Computational Physics PHYS-E0412 - Homework Week 3

Ari Viitala 432568

In [127]:
```
import numpy as np
import matplotlib.pyplot as plt
```

(i) Write a code that evaluates U(d) using the Metropolis Monte Carlo integration

In [10]:
```
#sampling function
def g(x):
    return np.exp(-x**2)
```

In [53]:
```
def metro(x, N, delta, d):
    g_now = g(x)
    accepted = 0
    xt = x
    ds = [] #U values
    for i in range(0, N): #iterate N steps
        index = np.random.randint(0,6) #select random coordinate to change
        xt = x.copy()
        rng = (np.random.random() - 0.5) * delta
        xt[index] += rng #perturb that coordinate
        g_trial = g(xt)
        if g_trial[index]/g_now[index] > np.random.random():
            #accept the change if it is larger than a random one
            x = xt.copy()
            g_now = g(x)
            accepted += + 1

        #calculate norm
        norm = np.linalg.norm(x[:3] - x[-3:] - np.array([d,0,0]))

        #we filter out zeros to prevent troubles with infs
        if norm != 0:
            ds.append(1 / norm #append the list with the U value



    mean_fs = np.array(ds).mean() #return the mean of the functionvalues
    acceptance = accepted / (N * 1.0) #return the acceptance value
    d_out = delta * np.log(0.5)/np.log(acceptance) #return the scaled delta

    return mean_fs, x, d_out, acceptance
```

In [94]:
```
def U(d, runs):
    #run the metropolis integration for certain d
    delta = 1
    x = np.array([0,0,0,d,0,0], dtype = np.float)
    N = 1000

    results = []
    acceptances = []
    res, new_x, delta, acceptance = metro(x, N, delta, d)

    for i in range(0,runs):
        #update the delta in between runs and save the values
        res, new_x, delta, acceptance = metro(new_x, N, delta, d)
        results.append(res)
        acceptances.append(acceptance)
        res, new_x, delta, acceptance = metro(new_x, N, delta, d)

    #return vectors of the individual simulation results and acceptance rates
    return np.array(results), np.array(acceptances)
```

(ii) Calculating U values, acceptance rates and error estimates

Case U(0)

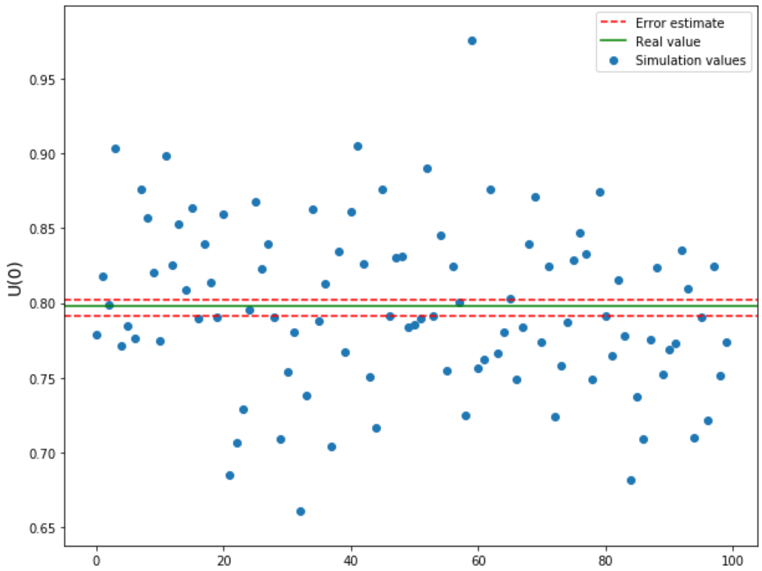
The error estimates are calculated usgin the formula $\epsilon = \frac{\sigma}{\sqrt{M}}$ where σ is the standard deviation between values given by individual metropolis runs and M is the number of runs.

In [113]:
```
runs = 100
d = 0
res0, acceptances0 = U(d, runs)
print("Value for U(0): {}".format(res0.mean()))
print("Error estimate: {}".format(res0.std() / np.sqrt(runs)))
print("Analytical U(0): {}". format(np.sqrt(2 / np.pi)))
```

```
Value for U(0): 0.7967746991405673
Error estimate: 0.005519667922748992
Analytical U(0): 0.7978845608028654
```

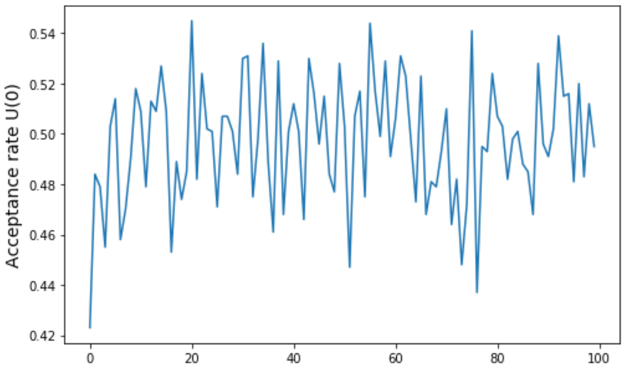
The algorithm seems to return the correct value within a reasonable error interval. Below can be seen that the actual value sits nicely between the error estimates.

```
In [114]: plt.figure(1, (10,8))
          plt.scatter(list(range(0,runs)), res0, label = "Simulation values")
          plt.axhline(res0.mean() - res0.std() / np.sqrt(runs), color = "red", label = "Error estimate", linestyle = "--")
          plt.axhline(res0.mean() + res0.std() / np.sqrt(runs), color = "red", linestyle = "--")
          plt.axhline(np.sqrt(2 / np.pi), color = "green", linestyle = "-", label = "Real value")
          plt.ylabel("U(0)", size = 14)
          plt.legend()
          plt.show()
```



The acceptance rates seem to oscillate around 0.5. The method for adjusting the delta is not the most delicate so the convergence probably wont get any better than this.

```
In [99]: plt.figure(1, (8,5))
         plt.plot(acceptances0)
         plt.ylabel("Acceptance rate U(0)", size = 14)
         plt.show()
```



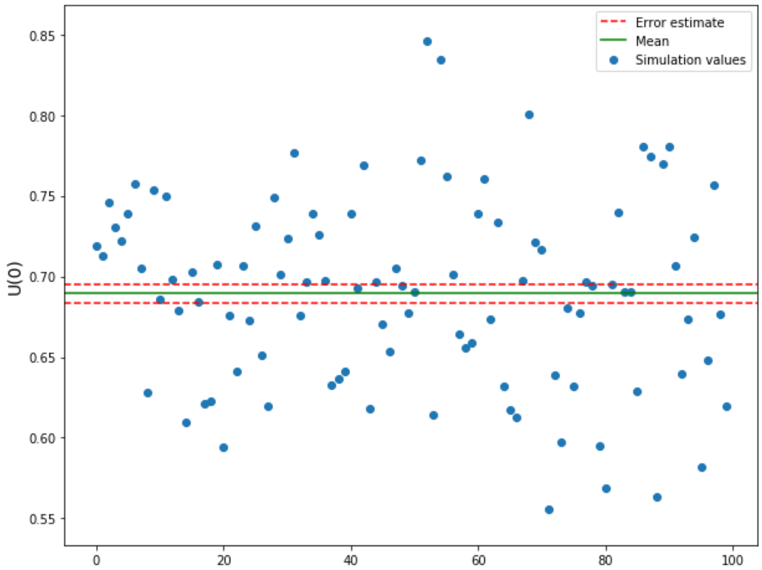
Case U(1)

The values for $U(1)$ seems to be about 0.69. Based on the previous part the real value probably falls within the error estimates. Also by looking at the scatter plot, this seems like a reasonable guess. The acceptance rates, like before, oscillate around 0.5.

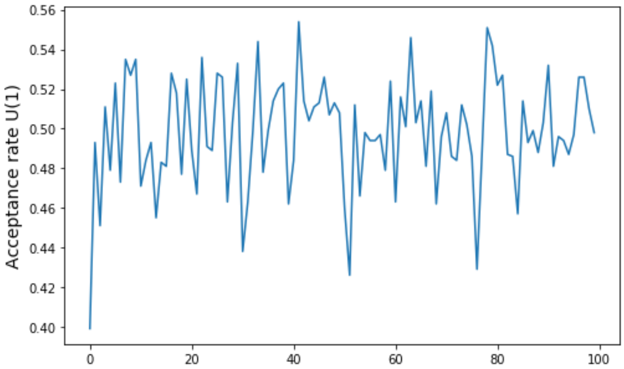
```
In [116]: runs = 100
          d = 1
          res1, acceptance1 = U(d, runs)
          print("Value for U(1): {}".format(res1.mean()))
          print("Error estimate: {}".format(res1.std() / np.sqrt(runs)))
```

```
Value for U(1): 0.6895531664841696
Error estimate: 0.00586530580113888
```

In [128]:
```
plt.figure(1, (10,8))
plt.scatter(list(range(0,runs)), res1, label = "Simulation values")
plt.axhline(res1.mean() - res1.std() / np.sqrt(runs), color = "red", label = "Error estimate", linestyle = "--")
plt.axhline(res1.mean() + res1.std() / np.sqrt(runs), color = "red", linestyle = "--")
plt.axhline(res1.mean(), color = "green", linestyle = "-", label = "Mean")
plt.ylabel("U(0)", size = 14)
plt.legend()
plt.show()
```



In [118]:
```
plt.figure(1, (8,5))
plt.plot(acceptance1)
plt.ylabel("Acceptance rate U(1)", size = 14)
plt.show()
```

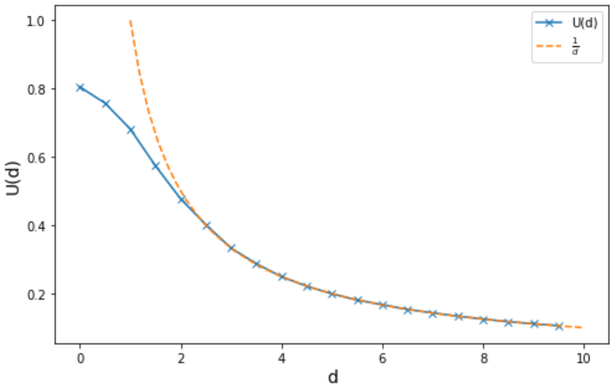


(iii) Investigate if the large d the interaction energy U(d) follows the 1/d-law

Here we calculate $U(d)$ values for 20 different d values between 0 and 10. If we plot the simulated values and also calculate directly values for $\frac{1}{d}$ we see that the plots align perfectly after about $d = 2$. Hence, $U(d)$ clearly follows the $\frac{1}{d}$-law.

In [125]:
```
ds = np.array(list(range(0,20))) / 2
us = []
for i in ds:
    res, acc = U(i, 100)
    us.append(res.mean())
```

In [131]:
```
x = np.linspace(1, 10)
one_per_x = 1 / x
plt.figure(1, (8, 5))
plt.plot(ds, us, label = "U(d)", marker = "x")
plt.plot(x, one_per_x, linestyle = "--", label = r"$\frac{1}{d}$")
plt.xlabel("d", size = 14)
plt.ylabel("U(d)", size = 14)
plt.legend()
plt.show()
```



(iiii) I used about 6 hours for this exercise.