Computational Physics Assignment 3

Dara Corr - 18483836

In this assignment, we will start to investigate algorithms and algorithm optimisation. We will be looking at Integration algorithms, namely the rectangle rule, the scipy.integrate.quad method and the Monte Carlo method.

In Task one we will create a programme for the rectangle rule to numerically integrate the function $f(x) = x^3 - 3x^2 - 2x + 190$ from -2 to 2 with 1000 as the number of steps. We can then compare that value to the exact value of the integral:

$$I = \int_{-2}^{2} f(x)dx = 744$$

In Part 2, we will analyse the accuracy of the rectangle rule by changing the step size of the numerical integration. Then we investigate the computation time for each evaluation of the integral and from that we can figure out the order of the Rectangle Rule algorithm from a log-log plot.

In Part 3, the same integral will be computed using Python's Scipy Library function scipy.integrate.quad(). We will compare the values obtained with both methods as well as the computation times for both methods.

In Part 4, we will look at the Monte Carlo method for solving integrals and use it to calculate the integral f(x). Then we will make a plot of performance of Monte Carlo method versus number of trials and we will also plot each point to get a visualisation of how the method works.

In the final part of the assignment, we will be simulating a game of craps and we will use the monte carlo method as a basis for plotting the probabilities of winning and losing the game.

```
In [126]: #part 1
          import numpy as np
          import matplotlib.pyplot as plt #import numpy and matplot lib

          #define function for rectangle rule
          def RectArea(F,a,b,N): #F = Function, a = lower limit, b = upper limit, n = number of rectangles
              total = 0
              dx = (b-a)/(N) #defining step size from parameters b,a and N
              for i in range(N):
                  total += F((a + (i*dx))) #adding rectangles of width dx (h) to find total area
              return dx*total

          #try for x^2 from -2 to 2
          def f(x):
              return x**3 - 3*x**2 - 2*x + 190

          print("Using the Rectangle Rule Algorithm I have just defined, I found the area of the function f to be {0:0.4f} units squared".format(RectArea(f,-2,2,1000))) #print area for the function f using this rectangle
           rule function
```

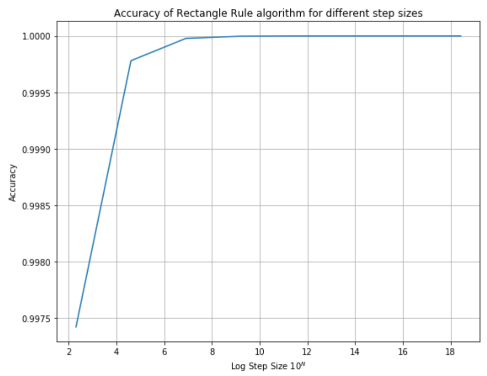
```
Using the Rectangle Rule Algorithm I have just defined, I found the area of the function f to be 743.984 units squared
```

The rectangle method produces an accurate value for I with 1000 steps. This value is 99.9978% accurate.

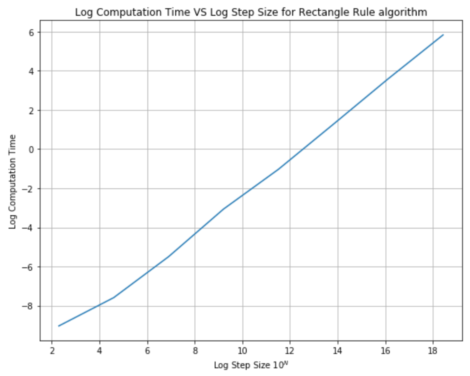
```
In [127]: import time

          #### change (6) to (8) in these arrays and the loop but for testing purposes keep as 6 to save computation time


          A_vals = np.zeros(8) #array to hold integral values
          T_vals = np.zeros(8) #array to hold time values

          for k in range(8):
              start = time.perf_counter()  #takes time measurement before computation
              A_vals[k] = RectArea(f,-2,2,int(10**(k+1))) #makes array of 8 computations of the Area integral for different powers of 10
              T_vals[k] = time.perf_counter() - start #time difference before and after computation is time taken to do each computation

          Accuracy =  np.zeros(8) #array to hold accuracy values
          for j in range(8):
              Accuracy[j] = 1 - (744 - A_vals[j])/744 #accuracy is 1 - error. error is absolute value of (real value - estimate)/real value
```

F	Accuracy	Time
742.08	99.742%	9.56390001e-05
743.8368	99.978%	3.34504000e-04
743.983968	99.9978%	3.98044800e-03
743.99839968	99.99978%	3.47202230e-02
743.99984	99.999978%	2.98437539e-01
743.999984	99.9999978%	2.94140128e+00
743.9999984	99.99999978%	2.99420488e+01
743.99999984	99.99999998%	3.05937274e+02

```
In [149]: fig= plt.figure(figsize=(9,7)) #plotting
          #plot log log graph of accuracy against step size for rectangle algorithm
          plt.plot(np.log([10,10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8]),np.abs(Accuracy))
          plt.title("Accuracy of Rectangle Rule algorithm for different step sizes")
          plt.xlabel("Log Step Size $10^N$")
          plt.ylabel("Accuracy")
          plt.grid()
          plt.show()
```



In [151]:
```
fig= plt.figure(figsize=(9,7))#plot size
#plot log log graph of Computation Time VS step size
plt.plot(np.log([10,10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8]),np.log(T_vals[0:8]))
plt.title("Log Computation Time VS Log Step Size for Rectangle Rule algorithm")
plt.xlabel("Log Step Size $10^N$")
plt.ylabel("Log Computation Time")
plt.grid()
plt.show()

#use polyfit function to find the trendline slope of the linear graph which is then used to find the order of the algorithm
m,c = np.polyfit(np.log([10,10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8]),np.log(T_vals[0:8]),1)
print("The order of this Algorithm is {0:5.3f}".format(m))
print("we can say that the rectangle rule is 1st order ")
```



```
The order of this Algorithm is 0.943
we can say that the rectangle rule is 1st order
```

We see from the table above that increasing the number of steps N, greatly increases accuracy of the rectangle method. As we can see from the first plot, values calculated using the rectangle method almost equal the exact value of the integral. If we want to be more precise using the rectangle method, we see that it is very ineffecient.

In the second graph we have showed that the order of the Rectangle Method is 1. This means that for example, by multiplying the number of steps by 10, the computation time is also scaled up by 10. This is not very noticable for small to medium values of N but for N = 10^7 or 10^8 it takes a significant amount of time to compute, about 30 seconds and 300 seconds respectively.

In [195]:
```
#part 3
import scipy.integrate #import scipy.integrate library
start = time.perf_counter()
print("The Value of the integral using scipy.integrate.quad is {0:0.4f}".format(scipy.integrate.quad(f,-2,2)[0]))
endtime = time.perf_counter() -  start
print("Computation time for scipy.integrate.quad is {0:0.7f} s".format(endtime)) #print computation time
```

```
The Value of the integral using scipy.integrate.quad is 744.0000
Computation time for scipy.integrate.quad is 0.0017178 s
```

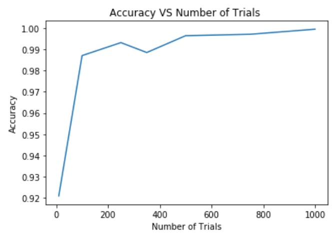
It is immediately apparent that the scipy.integrate.quad() function has been optimised for efficiency. It computes the exact value of the ntegral in only 0.00127 seconds. The rectangle method could only achieve 99.9978% accuracy with N = 1000 and in about 3 times the computation time.

In [241]:
```
#part 4
import random

def montecarlo(F,xmin,xmax, N): #Define function for Monte-Carlo Method F = function, xmin is lower x limit and xmax is upper x limit
    x_values = np.linspace(xmin,xmax,2000) # n is the number of trials

    y_values = np.zeros(2000)
    for i in range(2000):
        y_values[i] = F(x_values[i]) #obtain y values so we can find y_max for box area

    ymin = 0 #want area above x axis, so minimum value is 0
    ymax = np.max(y_values)
    boxarea = int((abs((xmax-xmin) * (ymax-ymin)))) #box area
    boxratio = 744/boxarea #ratio of integral to box area

    total = 0
    for j in range(N):
        x = random.uniform(xmin,xmax) #create random x value and y value inside box
        y = random.uniform(ymin,ymax)
        if y <= f(x):
            total += 1 #if its under the line drawn by the function, add one to the total
    print("area =", total) #area/integral = total
    return total


start = time.perf_counter()# find computation time for Monte Carlo
print(montecarlo(f,-2,2,761))
endtime = time.perf_counter() - start
print("Computation time is {0:0.5f} s".format(endtime))
```

```
area = 745
745
Computation time is 0.03110 s
```

In [252]:
```
boxarea = 761.2 #note: summation/trials is approximately equal to integral/box area
actual = 744/boxarea #boxarea from last cell
MonteCarloValues = np.zeros(7) #generate values of different number of trials for Monte Carlo to plot accuracy against no. of trials
MonteCarloValues[0] = montecarlo(f,-2,2, 10)/10
MonteCarloValues[1] = montecarlo(f,-2,2, 100)/100
MonteCarloValues[2] = montecarlo(f,-2,2, 250)/250
MonteCarloValues[3] = montecarlo(f,-2,2, 350)/350
MonteCarloValues[4] = montecarlo(f,-2,2, 500)/500
MonteCarloValues[5] = montecarlo(f,-2,2, 750)/750
MonteCarloValues[6] = montecarlo(f,-2,2, 1000)/1000

accuracies = np.zeros(7)
for i in range(7):
    accuracies[i] = 1 - abs((actual - MonteCarloValues[i])/actual) #calculate accuracy values

plt.plot([10,100,250,350,500,750,1000], accuracies) #create the plot
plt.title("Accuracy VS Number of Trials ")
plt.xlabel("Number of Trials")
plt.ylabel("Accuracy")
plt.show()
```

```
area = 9
area = 99
area = 246
area = 346
area = 487
area = 731
area = 977
```



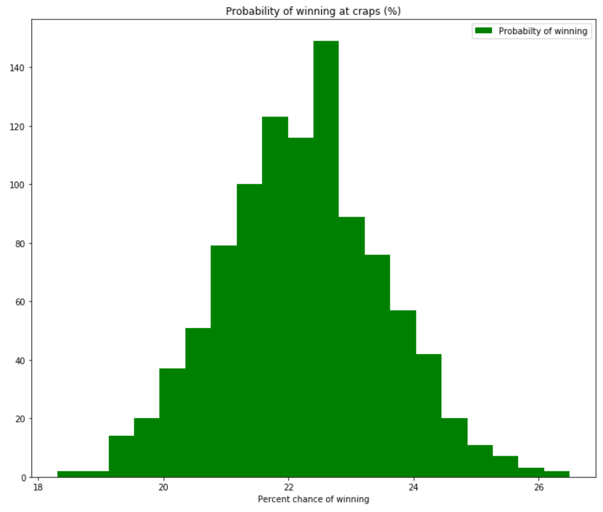
In [ ]:

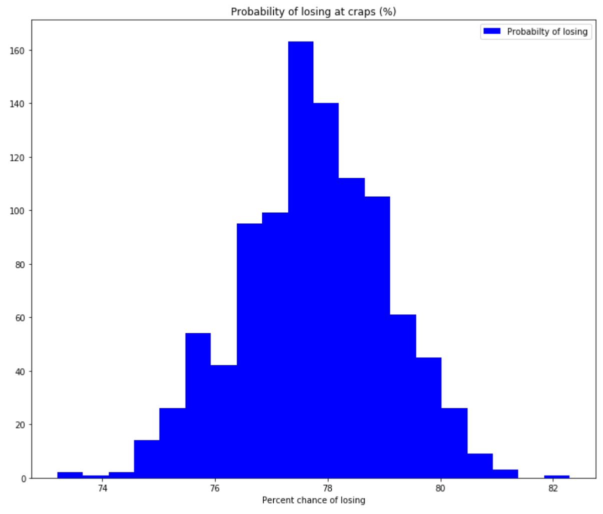
The Area for the Monte Carlo method is comparable to the area found from other numerical methods but it is the less accurate than the other two methods for a low amount of trials. From the graph plotted above, we see that accuracy increases for a higher number of trials.

In [253]:
```
#part 5 craps
def reroll(firstroll): #define recursive function for "roll for point" part of game
    roll = random.randint(1,6) + random.randint(1,6)
    if roll == 7:
        pass
    elif roll == firstroll:
        total += 1
    else:
        reroll(firstroll)


#play craps game n number of times
#function returns number of wins
#this is the main body of the game
#winning conditions add 1 to total and losing conditions add nothing to total

def craps(n):
    total = 0
    while n > 0:
        roll_0 = random.randint(1,6) + random.randint(1,6)  #first roll is 2 dice rolls evaluated by these random functions
        if roll_0 == 7 or roll_0 == 11:
            total += 1
        elif roll_0 == 2 or 3 or 12:
            pass
        else:
            reroll(roll_0)
        n -= 1
    return total
```

In [208]:
```
#probability/monte carlo:
n = 10000
craps_array = np.zeros(n) #array to hold craps values

totalwins = 0
for i in range(n):
    craps_result = craps(1)
    craps_array[i] = craps_result

totalwins = np.sum(craps_array)#find total wins and losses for one craps game repeated 1000 times
totallosses = n - totalwins

win_prob = totalwins/n #divide by n to find probabilities of winning and losing
loss_prob = totallosses/n

print("Probability of Winning one game of Craps is {0:0.4f}".format(win_prob))
print("Probability of Losing one game of Craps is {0:0.4f}".format(loss_prob))

#make histogram of probability of winning and histogram of probability of losing
N = 1000
percent_wins = np.zeros(N)
percent_losses =np.zeros(N)


for i in range(N):
    percent_wins[i] = craps(N) #calculate %wins for histogram plot
    percent_losses[i] = 1000 - craps(N)



plt.figure(figsize = (12,10)) #plot histogram
plt.hist(percent_wins/10, bins = 20, label="Probabilty of winning", color="green") #divide by 10 to get % since 1000 trials. put into bins of 20
plt.title("Probability of winning at craps (%)")
plt.xlabel("Percent chance of winning")
plt.legend(loc='upper right')
plt.show()

plt.figure(figsize = (12,10)) #plot histogram for probability of losing
plt.hist(percent_losses/10, bins = 20, label="Probabilty of losing", color="blue") #divide by 10 to get % since 1000 trials. put into bins of 20
plt.title("Probability of losing at craps (%)")
plt.xlabel("Percent chance of losing")
plt.legend(loc='upper right')
plt.show()
```

```
Probability of Winning one game of Craps is 0.2271
Probability of Losing one game of Craps is 0.7729
```





In part 5 I designed a programme to simulate a game of craps. for a given number n of craps games, the number of wins is returned.

Plotted above are probability distribution of playing craps for 1000 games. The probability of winning ranges from about 18% to just over 26% and the expected probability of winning is 22.71% . The probability distribution of losing ranges from about 73% to 82% with an expected probability of losing of 77.29%.

In [ ]: