f(x_val)

<ipython-input-3-9a0b189df97c> in f(x)
      1 def f(x):
----> 2     return sp.ln(x)

/Users/cort/anaconda/lib/python3.6/site-packages/sympy/core/function.py in __new__(cls, *args, **options)
    383
    384         evaluate = options.get('evaluate', global_evaluate[0])
--> 385         result = super(Function, cls).__new__(cls, *args, **options)
    386         if not evaluate or not isinstance(result, cls):
    387             return result

/Users/cort/anaconda/lib/python3.6/site-packages/sympy/core/function.py in __new__(cls, *args, **options)
    197         from sympy.sets.sets import FiniteSet
    198
--> 199         args = list(map(sympify, args))
    200         evaluate = options.pop('evaluate', global_evaluate[0])
    201         # WildFunction (and anything else like it) may have nargs defined

/Users/cort/anaconda/lib/python3.6/site-packages/sympy/core/sympify.py in sympify(a, locals, convert_xor, strict, rational, evaluate)
    280         try:
    281             return type(a)([sympify(x, locals=locals, convert_xor=convert_xor,
--> 282                             rational=rational) for x in a])
    283         except TypeError:
    284             # Not all iterables are rebuildable with their type.

ValueError: sequence too large; cannot be greater than 32
```

Note we get an error. The error output is not immediately informative, but indicates that there is a problem applying a numpy array - in our case x_val - within a sympy function - in our case sp.ln(x).

To evaluate a sympy function across an array of numpy values, requires us to "lambdify" the sympy function:

In [33]:
```
y=sp.lambdify(x,f(x),"numpy")
```

```
y_val=y(x_val)
y_val
```

Out[33]:
```
array([-6.90775528, -6.13185414, -5.70025928, -5.39976398, -5.16901057
,
       -4.98164776, -4.82391168, -4.68769964, -4.56783662, -4.46081523
,
       -4.3641482 , -4.27600751, -4.19501023, -4.12008478, -4.05038397
,
       -3.98522641, -3.9240559 , -3.86641249, -3.81191154, -3.76022804
,
       -3.71108497, -3.66424423, -3.61949973, -3.57667185, -3.53560311
,
       -3.49615468, -3.45820356, -3.42164021, -3.38636673, -3.35229518
,
       -3.31934636, -3.28744862, -3.25653697, -3.22655227, -3.19744054
,
       -3.16915239, -3.14164249, -3.11486917, -3.088794  , -3.06338151
,
       -3.03859884, -3.01441553, -2.99080326, -2.9677357 , -2.94518828
,
       -2.92313805, -2.90156356, -2.88044472, -2.85976268, -2.83949973
,
       -2.81963922, -2.80016549, -2.78106375, -2.76232006, -2.74392124
,
       -2.72585483, -2.70810903, -2.69067266, -2.67353511, -2.65668632
,
       -2.64011671, -2.62381719, -2.60777909, -2.59199415, -2.5764545
,
       -2.56115265, -2.54608142, -2.53123397, -2.51660374, -2.50218447
,
       -2.48797016, -2.47395507, -2.4601337 , -2.44650075, -2.43305116
,
       -2.41978007, -2.40668279, -2.39375484, -2.38099188, -2.36838977
,
       -2.3559445 , -2.34365222, -2.33150919, -2.31951186, -2.30765676
,
       -2.29594055, -2.28436003, -2.27291208, -2.2615937 , -2.250402
,
       -2.23933417, -2.22838749, -2.21755935, -2.2068472 , -2.19624858
,
       -2.18576112, -2.17538251, -2.1651105 , -2.15494294, -2.14487772
,
       -2.1349128 , -2.1250462 , -2.11527599, -2.10560033, -2.09601738
,
       -2.0865254 , -2.07712266, -2.06780752, -2.05857835, -2.04943357
,
       -2.04037167, -2.03139115, -2.02249056, -2.01366849, -2.00492357
,
       -1.99625446, -1.98765986, -1.9791385 , -1.97068914, -1.96231057
,
       -1.95400162, -1.94576114, -1.93758801, -1.92948114, -1.92143946
,
       -1.91346194, -1.90554755, -1.89769531, -1.88990425, -1.88217342
```

,
       -1.8745019 , -1.86688878, -1.85933318, -1.85183424, -1.84439112
,
       -1.83700299, -1.82966904, -1.82238849, -1.81516056, -1.8079845
,
       -1.80085957, -1.79378505, -1.78676022, -1.7797844 , -1.7728569
,
       -1.76597706, -1.75914423, -1.75235778, -1.74561706, -1.73892149
,
       -1.73227044, -1.72566334, -1.7190996 , -1.71257867, -1.70609999
,
       -1.699663 , -1.69326719, -1.68691203, -1.68059699, -1.67432159
,
       -1.66808532, -1.66188771, -1.65572826, -1.64960653, -1.64352204
,
       -1.63747434, -1.63146301, -1.62548759, -1.61954767, -1.61364282
,
       -1.60777263, -1.60193671, -1.59613464, -1.59036604, -1.58463053
,
       -1.57892773, -1.57325726, -1.56761877, -1.56201189, -1.55643627
,
       -1.55089157, -1.54537745, -1.53989356, -1.53443958, -1.52901519
,
       -1.52362006, -1.51825388, -1.51291635, -1.50760715, -1.50232599
,
       -1.49707258, -1.49184662, -1.48664783, -1.48147592, -1.47633063
,
       -1.47121168, -1.46611879, -1.46105171, -1.45601018, -1.45099394
,
       -1.44600274, -1.44103632, -1.43609445, -1.43117688, -1.42628337
,
       -1.42141369, -1.41656761, -1.41174491, -1.40694535, -1.40216871
,
       -1.39741479, -1.39268335, -1.3879742 , -1.38328712, -1.37862191
,
       -1.37397836, -1.36935627, -1.36475545, -1.3601757 , -1.35561682
,
       -1.35107864, -1.34656096, -1.34206359, -1.33758637, -1.33312909
,
       -1.3286916 , -1.32427371, -1.31987526, -1.31549606, -1.31113596
,
       -1.30679479, -1.30247238, -1.29816857, -1.29388321, -1.28961613
,
       -1.28536719, -1.28113622, -1.27692308, -1.27272761, -1.26854967
,
       -1.26438911, -1.2602458 , -1.25611958, -1.25201031, -1.24791786
,
       -1.24384209, -1.23978287, -1.23574006, -1.23171352, -1.22770313
,
       -1.22370877, -1.21973029, -1.21576758, -1.21182051, -1.20788896
,
       -1.2039728])

Likewise, the reason we need to set the value of "degree" when lambdifying the f_taylor()

Likewise, the reason we need to set the value of "degree" when lambdifying the f_taylor() function can be seen by trying lambdify without setting the value for "degree".

In [27]:
```
f1_taylor=sp.lambdify((x,x0,degree),f_taylor(x,x0,degree),"numpy")
```

```
---------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
<ipython-input-27-20caa7f9f50a> in <module>()
----> 1 f1_taylor=sp.lambdify((x,x0,degree),f_taylor(x,x0,degree),"num
py")

<ipython-input-16-618acf789ebe> in f_taylor(x, x0, degree)
      1 def f_taylor(x,x0,degree):
----> 2     return taylor(f(x),x0,degree)

<ipython-input-5-8a4c7913e19b> in taylor(function, x0, degree)
      2     i = 0
      3     apprx = 0
----> 4     while i <= degree:
      5         apprx = apprx + (sp.diff(function,x, i).subs(x, x0))/(
sp.mpmath.factorial(i))*(x - x0)**i
      6         i = i + 1

/Users/cort/anaconda/lib/python3.6/site-packages/sympy/core/relational
.py in __nonzero__(self)
    193
    194     def __nonzero__(self):
--> 195         raise TypeError("cannot determine truth value of Relat
ional")
    196
    197     __bool__ = __nonzero__

TypeError: cannot determine truth value of Relational
```

Note we get an error that it cannot determine whether the "while" statement is satisfied.

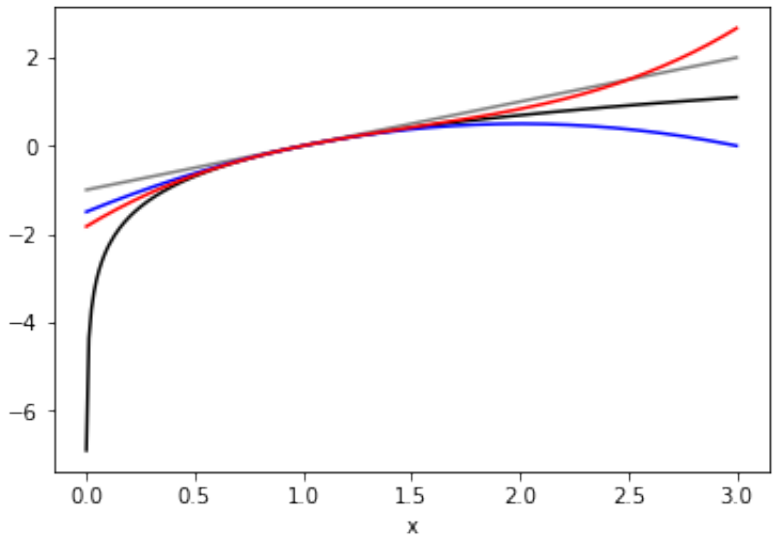
Let's go back and inspect the plot of the base function and its approximation.

In [28]:
```
x_val = np.linspace(0.001,3, 256, endpoint=True)
y=sp.lambdify(x,f(x),"numpy")
y_val=y(x_val)

f1_taylor=sp.lambdify((x,x0),f_taylor(x,x0,1),"numpy")
f2_taylor=sp.lambdify((x,x0),f_taylor(x,x0,2),"numpy")
f3_taylor=sp.lambdify((x,x0),f_taylor(x,x0,3),"numpy")

f1_val=f1_taylor(x_val,1)
f2_val=f2_taylor(x_val,1)
f3_val=f3_taylor(x_val,1)

plt.xlabel('x')
plt.plot(x_val,y_val,color='black')
plt.plot(x_val,f1_val,color='gray')
plt.plot(x_val,f2_val,color='blue')
plt.plot(x_val,f3_val,color='red')
plt.show()
```



Note that near the point $x = 1$, the base function is well approximated, even by the linear (first-degree) Taylor's approximation.

The equation for the linear (first-degree) Taylor's approximation is

In [29]:
```
f_taylor(x,1,1)
```

Out[29]:
```
1.0*x - 1.0
```

which is a simple function and more straightforward to interpret and work with than ln(x).

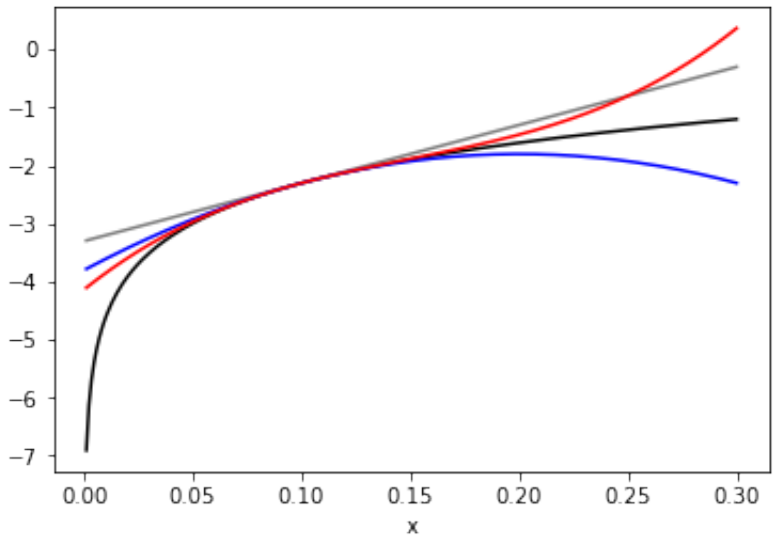
We can also approximate the function around a different centering value, for example x=0.1:

In [31]:
```
x_val = np.linspace(0.001,0.3, 256, endpoint=True)
y=sp.lambdify(x,f(x),"numpy")
y_val=y(x_val)

f1_taylor=sp.lambdify((x,x0),f_taylor(x,x0,1),"numpy")
f2_taylor=sp.lambdify((x,x0),f_taylor(x,x0,2),"numpy")
f3_taylor=sp.lambdify((x,x0),f_taylor(x,x0,3),"numpy")

f1_val=f1_taylor(x_val,0.1)
f2_val=f2_taylor(x_val,0.1)
f3_val=f3_taylor(x_val,0.1)

plt.xlabel('x')
plt.plot(x_val,y_val,color='black')
plt.plot(x_val,f1_val,color='gray')
plt.plot(x_val,f2_val,color='blue')
plt.plot(x_val,f3_val,color='red')
plt.show()
```



In [ ]: