Numerical Algorithms

All implementation was done in python using numpy.

Problem set 1

Discretization

1. Trapezoidal Rule

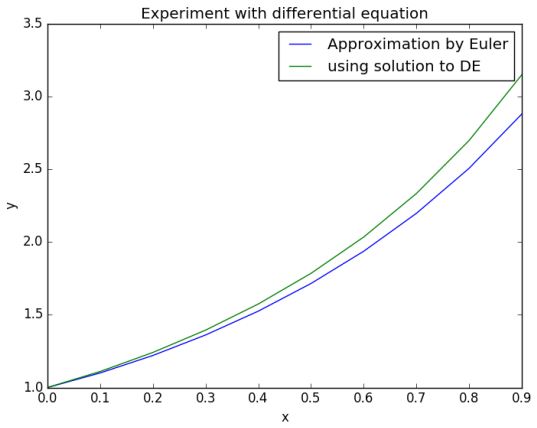
Computation was done using numpy's sin function and numpy's value for π.

We observe that the approximation of the integral is more accurate as we reduce the step size – or equivalently, increase n, the number of trapezoids used.

True value	2
h=0.1	1.99746892659
h=0.01	1.99998206504

```
'''
    @params
        f : Unary function that takes a float input and returns a float
        a,b: lower and upper range of integration
        h : step size for integration
    @returns
        ans: integral of f from a to b
'''
def trapezoidal_rule(f, a, b, h):
    # Doing it as stably as possible
    xl = a
    xu = xl + h
    ans = 0
    while xu<b:
        ans += h*(f(xl) + f(xu)) # Multiply by h here to avoid overflow?
        xl += h
        xu += h
    ans *= 0.5
    return ans
```

2. Differential Equation



We observe that the error in our approximation grows larger as x and y' increase. We consistently underestimate the function because of the assumption that the gradient stays constant over the step, whereas the function actually grows.

Code used:

```
'''''
    @params
        f : fn: float*float -> float
        n : number of values at step size of h
        h : step size
        y0: boundary condition for y at 0
    @return
        x,y : sequence of function values at points in the range(0:nh:h)
'''
def euler_method(f, n, h, y0):
    x = []
    y = []
    xk = 0
    yk = y0
    for k in range(0,n):
        x.append(xk)
        y.append(yk)
        yk += h*f(xk, yk)
        xk += h
    return x,y
```

Called as:

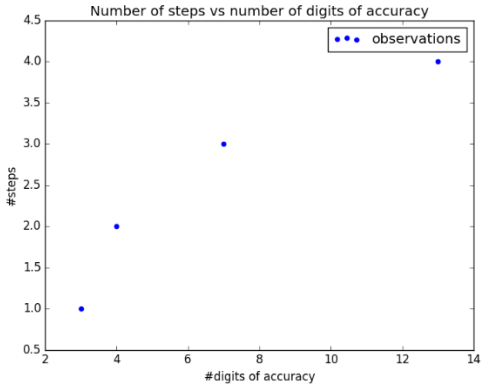
```
x,yapprox = euler_method(lambda xk,yk: 2*xk*yk - 2*xk*xk + 1, 10, 0.1, 1)
yexact = [ (np.exp(xk**2) + xk) for xk in x ]
```

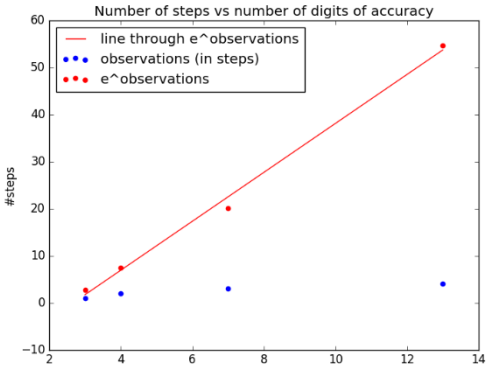
3. Newton's method

Correct value used: 1.41421356237

Value achieved: 1.41421356237

To verify the rate of convergence, we measure the number of iterations needed to achieve a certain number of digit-accuracy. We then take the exponential of the number of iterations needed.





We notice that the plot of #steps vs #digits-of-accuracy looks logarithmic. At the same time, the exponential of the same fits a straight line.

Unstable and Ill-conditioned Problems

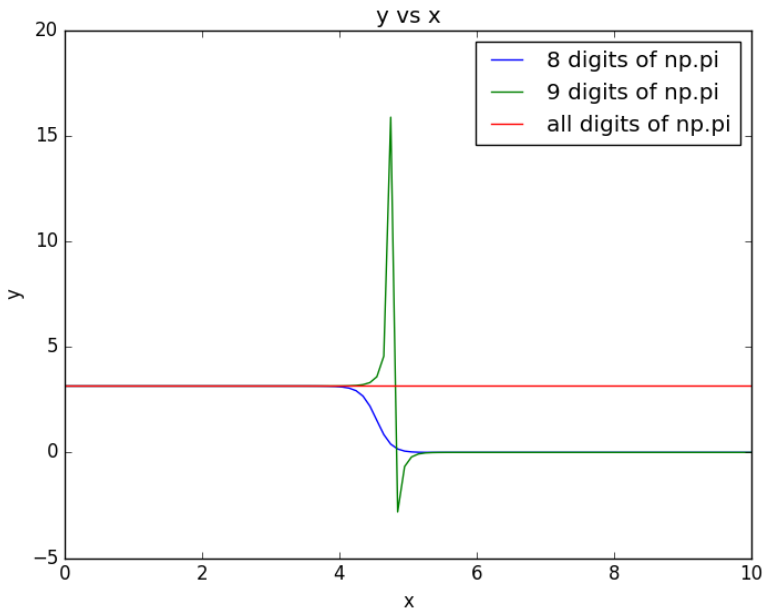
1. Newton's method

Let π_i denote π rounded to i digits. Notice that $\pi - \pi_8 > 0$ while $\pi - \pi_9 < 0$.

If we used $y_0 = \pi$ then the correct solution would have been $y(x) = \pi$.

Notice that if $\pi - y_0 > 0$ then the denominator of $y(x)$ is always positive and causes no major issues. It is a bad approximation for the true solution had $y_0 = \pi$, but the problem is stable.

However, if $\pi - y_0 < 0$ then the denominator switches sign at some x. Hence, the denominator is 0 at some value of x. The problem is hence unstable if $y_0 > \pi$ which is true if we use 9 digits of approximation for π.



Our suspicion is confirmed by the experiment. For $x < 4$, the two approximations work reasonably. However, after $x = 4$, approximating y_0 by π_8 causes the denominator to explode, forcing the function to 0.

On the other hand, the approximation of y_0 by π_9 has a very large absolute gradient in $x \in [4,6]$.

```
1.  f = lambda y0: lambda x: np.pi*y0/(y0 + (np.pi - y0)*(np.exp(x**2)))
2.  xs = np.linspace(0,10,100)
3.  pi8 = np.round(np.pi, 8)
4.  pi9 = np.round(np.pi, 9)
5.  f8 = f(pi8)
6.  f9 = f(pi9)
7.  y_pi8s = [f8(x) for x in xs]
8.  y_pi9s = [f9(x) for x in xs]
9.  y_true = [np.pi for x in xs]
10. plt.plot(xs,y_pi8s,label='8 digits of np.pi')
11. plt.plot(xs,y_pi9s,label='9 digits of np.pi')
12. plt.plot(xs,y_true,label='all digits of np.pi')
13. plt.xlabel('x')
14. plt.ylabel('y')
15. plt.title('y vs x')
16. plt.legend(loc='best')
17. plt.show()
```

2. "Qutb Minar to Gurugram"

The solutions are (1400.1, 699.8) and (1750., 874.75). The problem is not stable considering the large difference in answers caused due to a 10^{-3} change in input.

3. Polynomial roots

Let $P_1: x^3 - 102x^2 + 201x - 100$ and $P_2: x^3 - 102x^2 + 200x - 100$. Running it through a cubic equation solver we see that the roots are $\{100, 1\}$ and $\{100.01, 0.9948\}$. The value at $x = 1$ changes from 0 to 1.

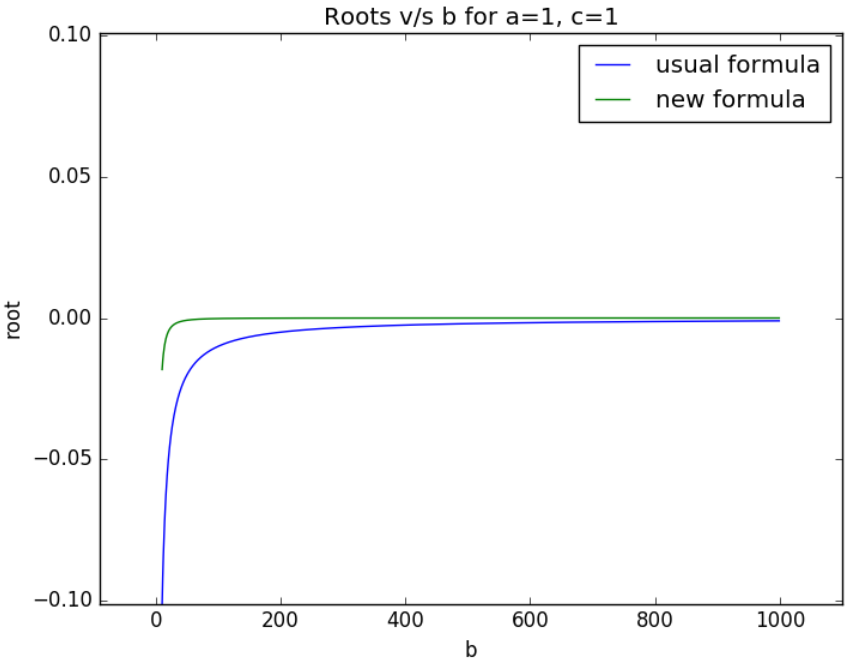
Unstable methods

1. Quadratic Equation

We see that $\sqrt{b^2 - 4ac} = 1000.0151$ approximately to 8 digits of accuracy. However, it is almost equal to b, which has 6 (or more?) digits of accuracy. In fact, the two values have 6 digits in common. The value of x is computed using subtraction of one of these values from the other. Hence, atleast 6 digits of accuracy are lost. The maximum number of accurate digits possible is 8 because $\sqrt{b^2 - 4ac}$ is computer only to 8 digits of accuracy. That means that x has 2 digits or less of accuracy.

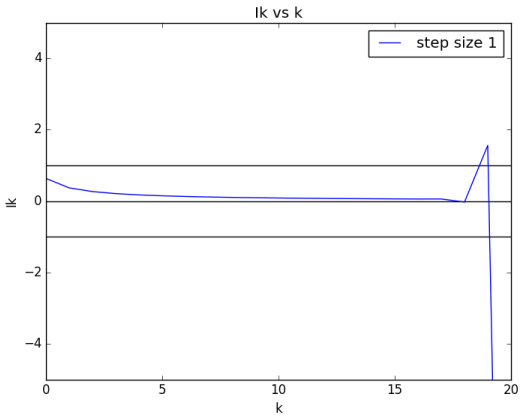
2. Alternate Formula for Quadratic root

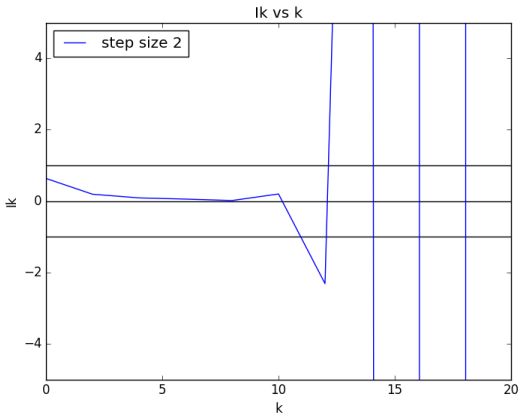
We know that errors are caused in the original formula when $\sqrt{b^2 - 4ac} \approx b$. However, in the new formula, for very large b, precision is lost due to dividing a small number $2c$ by a large number $\mathcal{O}(b^2)$



The experiment was performed for $a = c = 1$ and $b \in [10,1000]$. We see that the usual formula is unstable at low b. However, the new formula loses precision for large values of b and hence becomes 0 quickly.

3. Integration by parts





First, we look at the values of I_k obtained using $I_k = 1 - kI_{k-1}$. We also plot the lines $y = 0, y = 1, y = -1$. We expect the values of I_k to be less than 1 because it is integral of a term that is less than 1 over the range $(0,1)$. However, as the plot indicates, it is not. The error is because at $k = 18$, $kI_k \approx 1$. Hence, on the next iteration, we lose a lot of precision. Consequently, we get incorrect values for I_k.

Using the formula $I_k = \frac{1}{\pi} - \frac{k(k-1)}{\pi^2} I_{k-2}$, we run into larger errors due to similar reasons.

4. Standard Deviation Formulae

(a) The second formula is instable because when all the x_i are close together, $\forall i\ \bar{x} \approx x_i$. Hence, in the subtraction $x_i^2 - \bar{x}^2$, we lose a lot of precision. Further, if $x_i^2 - \bar{x}^2$ alternates signs, then we lose precision while adding them up too.

(b) We expect the second formula to produce a negative result when some information is lost. Specifically, notice that $x_i^2 - \bar{x}^2$ is sometimes positive and sometimes negative. If we make the positive values small enough to be lost, we're done.

For $x_1 = 1 - 11 * 10^{-12}$, $x_2 = x_3 \ldots = x_{10} = 1 + 10^{-12}$, $\bar{x} = 1.0$

The first formula produces 1.2960146036e-23 while the second formula produces -1.55431223448e-16.