Numerics 1: HW 1

Cooper Simpson

September 3, 2020

Setup

```
[1]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     from IPython.display import display
```

Problem 1

We consider the following calculations and how to avoid cancellation due to numerical imprecision.

a).

$$\sqrt{x+1}-1,\ x\approx 0$$

The problem here is that for small magnitude x we will be subtracting 1 from something very close to 1. This follows not only analytically but also because we can only approximate the square root up to a certain precision. We can multiply by the conjugate to try and eliminate this problem.

$$\sqrt{x+1}-1\cdot\frac{\sqrt{x+1}+1}{\sqrt{x+1}+1}=\frac{x}{\sqrt{x+1}+1}$$

Using the expression $\boxed{\dfrac{x}{\sqrt{x+1}+1}}$ we can eliminate numerical imprecision for $x\approx 0$ as we are no longer using any subtraction.

b).

$$\sin(x)-\sin(y),\ x\approx y$$

In the above expression the two sine values will be almost identical. We can use a trig sum identity to try and eliminate this subtraction.

$$\sin(x)-\sin(y)=2\cos(\frac{x+y}{2})\sin(\frac{x-y}{2})$$

Using $\boxed{2\cos(\frac{x+y}{2})\sin(\frac{x-y}{2})}$ to compute the desired expression will be much more numerically stable. We can note that we have not eliminated all subtraction as an $(x-y)$ term remains. However, given that the numbers representable by a machine are only a finite subset of the real numbers, we reduce error by doing the subtraction before approximating the sin.

c).

$$\frac{1-\cos(x)}{\sin(x)},\ x \approx 0$$

The cancellation problem in the above expression comes from the $1-\cos(x)$ term where $\cos(x) \approx 1$ for small magnitude x. We can solve this problem by multiplying by the conjugate.

$$\frac{1-\cos(x)}{\sin(x)} \cdot \frac{1+\cos(x)}{1+\cos(x)} = \frac{1-\cos^2(x)}{\sin(x)(1+\cos(x))} = \frac{\sin(x)}{1+\cos(x)}$$

Using $\boxed{\frac{\sin(x)}{1+\cos(x)}}$ we can avoid numerical issues due to cancellcation.

Problem 2

Consider the following polynomial:

$$p(x) = (x-2)^9 = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$$

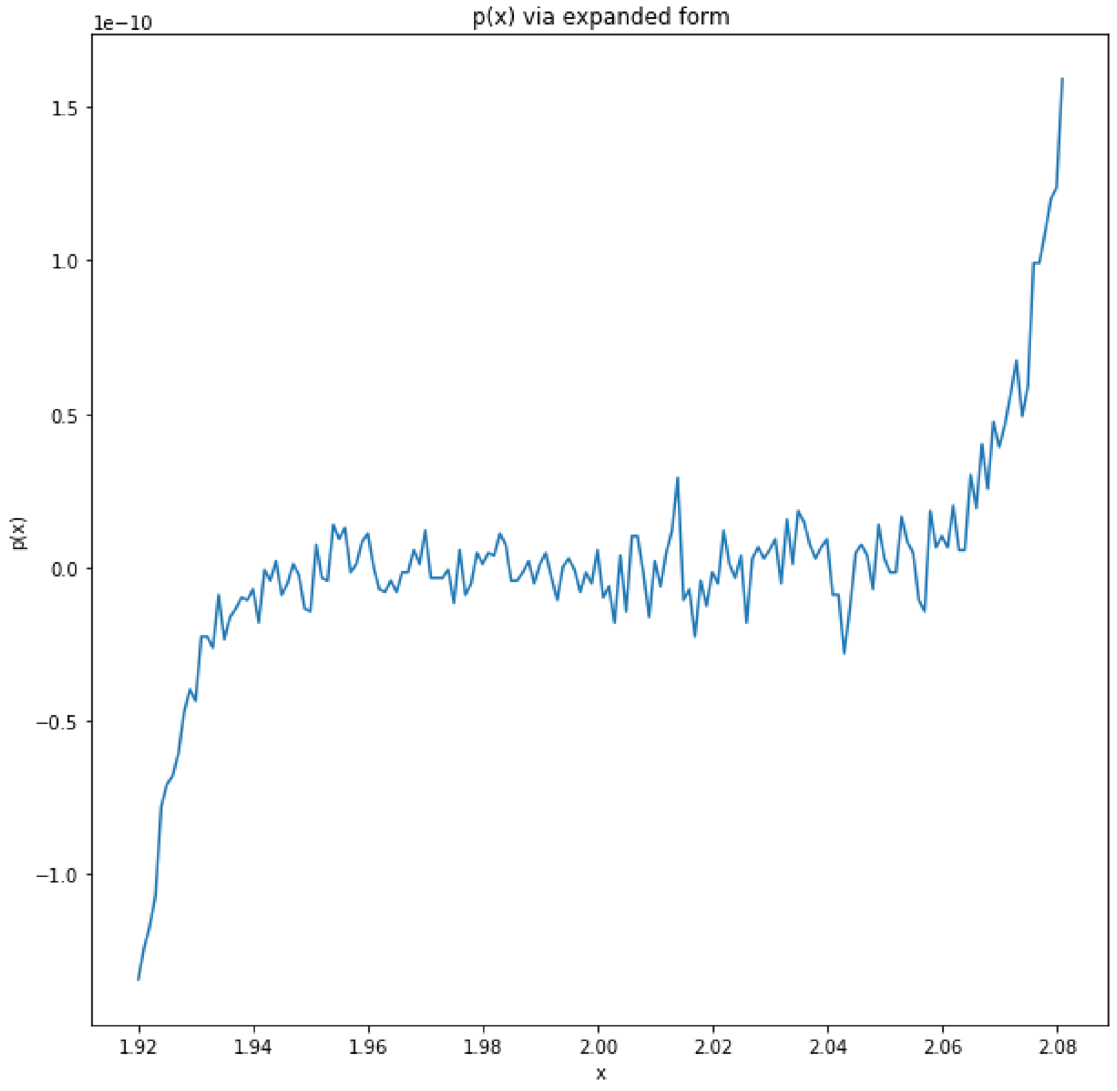
Taking $x = [1.920 : 0.001 : 2.080]$ we will examine the variation in evaluation depending on the form of the polynomial used.

[11]:
```
step = 0.001
x = np.arange(1.920, 2.080+step, step)

def p_factored(x):
    return (x-2)**9

def p_expanded(x):
    return x**9 - 18*x**8 + 144*x**7 - 672*x**6 +\
            2016*x**5 - 4032*x**4 + 5376*x**3 -\
            4608*x**2 + 2304*x - 512
```

a).

We plot $p(x)$ via the expanded form.

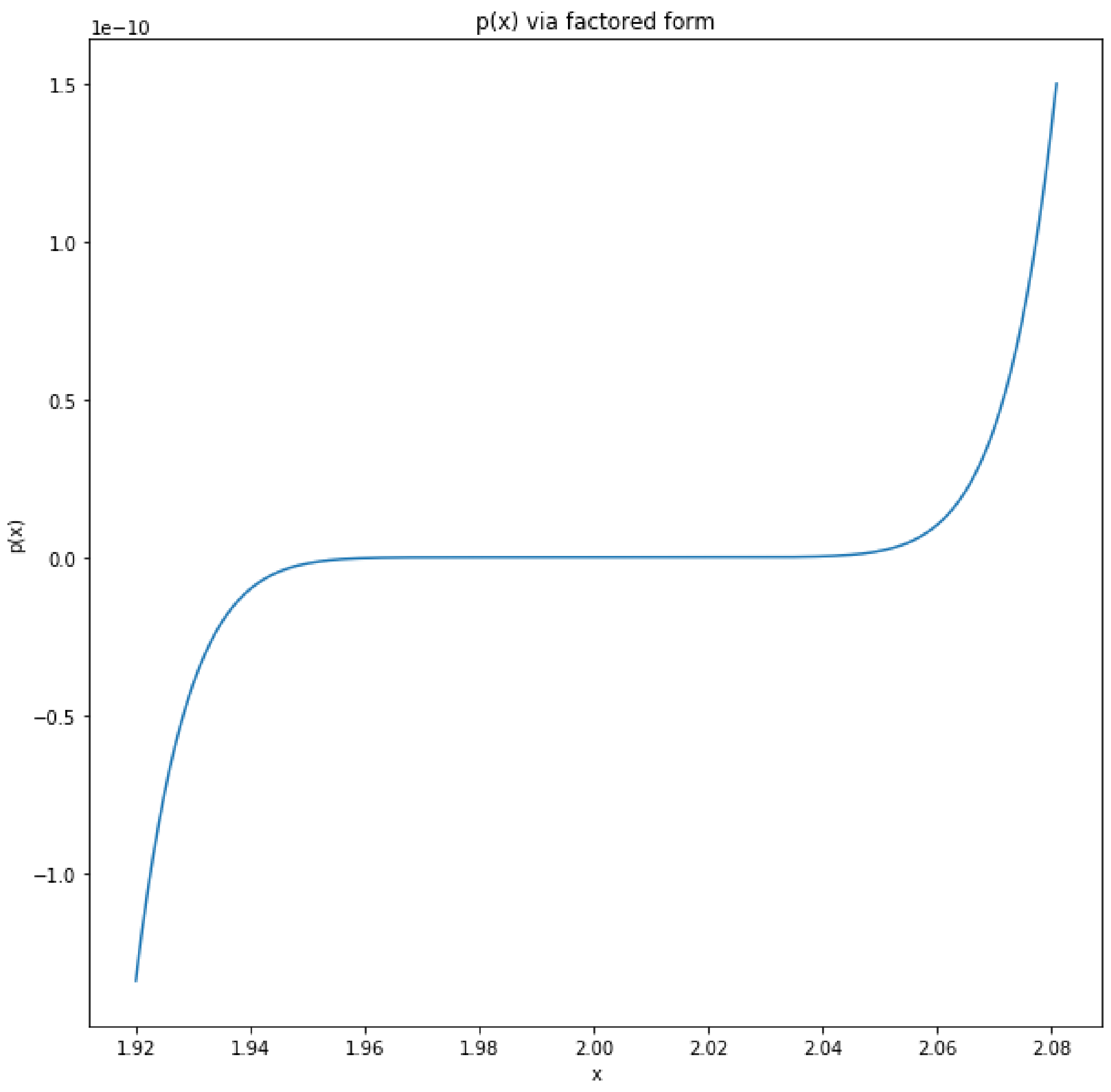
[12]:
```
fig, ax = plt.subplots(1,1,figsize=(10,10))
ax.set_title('p(x) via expanded form')
ax.set_xlabel('x')
ax.set_ylabel('p(x)')

ax.plot(x, p_expanded(x));
```



b).

We plot $p(x)$ using the factored form.

[13]:
```
fig, ax = plt.subplots(1,1,figsize=(10,10))
ax.set_title('p(x) via factored form')
ax.set_xlabel('x')
ax.set_ylabel('p(x)')

ax.plot(x, p_factored(x));
```



c).

Thinking about our function $(x-2)^9$ we expect only one zero at $x=2$ (with higher multiplicity) and a curve that is somewhat similar to a cubic, but with a wider range where the slope is almost zero around $x=2$. In the two plots above we can see wildly different outputs. In the first of the plots – where we have evaluated the polynomial using its expanded form – we see a jagged curve that in a general sense resembles our expectation. In the second plot we have evaluated the polynomial using the factored form, and we can see the curve is much smoother and looks exactly as we would expect. Given this we can clearly tell that the second plot (using the factored form) is the correct plot. The discrepancy comes from the way the polynomial was evaluated. When using the expanded form there are far more multiplies and adds introducing more and more numerical imprecision. The general shape looks correct, but there is significant numerical noise.

Problem 3

We consider computing $y = x_1 - x_2$ where $\bar{x}_1 = x_1 + \Delta x_1$ and $\bar{x}_2 = x_2 + \Delta x_2$ are approximations to the exact value. This operation analytically results in $\bar{y} = y + (\Delta x_1 + \Delta x_2)$ where we denote $\Delta y = \Delta x_1 + \Delta x_2$.

a).

We find upper bounds on the absolute error $|\Delta y|$ and the relative error $\frac{|\Delta y|}{|y|}$.

We will use the Triangle Inequality ($|a+b| \leq |a| + |b|$) and the Reverse Triangle Inequality ($||a| - |b|| \leq |a+b|$) to bound our expression.

For the absolute error a direct application of our inequality will give us our bound.

$$\boxed{|\Delta y| = |\Delta x_1 + \Delta x_2| \leq |\Delta x_1| + |\Delta x_2|}$$

For the relative error we can do the same thing with an extra step using the reverse inequality.

$$\frac{|\Delta y|}{|y|} = \frac{|\Delta x_1 + \Delta x_2|}{|x_1 - x_2|} \tag{1}$$

$$\tag{2}$$

$$|\Delta x_1 + \Delta x_2| \leq |\Delta x_1| + |\Delta x_2| \text{ and } |x_1 - x_2| \geq ||x_1| - |x_2|| \tag{3}$$

$$\tag{4}$$

$$\implies \boxed{\frac{|\Delta y|}{|y|} \leq \frac{|\Delta x_1| + |\Delta x_2|}{||x_1| - |x_2||}} \tag{5}$$

We used the Triangle Inequality to replace the numerator with something larger and the Reverse Triangle Inequality to replace the denominator with something smaller (making the whole expression larger).

With this we can see that when x_1 and x_2 are close to each other in magnitude the relative error is large. This follows because the numerator is large and the denominator is small.

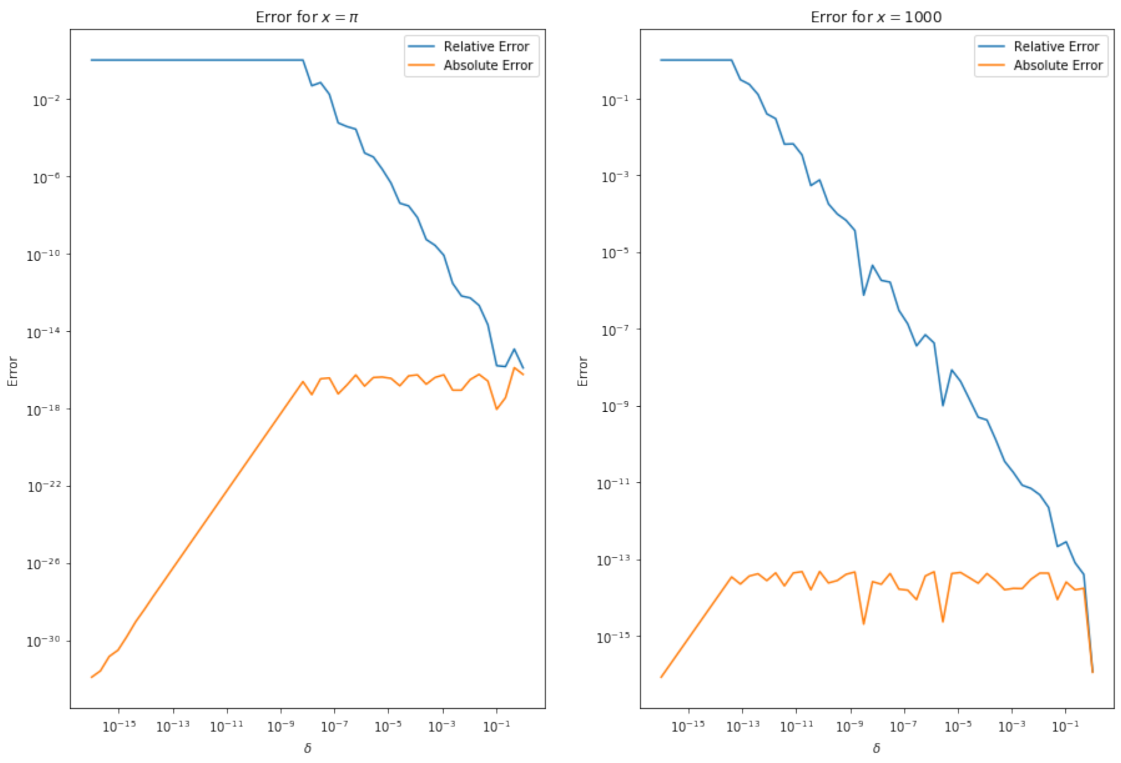
b).

We examine the expression $\cos(x+\delta) - \cos(x)$ and the numerical differences between it and an equivalent expression with no subtraction.

We can use a trig sum identity to transform our original expression into the following: $-2\sin(\frac{2x+\delta}{2})\sin(\frac{\delta}{2})$. We will then plot the difference between the two expressions below.

[16]:
```python
delta = np.logspace(-16, 0, base=10)
x_vec = [np.pi, 1E3] #Arbitrary

def f1(d, x):
    return np.cos(x+d)-np.cos(x)

def f2(d, x):
    return -2*np.sin((2*x+d)/2)*np.sin(d/2)
```

```
[17]: fig, ax = plt.subplots(1,2,figsize=(15,10))
      ax[0].set_title(r'Error for $x=\pi$')
      ax[1].set_title(r'Error for $x=1000$')

      for i,x in enumerate(x_vec):
          ax[i].set_xlabel(r'$\delta$')
          ax[i].set_ylabel('Error')

          ax[i].loglog(delta, np.abs(f1(delta, x)-f2(delta,x))/np.abs(f2(delta,x)))
          ax[i].loglog(delta, np.abs(f1(delta,x)-f2(delta, x)))

          ax[i].legend(['Relative Error', 'Absolute Error']);
```



In the plots above we see the relative and absolute error computed for $x = \pi, 1000$. We can see that in both cases the relative error starts out large and will decrease as δ increases while the absolute error does the opposite.

c).

Talyor expansion on $f(x+\delta) - f(x)$ results in,

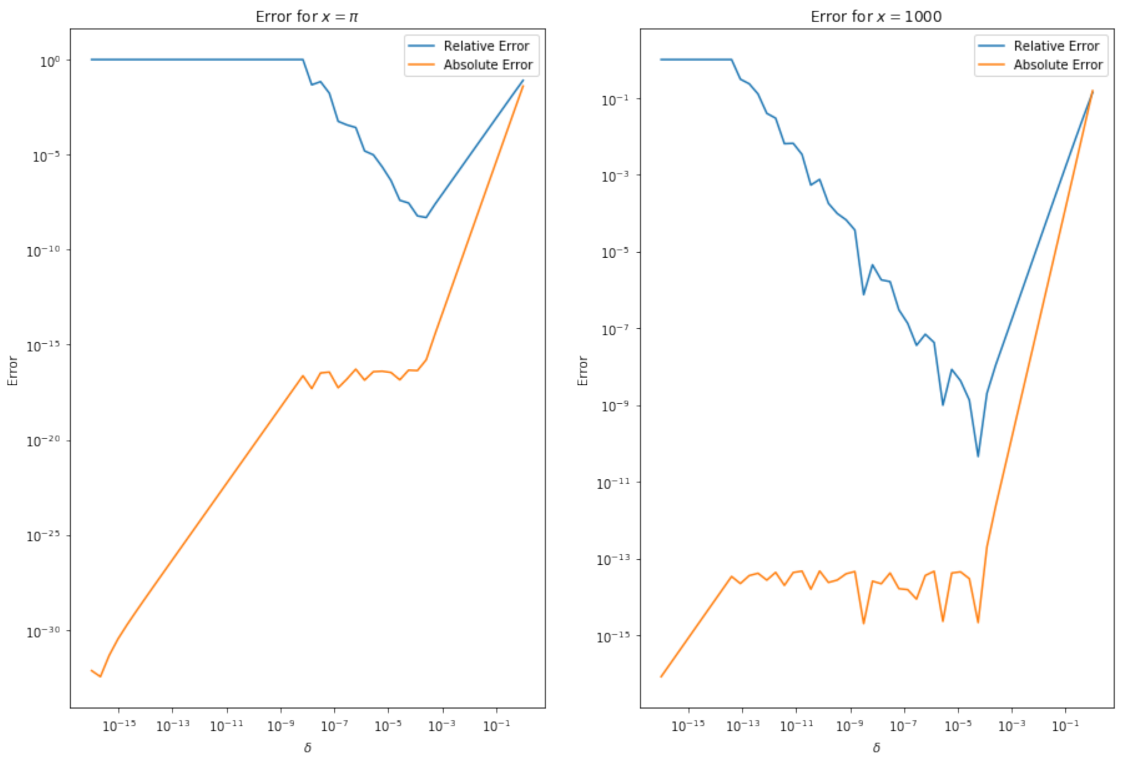
$$\delta f'(x) + \frac{\delta^2}{2} f''(\zeta),\ \zeta \in [x, x+\delta]$$

Applying this to $cos(x+\delta) - cos(x)$ we have the following result:

$$\delta(-sin(x)) + \frac{\delta^2}{2}(-cos(\zeta))$$

```
[18]: zeta = x_vec #Maximize cos on [pi, pi+epsilon] and [pi/4,pi/4+epsilon]

      def f3(d, x, z):
          return -d*np.sin(x)-(d**2/2)*np.cos(z)
```

```
[21]: fig, ax = plt.subplots(1,2,figsize=(15,10))
      ax[0].set_title(r'Error for $x=\pi$')
      ax[1].set_title(r'Error for $x=1000$')

      for i,x in enumerate(x_vec):
          ax[i].set_xlabel(r'$\delta$')
          ax[i].set_ylabel('Error')

          ax[i].loglog(delta, np.abs(f1(delta, x)-f3(delta,x,x))/np.abs(f3(delta,x,x)))
          ax[i].loglog(delta, np.abs(f1(delta,x)-f3(delta, x,x)))

          ax[i].legend(['Relative Error', 'Absolute Error']);
```



Again we see the absolute and relative error for $x = \pi, 1000$. When using the Taylor expansion we see that for larger delta (i.e closer to 1) the error is very large. Specifically, we see the same behavior as in the previous plots, but with a sharp increase towards the end of our range of δ. This makes sense as we are only using a first order approximation, so for larger δ we are leaving a lot of information out. Thus we can conclude that for larger δ the first method (using the trig identity) is the better option. For smaller δ both methods seem equivalent.

Problem 4

We want to show that $(1+x)^n = 1 + nx + \wr(x)$, $n \in \mathbf{Z}$ as $x \to 0$.

We will show that $(1+x)^n - 1 - nx = \wr(x)$ as $x \to 0$.

$$\lim_{x \to 0} \frac{|(1+x)^n - 1 - nx)|}{|x|} \tag{6}$$

$$\tag{7}$$

$$\text{Expanding } (1+x)^n \text{ as a Taylor series centered at 0 gives the following:} \tag{8}$$

$$\tag{9}$$

$$\lim_{x \to 0} \frac{|(1 + nx + \frac{n(n-1)x^2}{2} + \cdots) - 1 - nx)|}{|x|} \tag{10}$$

$$= \lim_{x \to 0} \frac{|\frac{n(n-1)x^2}{2} + \cdots|}{|x|} = 0 \tag{11}$$

Thus, because the limit is zero we can conclude that $(1+x)^n = 1 + nx + \wr(x)$, $n \in \mathbf{Z}$ as $x \to 0$.

Problem 5

We want to show $x\sin(\sqrt{x}) = \mathcal{O}(x^{\frac{3}{2}})$ as $x \to 0$.

$$\lim_{x \to 0} \frac{x\sin(\sqrt{x})}{x^{\frac{3}{2}}} = \lim_{x \to 0} \frac{\sin(\sqrt{x})}{\sqrt{x}} \tag{12}$$

$$\text{Taylor} \implies = \lim_{x \to 0} \frac{\sqrt{x} - x^{\frac{3}{2}} + \cdots}{\sqrt{x}} \tag{13}$$

$$= \lim_{x \to 0} 1 - x + \cdots = 1 \tag{14}$$

Thus, because the limit is constant we can conclude that $x\sin(\sqrt{x}) = \mathcal{O}(x^{\frac{3}{2}})$.

Problem 6

We will use the Bisection method on the factored and expanded form of the polynomial $f(x) = (x-5)^9$ which has a root with multiplicity 9 at $x = 5$.

[23]:
```
'''
Function: Bisection
Input:
    a: left end point guess
    b: righ end point guess
    f: a callable function to find the root
    tol: tolerance for the root
    imax: max iterations
    info: whether to return number of iterations
Output:
    -the root within the tolerance
    -number of iterations to identify root (optional)
Errors:
    -if given function is not callable
    -if no identifiable root bounded by initial guesses
'''
def bisection(a, b, f, tol=1E-4, imax=1000, info=False):
    #Check if f is callable
    if not callable(f):
        raise ValueError('Given function not callable')

    #Check if there is a zero
    if f(a)*f(b)>0:
        raise ValueError('Initial guesses do not bound a zero, or guesses are␣
 ↪poor.')

    #Bisect
    for i in range(imax): #Stay under max iterations
        c = (a+b)/2 #Take the midpoint

        if f(c)==0 or (b-a)/2<tol: #Check to see if root is found
            if info:
                return c, i+1
            return c

        if f(a)*f(c)<0: #Reset the interval
            b = c
        else:
            a = c

    return None #Something failed along the way

a=4.8
b=5.3
```

a).

Using the factored form.

```
[34]: def f_factored(x):
          return (x-5)**9

      root, i = bisection(a, b, f_factored, info=True)
      print('Root @ x=%f, Took %i iterations' % (root, i))
```

```
Root @ x=5.000012, Took 13 iterations
```

b).

Using the expanded form.

```
[35]: def f_expanded(x):
          return x**9 - 45*x**8 + 900*x**7 - 10500*x**6 + 78750*x**5 + \
                  -393750*x**4 + 1312500*x**3 - 2812500*x**2 + 3515625*x + \
                  - 1953125

      root, i = bisection(a, b, f_expanded, info=True)
      print('Root @ x=%f, Took %i iterations' % (root, i))
```

```
Root @ x=5.050000, Took 1 iterations
```

c).

Using the factored form of the polynomial our Bisection method found the root to within 10^-5 in 13 iterations – outperforming our required tolerance. However, using the expanded form the algorithm only ran for 1 iteration, and was not able to achieve the required tolerance. We can note that the root estimate it gave us was the first midpoint $\frac{5.3+4.8}{2} = 5.05$. To investigate what may have gone wrong we can evaluate the expanded polynomial at this location.

```
[44]: fe = f_expanded(5.05)
      ff = f_factored(5.05)

      print('Expanded p(5.05)=%.15f, p(5.05)==0? %s' % (fe, fe==0))
      print('Factored p(5.05)=%.15f, p(5.05)==0? %s' % (ff, ff==0))
```

```
Expanded p(5.05)=0.000000000000000, p(5.05)==0? True
Factored p(5.05)=0.000000000001953, p(5.05)==0? False
```

Ah! So the Bisection method computed the first midpoint, checked the function value at that midpoint, and then concluded it was the root. It only took one iteration because it thought that was all it needed. As we can see above, both methods evaluate to something close to zero, but it is only the expanded form that is identically zero.

Clearly the expanded form of the polynomial has enough numerical imprecision in it that it evaluated to zero when it should not have. All of the error in each multiply and add compounded to result in $p(5.05) = 0$ when that should not be the case. The function is indeed close to zero at around $x = 5$, but it was only the factored form the was stable enough to find the real root.