Lab 6 Computational Geometry

December 2, 2019

1 Computational Geometry

Develop two methods to compute the integral of a function over a triangle. The first method uses Monte Carlo methods and the second uses quadrature methods. T is a triangle with vertices A, B, and C. For both problems, you will test your routines for the triangle with vertices A = (4,0), B = (3,4), and C = (0,1), and the function $f(x, y) = x^2 + y^2$. The triangle has area 7.5 and the integral of *f(x,y)* over the triangle is 72.5.

1.1 Computing Integrals with Monte Carlo

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

```
In [2]: #coordinates of the vertices
        A = (4,0)
        B = (3,4)
        C = (0,1)

        #function to integrate over
        f = lambda x, y: np.power(x, 2) + np.power(y, 2)
```

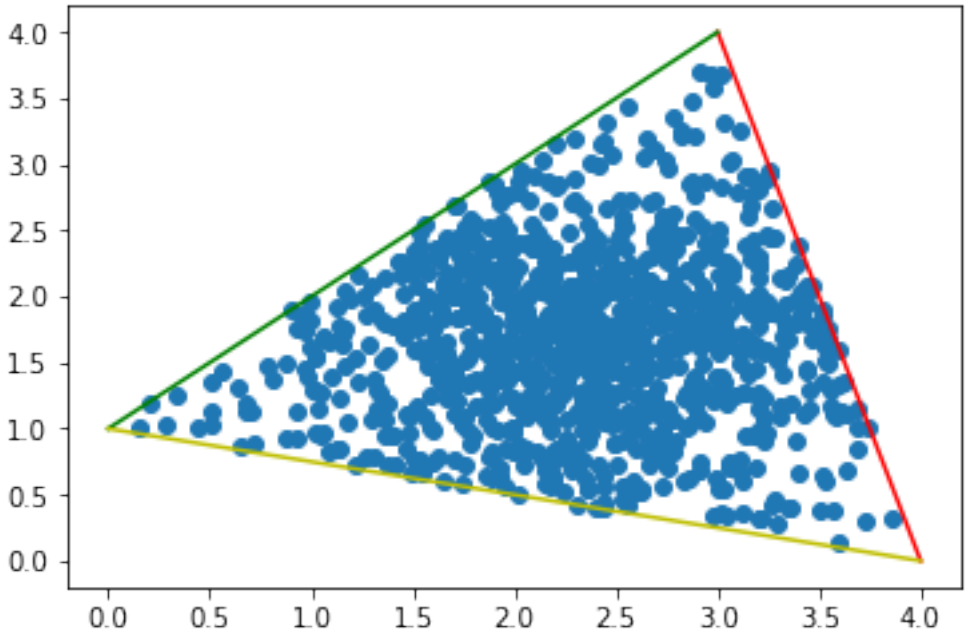
```
In [3]: def area(A, B, C):
            """
            Compute the area of a triangle given the vertices A, B, C.
            """
            return 1/2*(4*(4-1)+3*(1-0)+0*(0-4))
```

```
In [4]: def points_on_triangle(A, B, C, N, method):
            """
            Calculate random points inside a triangle. The function takes in
            the triangle's vertices A, B, C, N for the number of random samples, and
            method to determine the choice of selecting random points. Returns
            x and y coordinates.
            """

            #generate N random points for r1, r2, and r3
```

```
    r1, r2, r3 = np.random.random(N), np.random.random(N), np.random.random(N)

    #method 1
    if method == 1:
        #calculate method 1
        alpha = (r1/(r1+r2+r3))
        beta = (r2/(r1+r2+r3))
        rho = (r3/(r1+r2+r3))

    #method 2
    if method == 2:
        #if r1 and r2 is greater than 1
        for i in range(N):
            if (r1[i] + r2[i]) > 1:
                #replace r1 by 1 - r1 and r2 by 1 - r2
                r1[i] = 1 - r1[i]
                r2[i] = 1 - r2[i]
        #calculate method 2
        alpha = (1 - r1 - r2)
        beta = r1
        rho = r2

    #method 3
    if method == 3:
        #calculate method 3
        alpha = 1 - np.sqrt(r1)
        beta = np.sqrt(r1)*r2
        rho = np.sqrt(r1)*(1-r2)

    #calculate points
    p = [alpha[i]*np.array(A) + beta[i]*np.array(B) + rho[i]*np.array(C) for i in rang

    #split points into x and y coordinates
    x, y = zip(*p)

    #return coordinates
    return x, y
```

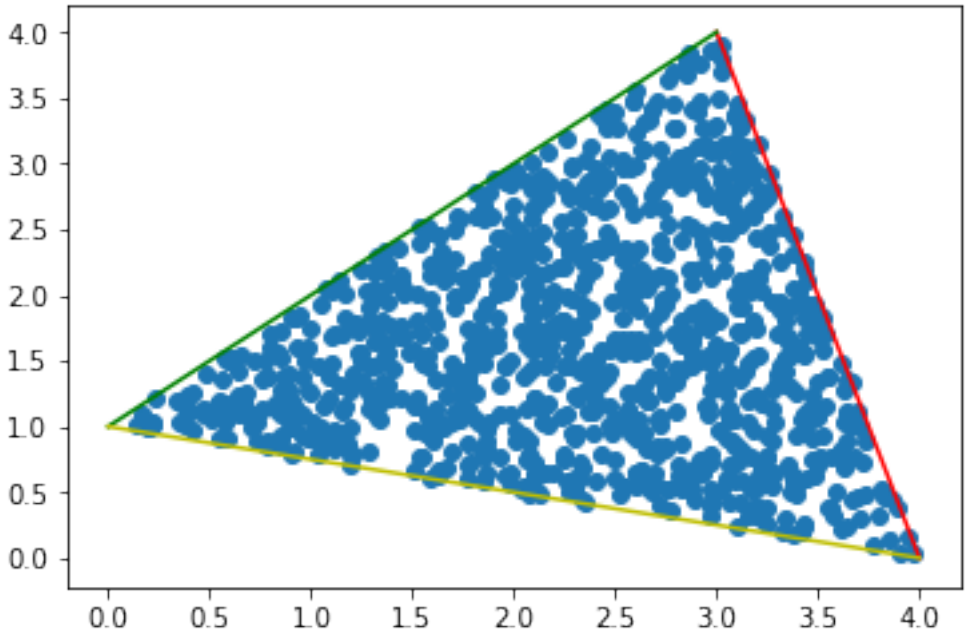
In [5]:
```
#create random points within the triangle using method 1
x, y = points_on_triangle(A,B,C,1000,1)

#plot x and y coordinates and outline triangle
plt.plot((A[0],B[0]),(A[1],B[1]), color='r')
plt.plot((B[0],C[0]),(B[1],C[1]), color='g')
plt.plot((A[0],C[0]),(A[1],C[1]), color='y')
plt.scatter(x,y)
```

Out[5]:
```
<matplotlib.collections.PathCollection at 0x1c0ec4423c8>
```



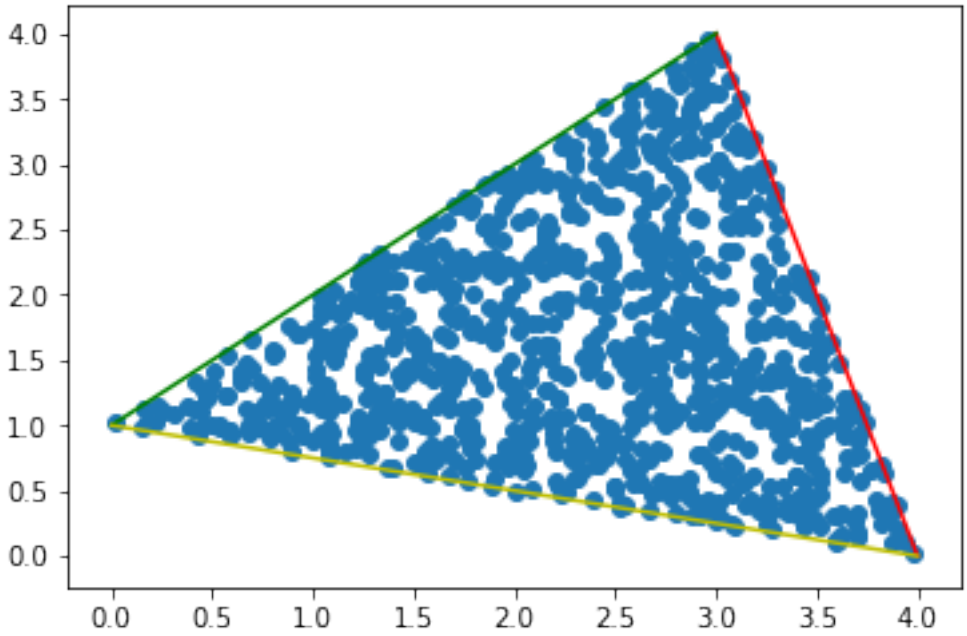
In [6]:
```
#create random points within the triangle using method 2
x, y = points_on_triangle(A,B,C,1000,2)

#plot x and y coordinates and outline triangle
plt.plot((A[0],B[0]),(A[1],B[1]), color='r')
plt.plot((B[0],C[0]),(B[1],C[1]), color='g')
plt.plot((A[0],C[0]),(A[1],C[1]), color='y')
plt.scatter(x,y)
```

Out[6]:
```
<matplotlib.collections.PathCollection at 0x1c0ec4e95f8>
```



In [7]:
```
#create random points within the triangle using method 3
x, y = points_on_triangle(A,B,C,1000,3)

#plot x and y coordinates and outline triangle
plt.plot((A[0],B[0]),(A[1],B[1]), color='r')
plt.plot((B[0],C[0]),(B[1],C[1]), color='g')
plt.plot((A[0],C[0]),(A[1],C[1]), color='y')
plt.scatter(x,y)
```

Out[7]:
```
<matplotlib.collections.PathCollection at 0x1c0ec569160>
```



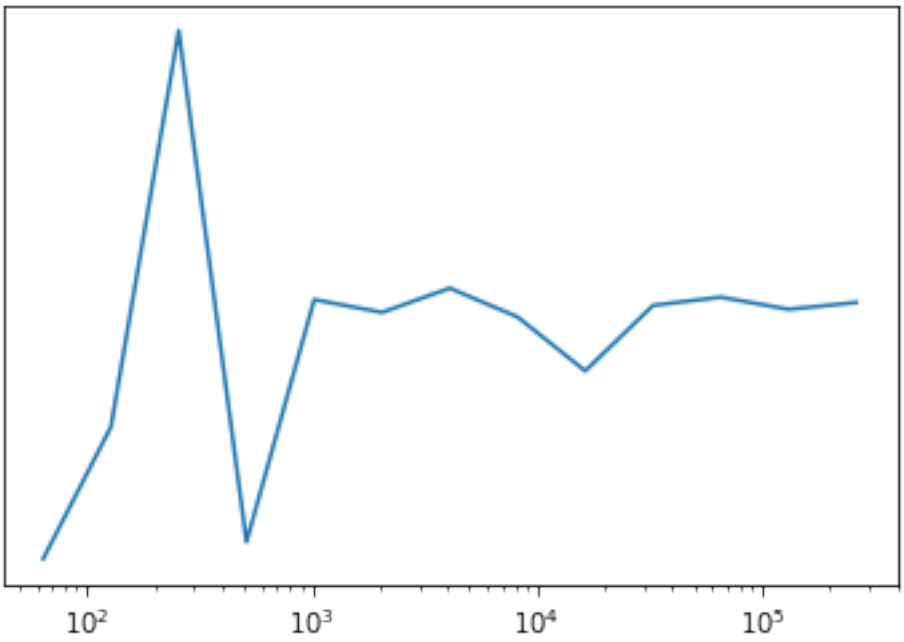
The three triangles all show 1000 random points within a triangle given vertices A, B, and C (defined in earlier cells). The first scatter plot has more random points oriented towards the center of the triangle and loose points towards the edges. The next two plots are similar, it looks like the random points are dispersed throughout the triangle.

```
In [8]: def mc(A, B, C, N, f, method):
            """
            Monte Carlo method to approximate integrals over triangles.
            Approximate the integral 100 times and compute the mean value
            of the approximation.
            """

            #initialize error array
            err = []

            #calculate area
            T = area(A, B, C)

            #exact value of the integral
            exact = 72.5

            #initialize integral calculation variable
            calc = np.zeros(100)

            #MC method
            #loop through each N value
```

```
            for h in N:
                #at each N value calculate the approximate of the integral
                for i in range(100):
                    #calculate points
                    x, y = points_on_triangle(A, B, C, h, method)
                    #calculate the approximate of the integral
                    calc[i] = np.abs(T)*(1/h)*np.sum(f(x, y))

                #take the mean of all the approximations
                approx = np.mean(calc)

                #calculate the error by subtracting the approximate solution from the exact
                #appen error values to err array
                E = abs(approx - exact)
                err.append(E)

            #plot loglog to calculate convergence
            plt.loglog(N, err)

            #return the approximate solution
            return approx
```

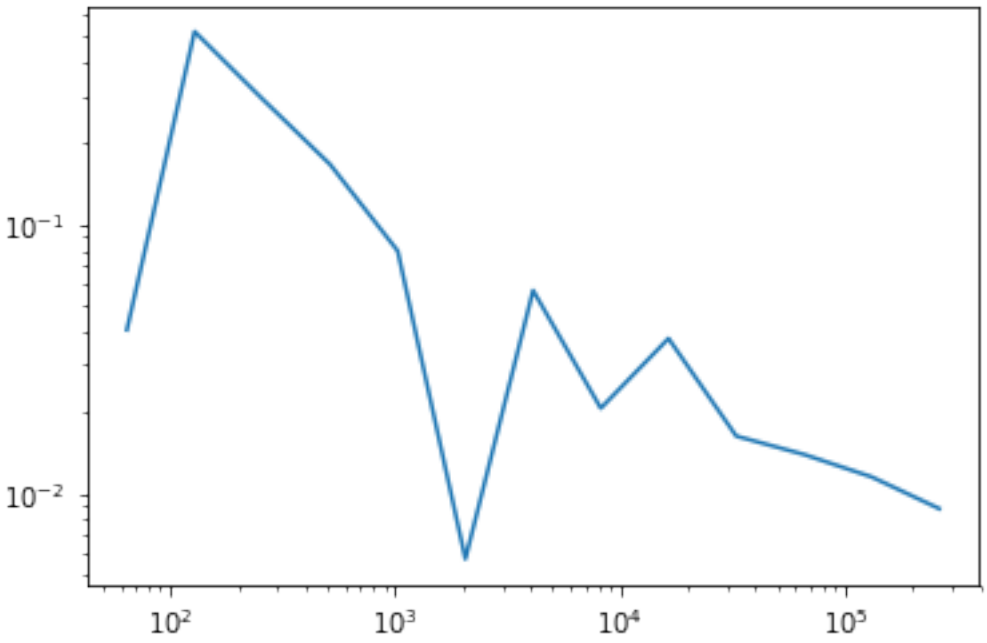
In [9]:
```
        #create N = 64, 128, ..., 262144 which is 2^6-2^18
        N = np.asarray([2**x for x in range(6, 19)])

        #call monte carlo method
        approx = mc(A, B, C, N, f, 1)
        print('Approximate solution',approx)
```

```
Approximate solution 67.962957392578
```



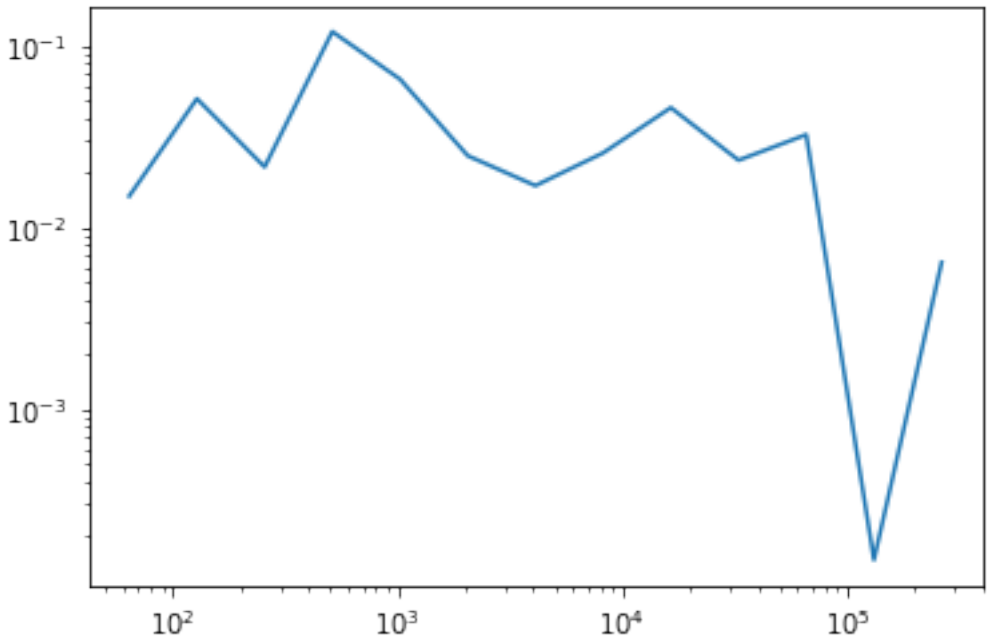
In [10]:
```
#call monte carlo method
approx = mc(A, B, C, N, f, 2)
print('Approximate solution',approx)
```

```
Approximate solution 72.49121072905378
```



```
In [11]: #call monte carlo method
         approx = mc(A, B, C, N, f, 3)
         print('Approximate solution',approx)
```

```
Approximate solution 72.50644650000955
```



1.2 Computing Integrals With Quadrature

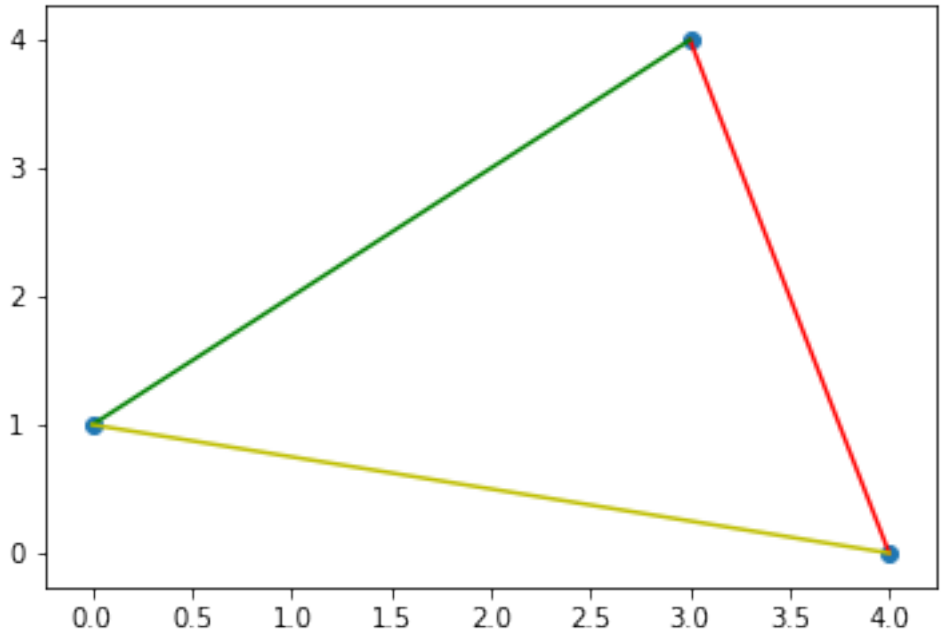
```
In [12]: def quadrature(A, B, C, f, method):
             """
             This function computes the quadrature points using the change of
             coordinates method for a general triangle and approximates the integral
             with the quadrature formula.
             """

             #calculate the area of the triangle given the verices
             a = area(A, B, C)

             #first order quadrature n = 3
             if method == 1:

                 #set x and y
```

```
            x = np.array([1, 0, 0])
            y = np.array([0, 1, 0])

            #initialize X and Y
            X = np.zeros(len(x))
            Y = np.zeros(len(y))

            #calculate the quadrature points using the change of coordinates
            for i in range(len(x)):
                X[i], Y[i] = np.array(A)*x[i] + np.array(B)*y[i] + np.array(C)*(1 - x[i]

            #approximate the integral using the quadrature formula
            approx = abs(a)*np.sum((1/3)*f(X,Y))

        if method == 2:

            x = np.array([0.816847572980459, 0.091576213509771, 0.091576213509771, 0.1081
            y = np.array([0.091576213509771, 0.816847572980459, 0.091576213509771, 0.4459
            w = np.array([0.109951743655322, 0.109951743655322, 0.109951743655322, 0.2233

            X = np.zeros(len(x))
            Y = np.zeros(len(y))

            for i in range(len(x)):
                X[i], Y[i] = np.array(A)*x[i] + np.array(B)*y[i] + np.array(C)*(1 - x[i]

            approx = abs(a)*np.sum(w*f(X,Y))

        return np.round(approx,1), X, Y
```

In [13]:
```
#approximation of the integral using the second order quadrature method n = 6
q, X, Y = quadrature(A, B, C, f, 1)
print('Approximate solution:', q)

#plot the quadrature points and include the edges of the triangle in the plot
plt.plot((A[0],B[0]),(A[1],B[1]), color='r')
plt.plot((B[0],C[0]),(B[1],C[1]), color='g')
plt.plot((A[0],C[0]),(A[1],C[1]), color='y')
plt.scatter(X, Y)
```

```
Approximate solution: 105.0
```

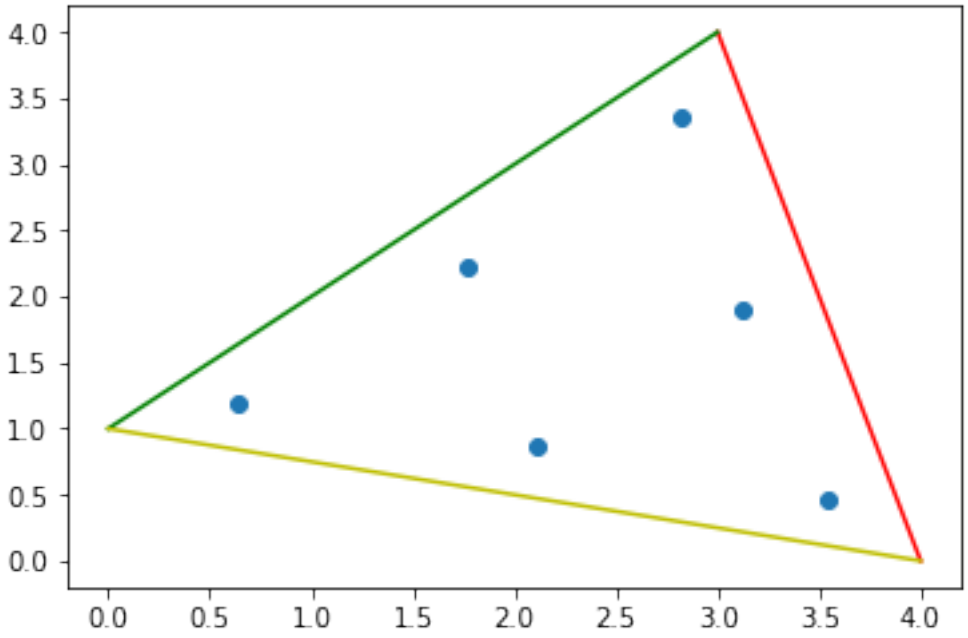
Out[13]:
```
<matplotlib.collections.PathCollection at 0x1c0ec6112e8>
```



In [14]:
```
#approximation of the integral using the second order quadrature method n = 6
q, X, Y = quadrature(A, B, C, f, 2)
print('Approximate solution:', q)

#plot the quadrature points and include the edges of the triangle in the plot
plt.plot((A[0],B[0]),(A[1],B[1]), color='r')
plt.plot((B[0],C[0]),(B[1],C[1]), color='g')
plt.plot((A[0],C[0]),(A[1],C[1]), color='y')
plt.scatter(X, Y)
```

```
Approximate solution: 72.5
```

Out[14]:
```
<matplotlib.collections.PathCollection at 0x1c0f095a240>
```



The quadrature points for the first quadrature method (n = 3) are located at the triangles vertices with an approximate solution of 105. This error is much larger compared to the second quadrature method (n = 6). The approximate solution for the second method is 72.5 which is the exact solution of the integral, making the error 0. This is because of the added x, y, and w values in the second method. You can see that the quadrature points of the second quadrature method are located within within the original triangle with vertices A, B, C.