Prerequisite

Since this work is using alot of pre-computed weights and initalized training data should this notebook be the same folder as the accompanying .pkl files.

If you want to test the code can you run the following block to test if you have the necessarry python libraries. All of them should be installed if you are using most recent version of Anaconda.

In [3]:

```
import numpy as np
from copy import deepcopy as copy
import matplotlib.pyplot as plt
import os
import pickle
```

TMA4215 Numerical Mathematics - Project 2

Implementation of Neural Network with ResNet-Architecture.

Table of Contents with links can be found at the bottom of the project.

Importing Training data

In [4]:

```
import numpy as np
import os


def import_batches():
    n_batches = 50

    data_prefix = "datalist_batch_"
    data_path = os.path.join( os.path.join(os.getcwd()), "project_2_trajectories")

    batches = {}

    for i in range(n_batches):
        # assemble track import path
        batch_path = os.path.join(data_path, data_prefix + str(i) + ".csv")
        batch_data = np.loadtxt(batch_path, delimiter=',', skiprows=1)

        # np.newaxis is adding a dimension such that (I,) -> (I, 1)
        batch = {}
        batch["t"] = batch_data[:, 0, np.newaxis]
        batch["Y_q"] = batch_data[:, 1:4].T
        batch["Y_p"] = batch_data[:, 4:7].T
        batch["c_p"] = batch_data[:, 7, np.newaxis]
        batch["c_q"] = batch_data[:, 8, np.newaxis] # potential energy

        batches[i] = batch

    return batches
```

1.

Implement functions for generating synthetic input data.

These are the functions used for generating synthetic data.

$$f_1(y) = \frac{1}{2}y_1^2 + \frac{1}{2}y_2^2$$
$$f_2(y) = \frac{1}{2}y^2$$
$$f_3(y) = 1 - cos(y)$$
$$f_4(y) = -\frac{1}{||y||}$$

In [5]:

```
def f_1(y):
    return 0.5*y[0]**2 + 0.5*y[1]**2

def f_2(y):
    return 0.5* np.square(y)

def f_3(y):
    return 1 - np.cos(y)

def f_4(y):
    return -1/np.sqrt(y[0]**2 +  y[1]**2 )


def generate_synthetic_batches(I,func = "2sqr", low=None, high=None):

    batch = {}

    if func == "2sqr":

        d_0 = 2
        if (high==None) and (low==None):
            high=2
            low=-2
        batch["Y"] = np.random.uniform(high, low, size=(d_0,I) )
        batch["c"] = f_1(batch["Y"])
        batch["c"] = batch["c"][:, np.newaxis]

        ct = f_1(batch["Y"] )
        return batch

    elif func == "1sqr":
        d_0 = 1

        if (high==None) and (low==None):
            high=2
            low=-2

        batch["Y"] = np.random.uniform(high, low, size=(d_0,I) )
        batch["c"] = f_2(batch["Y"] )
        batch["c"] = batch["c"].T

        return batch

    elif func == "1cos":
        d_0 = 1

        if (high==None) and (low==None):
            high=np.pi/3
            low=-np.pi/3

        batch["Y"] = np.random.uniform(high, low, size=(d_0,I) )
        batch["c"] = f_3(batch["Y"] )
        batch["c"] = batch["c"].T
        return batch

    elif func == "2norm-1":
        if (high==None) and (low==None):
            high=2
            low=-2
```

```
        d_0 = 2
        batch["Y"] = np.random.uniform(high, low, size=(d_0,I))

        for y in batch["Y"].T:
            if (np.all(y == 0)):
                y = np.array([0.1,0.1])

        batch["c"] = f_4(batch["Y"])
        batch["c"] = batch["c"].T
        batch["c"] = batch["c"][:, np.newaxis]

        return batch


    else:
        raise Exception("Not axeped func")
```

2.

Implement the neural network for training approximation of Hamiltonian function

The proposed model is formulated as

$$\tilde{F}(Y;\theta) = \eta((Z^{(K)})^T w + \mu \mathbf{1})$$

where for $k = 0, 1, 2, \ldots, K-1$

$$Z^{(k+1)} = Z^{(k)} + h\sigma(W_k Z^{(k)} + b_k),$$
$$Z^{(0)} = \hat{I}Y.$$

We have that $Y \in \mathbb{R}^{d_0 \times I}$ and $Z^{(k+1)}\mathbb{R}^{d \times I}$ where $d_0 \leq d$

We use the matrix $\hat{I}$ to embed Y into $\mathbb{R}^{d \times I}$

$$\hat{I} = \begin{bmatrix} Id_{d_0 \times d_0} \\ \mathbf{0} \end{bmatrix}$$

In [6]:

```
def F_tilde(Y, th, d_0, d, K, h):

    Z = {}
    Ihat = np.identity(d)[:,:d_0]
    Z[0] = Ihat@Y

    for k in range(K):
        Z_hat = th["W"][k]@Z[k]+th["b"][k]
        Z[k+1] = Z[k] + h*sigma(Z_hat, False)

    Upsilon = eta(Z[K].T@th["w"]+th["mu"])

    return Z, Upsilon


def initialize_weights(d_0, d, K):
    th = {}

    th["W"] = np.zeros((K, d, d))
    th["b"] = np.zeros((K, d, 1))

    for i in range(K):
        th["W"][i] = np.identity(d)
        th["b"][i] = np.zeros((d, 1))

    th["w"] = np.ones((d, 1 ))
    th["mu"] = np.zeros((1, 1))

    return th



def sigma(x, derivative=False):
    if (derivative):
        return 1 / np.cosh(x)**2
    return np.tanh(x)

def eta(x, derivative=False, identity=False):
    if identity==True:
        if (derivative):
            return np.ones(x.shape)
        return x
    else:
        if (derivative):
            return 0.25*(np.cosh(0.5*x) )**(-2)
        return 0.5*np.tanh(0.5*x) + 0.5
```

The objective function is on the form

$$J(\theta) = \frac{1}{2}\|\tilde{F}(Y;\theta) - c\|^2$$

where $\theta = \{W_0, \ldots, W_{K-1}, b_0, \ldots, b_{K-1}, w, \mu\}$, and c is the given data.

To be able to optimize the parameters θ we need the gradient of $\nabla J(\theta)$. The elements of the gradient corresponding to μ and w are,

$$\begin{aligned}
\frac{\partial J}{\partial \mu} &= \eta'((Z^{(K)})^T w + \mu \mathbf{1})^T (\tilde{F}(Y;\theta) - c) \\
\frac{\partial J}{\partial w} &= Z^{(K)} \left[(\tilde{F}(Y;\theta) - c) \odot \eta'((Z^{(K)})^T w + \mu \right],
\end{aligned}$$

and similarly for W_k and b_k,

$$\begin{aligned}
P^{(K)} &= w \cdot \left[(\tilde{F}(Y;\theta) - c) \odot \eta' \left((Z^{(K)})^T w + \mu \mathbf{1} \right) \right]^T \\
P^{(k-1)} &= P^{(k)} + h W_{k-1}^T \cdot \left[\sigma' \left(W_{k-1} Z^{(k-1)} + b_{k-1} \right) \odot P^{(k)} \right] \\
\frac{\partial J}{\partial W_k} &= h \left[P^{(k+1)} \odot \sigma'(W_k Z^{(k)} + b_k) \right] (Z^{(k)})^T \\
\frac{\partial J}{\partial b_k} &= h \left[P^{(k+1)} \odot \sigma'(W_k Z^{(k)} + b_k) \right] \cdot \mathbf{1}.
\end{aligned}$$

In [7]:

```
def J_func(Upsilon, c):
    return 0.5*np.linalg.norm(c - Upsilon)**2


def dJ_func(c, Y, th, d_0, d, K, h):
    Z, Upsilon = F_tilde(Y, th, d_0, d, K, h)
    I = Upsilon.shape[0]

    etahat = eta(Z[K].T@th["w"] + th["mu"]*np.ones(( I, 1)), derivative=True )

    P = np.zeros(( K+1, d, I))

    P[K] = np.outer(th["w"], ( (Upsilon - c)* etahat).T)

    dJ_mu = etahat.T @(Upsilon - c)

    dJ_w = Z[K] @ ((Upsilon - c) * etahat)

    for k in range(K, 0, -1):
        P[k-1] = P[k] + h*th["W"][k-1].T @ (sigma(th["W"][k-1]@Z[k-1]+np.outer(th["b"][
k-1],np.ones(I)), True) * P[k])

    dJ_W = np.zeros((K, d, d))
    dJ_b = np.zeros((K, d, 1))

    for k in range(K):
        dsigma = sigma(th["W"][k]@Z[k]+np.outer(th["b"][k],np.ones(I)),True)

        dJ_W[k] = h*(P[k+1]*dsigma) @ Z[k].T
        dJ_b[k] = (h*(P[k+1]*dsigma) @ np.ones(I))[:,np.newaxis]
    dJ = {}
    dJ["w"], dJ["mu"], dJ["W"], dJ["b"] = dJ_w, dJ_mu, dJ_W, dJ_b
    return dJ
```

To make sure our training data is inside the range of the hypothesis function η we use a linear scaling funtion to scale the data. The scaling function is given by

$$\hat{x} = \frac{(b-x)\alpha + (x-a)\beta}{b-a},$$

and the inverse function,

$$x = \frac{(\hat{x}+\alpha)b - (\hat{x}-\beta)a}{\beta-\alpha},$$

where $b = \max_{i,j} x$ and $a = \min_{i,j} x$ such that all values in $[a, b] \mapsto [\alpha, \beta]$. Since the range of our η is $[0, 1]$ we choose $\beta = 1$ and $\alpha = 0$. Later in the project will we also have use for inverted scaling without shift, this is in the context of finding the numerical gradients.

$$x = \frac{\hat{x}(b-a)}{\beta-\alpha}$$

In [8]:

```
def scale(x, alpha=0, beta=1, returnParameters = False):

    a = np.min(x)
    b = np.max(x)

    if returnParameters:
        return alpha, beta, a, b

    else:
        def  invscale(x):
            return ((x + alpha)*b - (x - beta)*a) / (beta-alpha)

        return ( (b - x)*alpha + (x - a)*beta)/(b - a), invscale

def invscaleparameter(x, alpha, beta, a, b):
    return ((x+alpha)*b - (x-beta)*a) / (beta-alpha)

def invscaleparameter_no_shift(x, alpha, beta, a, b):
    return x*(b-a)/(beta-alpha)
```

Optimization Algorithms

We will evaluate two optimization algorithms in the project.

Gradient Descent Algorithm

$$\theta_{i+1} = \theta_{i+1} - \tau \nabla J(\theta^{(r)})$$

Adams Algorithm

```
beta_1, beta_2 = 0.9, 0.999
alpha = 0.01
epsilon = 10^(-8)

v_0, m_0 = 0, 0

while not converged:
    g = dJ(th)
    m = beta_1 m + ( 1- beta_1)g_j
    v = beta_2 v + (1 - beta_2)(g*g)
    m_hat = m/(1 - beta_1^j)
    v_hat = v/(1 - beta_2^j)
    theta = theta - alpha m_hat/ ( v_hat^(0.5) + epsilon )
```

In [9]:

```
def gradientDesent(K, th, dJ, tau):

    th["mu"] = th["mu"] - tau*dJ["mu"]
    th["w"] = th["w"] - tau*dJ["w"]

    th["W"] = th["W"] -  tau*dJ["W"]
    th["b"] = th["b"] -  tau*dJ["b"]
    return th


def adam_algebra(th, dJ, v, m, key, j, alpha =10**(-5)):
        beta_1, beta_2 =  0.9, 0.999
        epsilon =  10**(-8)

        g = dJ[key]
        m[key] = beta_1*m[key] + (1- beta_1)*g
        v[key] = beta_2*v[key] + (1 - beta_2)*(g*g)
        mhat = m[key]/(1 - beta_1**(j+1))
        vhat = v[key]/(1 - beta_2**(j+1))
        th[key] -= alpha*mhat/(np.square(vhat) + epsilon)
        return th, v, m
```

Training Procedure The inputs of the training function are the given data c and Y, with coresponding dimention d_0, aswell as the preinitialized weights θ, the number of layers K, the dimention of the inner layers d, and h.

We also set the sensitivity parameters for the Gradient desent and Adam methods, τ and α.

In [10]:

```
def train(c, d, d_0, K, h, Y, th, tau=0.0005, max_it=60, print_it=False, method="gd", a
lpha =7.5*10**(-5)):
    # compute Zk
    err = np.inf
    tol = 0.01

    itr = 0
    Z, Upsilon = F_tilde(Y, th, d_0, d, K, h)
    JJ = np.zeros(max_it+1)
    err = J_func(Upsilon,c)

    JJ[0] = err

    # Adam parameters
    m = {}
    m["mu"] = np.zeros(th["mu"].shape)
    m["w"] = np.zeros(th["w"].shape)
    m["W"] = np.zeros(th["W"].shape)
    m["b"] = np.zeros(th["b"].shape)
    v = copy(m)

    while (itr < max_it ):

        Z, Upsilon = F_tilde(Y, th, d_0, d, K, h)

        if (method=="gd"):
            dJ = dJ_func(c, Y, th, d_0, d, K, h)
            th = gradientDesent(K, th, dJ, tau)

        elif (method=="adam"):
            j = itr

            dJ = dJ_func(c, Y, th, d_0, d, K, h)

            th, v, m = adam_algebra(th, dJ, v, m, "mu", j, alpha)
            th, v, m = adam_algebra(th, dJ, v, m, "w", j, alpha)
            th, v, m = adam_algebra(th, dJ, v, m, "W", j, alpha)
            th, v, m = adam_algebra(th, dJ, v, m, "b", j, alpha)

        else:
            print("No optimization method")

        err = J_func(Upsilon, c)

        JJ[itr+1] = err

        itr += 1

        if(itr%600 == 0) and (print_it == True):
            print(itr,err)

    return JJ , th
```

Stochastic Gradient

Since each data point is sent through the network indepentently we can, therefore we can exchange the full data set Y with a smaller subset $\hat{Y}$. Using this fact can we sweep through the dataset faster by random shuffling and slizing the entire dataset.

In [11]:

```
def stocgradient(c, d, d_0, K, h, Y, th, tau, max_it , bsize, sifts = 100):

    JJ = np.array([])
    I = Y.shape[1]
    totitr = int(I/bsize)
    for siftnum in range(sifts):

        indexes = np.array(range(I))
        np.random.shuffle(indexes)

        Z, Upsilon = F_tilde(Y, th, d_0, d, K, h)
        err = J_func(Upsilon, c)
        JJ = np.append(JJ, err)

        while len(indexes) > 0:

            if len(indexes) >= bsize:
                bsliceI = indexes[:bsize]
                Yslice = Y[:,bsliceI]
                cslice = c[bsliceI]

                dJJ, th = train(cslice, d, d_0, K, h, Yslice, th, tau, max_it)
                indexes = indexes[bsize:]

            else:
                Yslice = Y[:,indexes]
                cslice = c[indexes]

                dJJ, th = train(cslice, d, d_0, K, h, Yslice, th, tau, max_it)
                indexes = []

    return JJ, th
```

2. a)

Test the model by using the suggested functions

In [12]:

```
def train_func(func):

    I = 8000
    K = 20
    h = 0.1
    sifts = 400
    if func == "2norm-1":
        sifts = 2400
    Ihat = 320
    tau = 3/Ihat

    data = generate_synthetic_batches(I, func)

    Y =data["Y"]
    c = data["c"]
    sc , invc = scale(c)
    sparameters = scale(c,returnParameters = True)

    inv_file = open( func+"_inv.pkl", "wb")
    pickle.dump(sparameters, inv_file)
    inv_file.close()

    d_0 = Y.shape[0]
    d = d_0*2

    th = initialize_weights(d_0, d, K)

    JJ, th = stocgradient(sc, d, d_0, K, h, Y, th, tau, 1 , Ihat, sifts)

    plt.plot(JJ)
    plt.yscale("log")
    plt.title("Cost Function for " + func)
    plt.show()

    th_file = open(func + "_th.pkl", "wb")
    pickle.dump(th, th_file)
    th_file.close()



def test_func(func):

    numData = 2000

    K = 20
    h = 0.1


    if func == "1sqr":
        Y  = np.linspace(-2,2,numData)
        Y = Y[:,np.newaxis].T
        c = 1/2*Y**2
        c = c.T

    elif func == "1cos":
        Y = np.linspace(-np.pi/3,np.pi/3,numData)
        Y = Y[:,np.newaxis].T
        c = 1-np.cos(Y)
        c = c.T
```

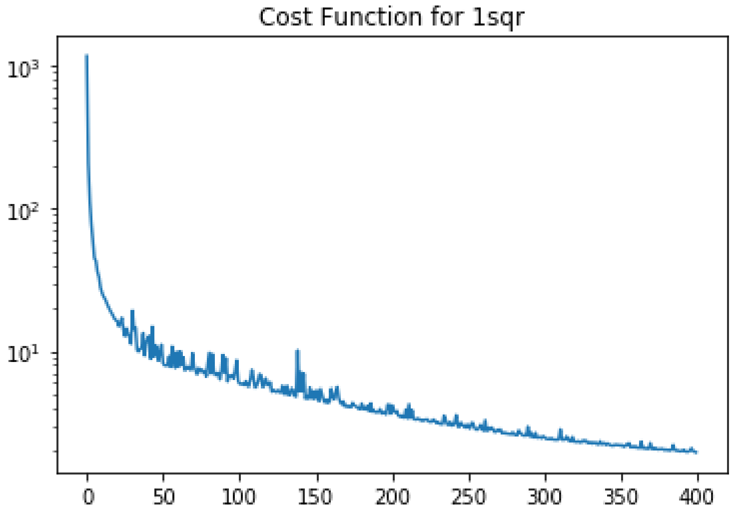
```
    elif func == "2sqr":
        x1 = np.linspace(-2,2,numData)
        Y = np.array([x1,-x1])
        c = 1/2*(Y[0]**2 + Y[1]**2)
        c = c.T

    elif func == "2norm-1":
        x2hat1 = np.linspace(-2,-1/4,int(numData/2))
        x2hat2 = np.linspace(1/4,2,int(numData/2))
        x2 = np.append(x2hat1,x2hat2)
        Y = np.array([x2,-x2])
        c = -1/np.sqrt(Y[0]**2 + Y[1]**2)
        c = c.T

    else:
        raise Exception("No func")


    d_0 = Y.shape[0]
    d = d_0*2


    inv_file = open( func+"_inv.pkl", "rb")
    inv = pickle.load(inv_file)
    inv_file.close()

    th_file = open(func + "_th.pkl", "rb")
    th = pickle.load(th_file)
    th_file.close()
    z, yhat = F_tilde(Y, th, d_0, d, K, h)

    y = invscaleparameter(yhat, inv[0], inv[1], inv[2], inv[3])
    if d_0 == 1:
        plt.plot(Y.T, y, label ="Estimated function")
        plt.plot(Y.T,c, label ="Analytical function")
        plt.title("Comparison for " + func)
        plt.legend()
        plt.show()
    else:
        plt.plot(y, label ="Estimated function")
        plt.plot(c, label ="Analytical function")
        plt.title("Comparison for " + func)
        plt.legend()
        plt.show()
```

Running the train functions takes approximately 4 min.

In [41]:

```
print("Train model for 1sqr Function")
train_func(func="1sqr")
```

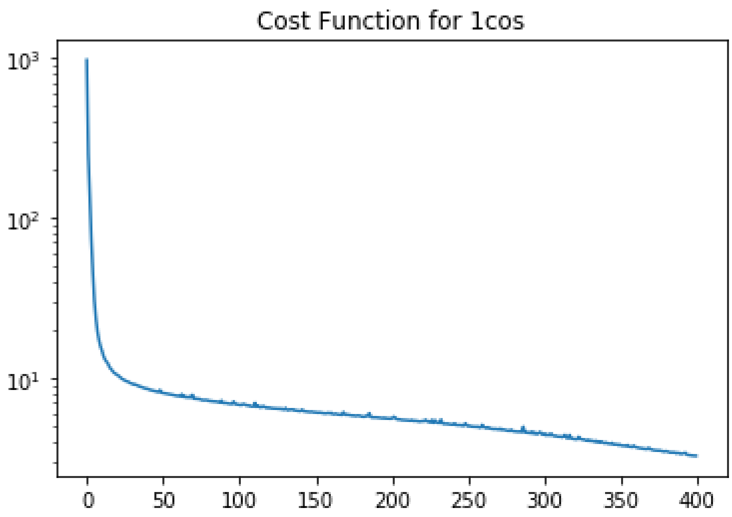
Train model for 1sqr Function



In [45]:

```
print("Train model for cos Function")
train_func(func="1cos")
```

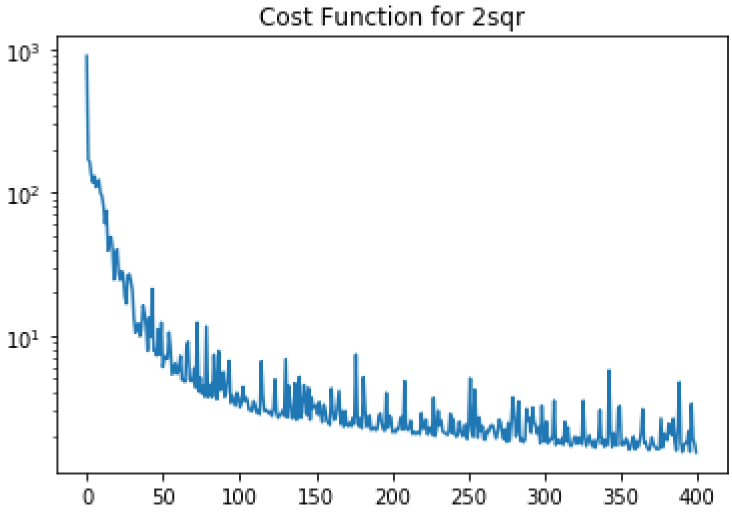
Train model for cos Function



In [46]:

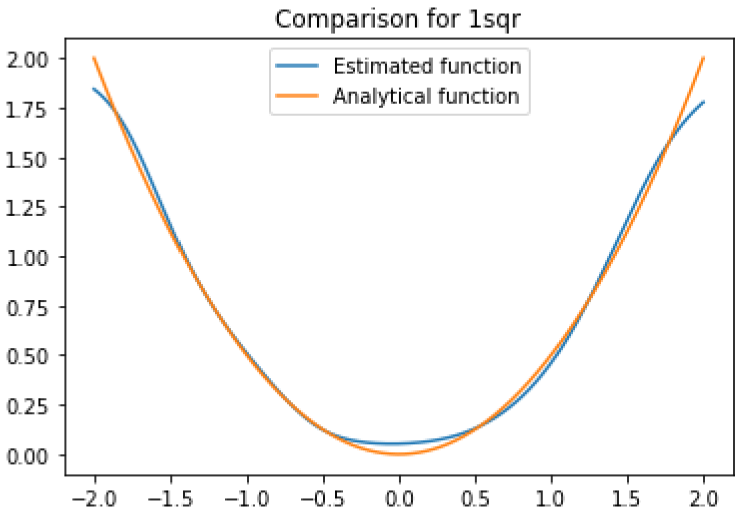
```
print("Train model for 2sqr Function")
train_func(func="2sqr")
```

```
Train model for 2sqr Function
```



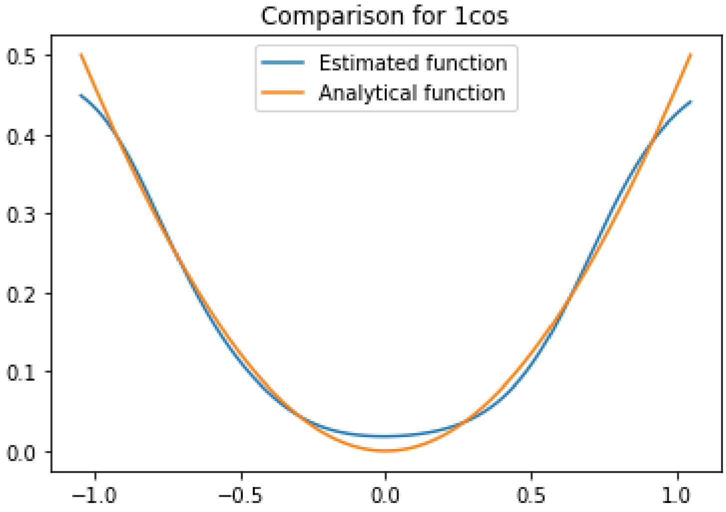
In [48]:

```
test_func(func="1sqr")
```



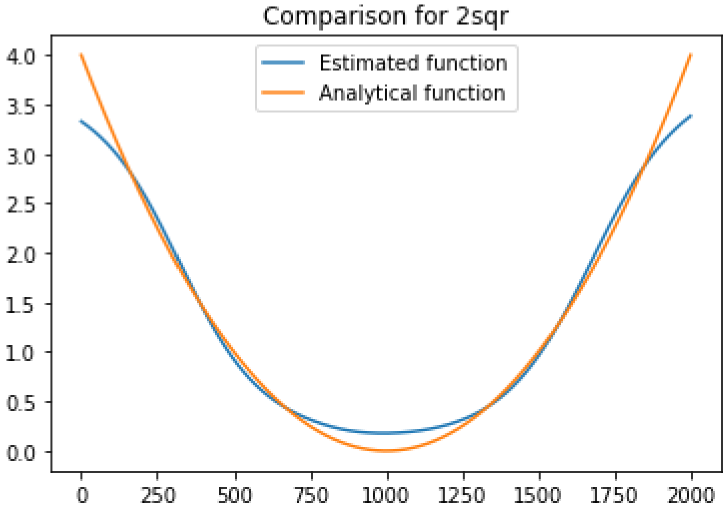
In [49]:

```
test_func(func="1cos")
```



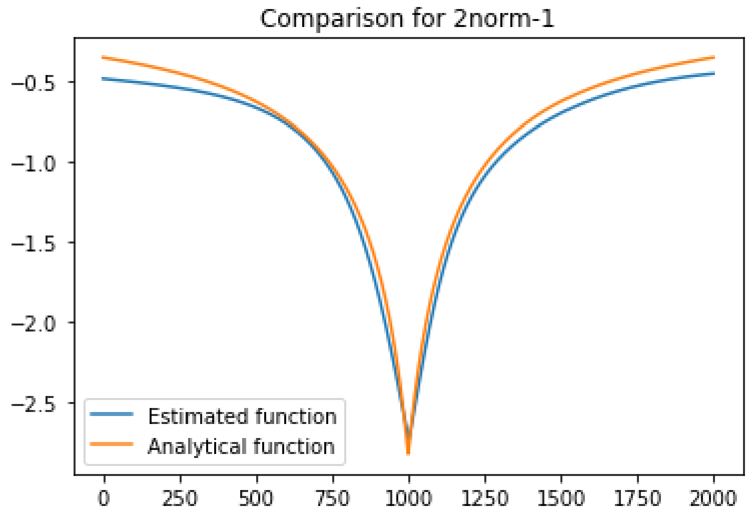
In [50]:

```
test_func(func="2sqr")
```



In [73]:

```
test_func(func="2norm-1")
```



2. b)

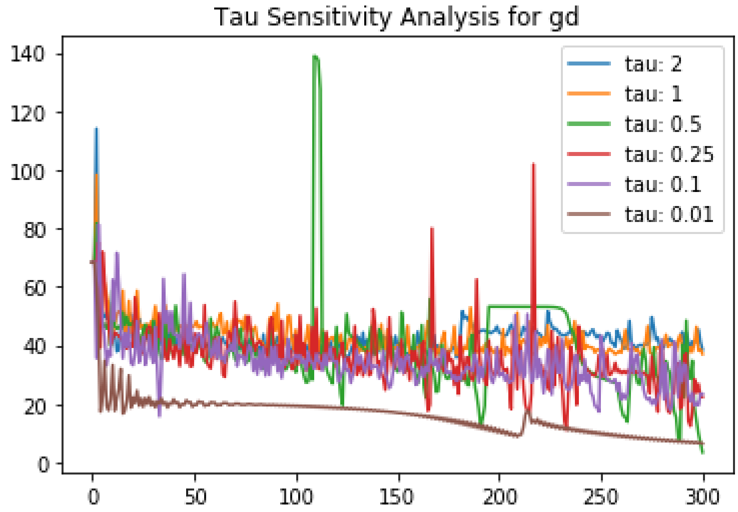
Investigate systematically what are optimal choices for K, τ , d, h and any other choices you need to make. Balance performance in the generalisation phase with time consumption of training

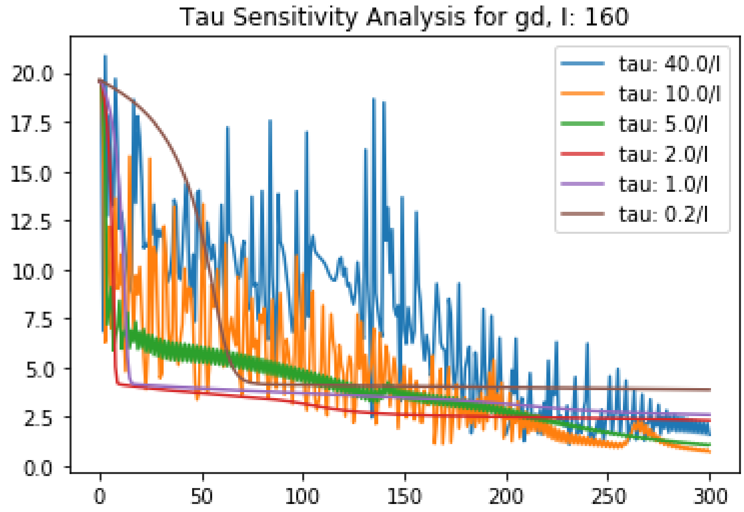
Tau Sensitivity Running this code takes ~10min.

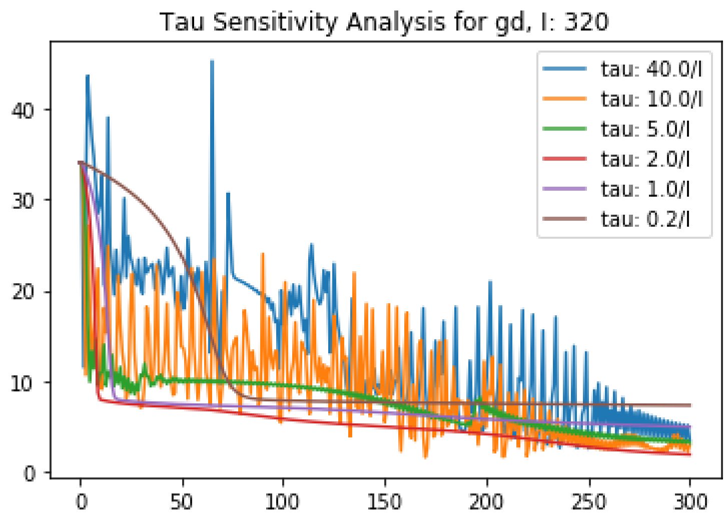
In [53]:

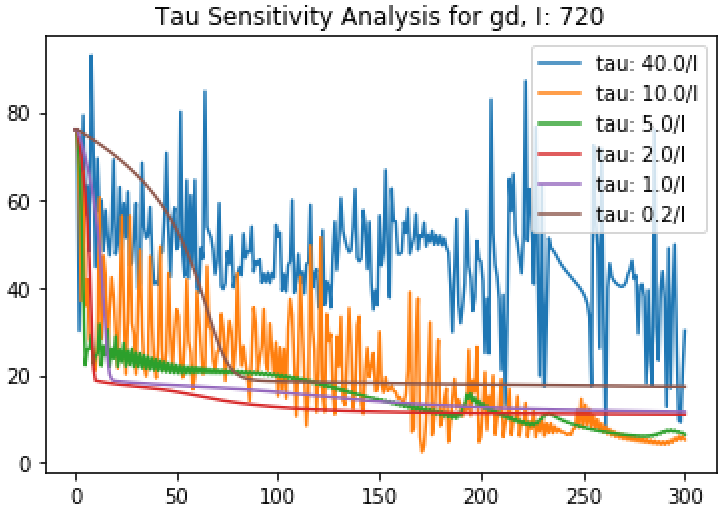
```
def tau_sensitivity(method="gd"):
    # Chosen default values
    K = 20
    h = 0.1
    d_0 = 2
    d = 4
    I = 600
    max_it = 300
    tauxI = 2
    b = generate_synthetic_batches(I)

    c, inv = scale(b["c"])
    Y = b["Y"]
    #Y = scale(b["Y"])
    d_0 = Y.shape[0]

    var = [ 2, 1, 0.5, 0.25, 0.1, 0.01]
    it = np.arange(0,max_it+1)

    for i in range(len(var)):
        tau = var[i]
        th = initialize_weights(d_0, d, K)
        JJ,th = train(c, d, d_0, K, h, Y, th, tau=tau, max_it=max_it, method=method)
        plt.plot(it, JJ, label="tau: "+ str(tau))

    plt.title("Tau Sensitivity Analysis for " + method)
    plt.legend()
    plt.show()

def tauI_sensitivity(I, method="gd"):

    K = 20
    h = 0.1
    d_0 = 2
    d = 4
    max_it = 300
    tauxI = 2

    b = generate_synthetic_batches(I)

    c, inv = scale(b["c"])
    Y = b["Y"]
    #Y = scale(b["Y"])
    d_0 = Y.shape[0]

    var = np.array([ 40, 10, 5, 2, 1, 0.2])/I
    it = np.arange(0,max_it+1)

    for i in range(len(var)):
        tau = var[i]
        th = initialize_weights(d_0, d, K)
        JJ,th = train(c, d, d_0, K, h, Y, th, tau=tau, max_it=max_it, method=method)
        plt.plot(it, JJ, label="tau: "+ str(tau*I) + "/I")

    plt.title("Tau Sensitivity Analysis for " + method + ", I: "+str(I))
    plt.legend()
    plt.show()

tau_sensitivity(method="gd")
```

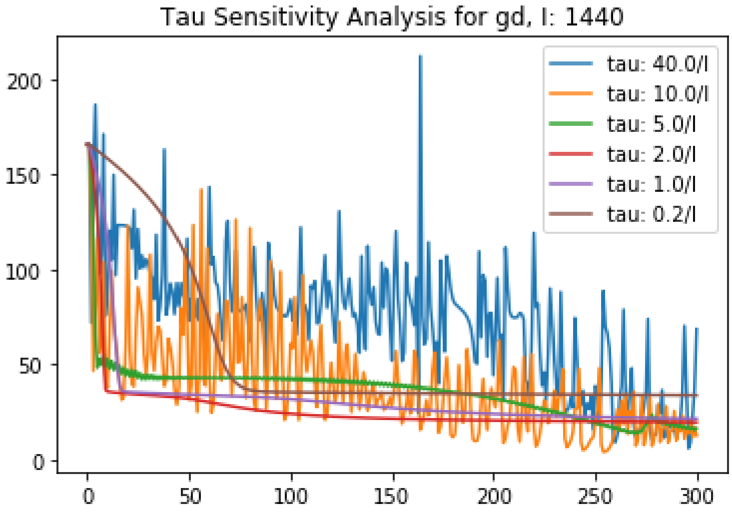
```
I = [160, 320, 720, 1440, 2880, 5760]
for i in I:
    tauI_sensitivity(i)
```

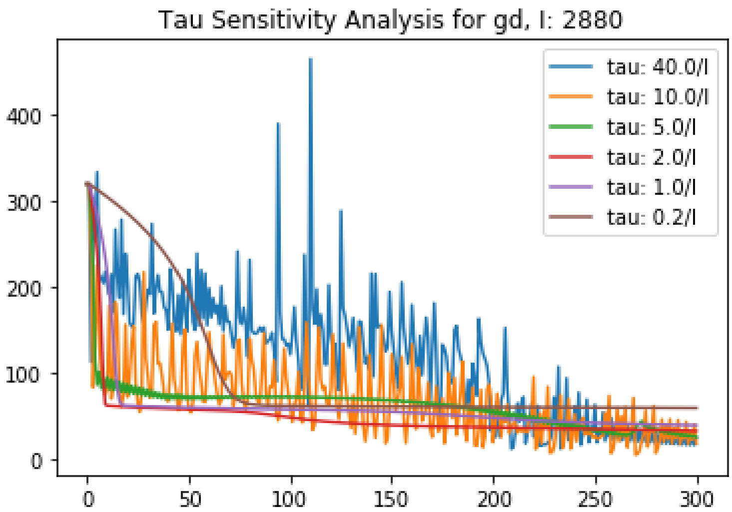

Tau Sensitivity Analysis for gd
tau: 2
tau: 1
tau: 0.5
tau: 0.25
tau: 0.1
tau: 0.01
0 20 40 60 80 100 120 140
0 50 100 150 200 250 300

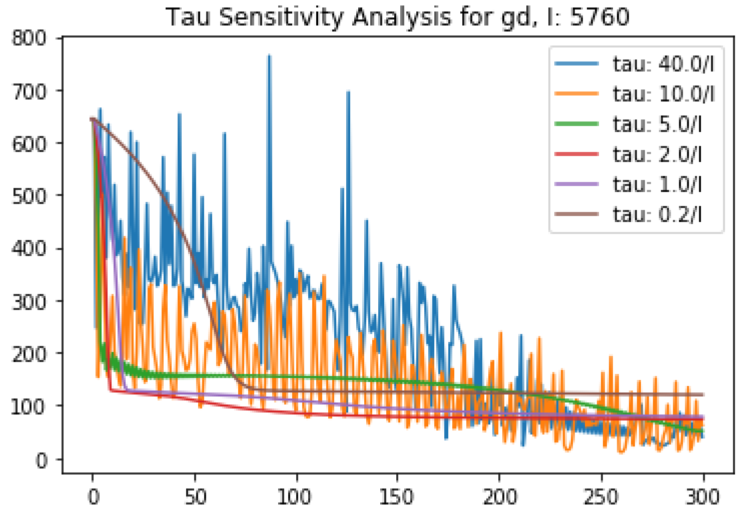

Tau Sensitivity Analysis for gd, l: 160
tau: 40.0/l
tau: 10.0/l
tau: 5.0/l
tau: 2.0/l
tau: 1.0/l
tau: 0.2/l
0.0 2.5 5.0 7.5 10.0 12.5 15.0 17.5 20.0
0 50 100 150 200 250 300


Tau Sensitivity Analysis for gd, l: 320
tau: 40.0/l
tau: 10.0/l
tau: 5.0/l
tau: 2.0/l
tau: 1.0/l
tau: 0.2/l
0 10 20 30 40
0 50 100 150 200 250 300


Tau Sensitivity Analysis for gd, l: 720
tau: 40.0/l
tau: 10.0/l
tau: 5.0/l
tau: 2.0/l
tau: 1.0/l
tau: 0.2/l


Tau Sensitivity Analysis for gd, l: 1440
tau: 40.0/l
tau: 10.0/l
tau: 5.0/l
tau: 2.0/l
tau: 1.0/l
tau: 0.2/l


Tau Sensitivity Analysis for gd, l: 2880
tau: 40.0/l
tau: 10.0/l
tau: 5.0/l
tau: 2.0/l
tau: 1.0/l
tau: 0.2/l

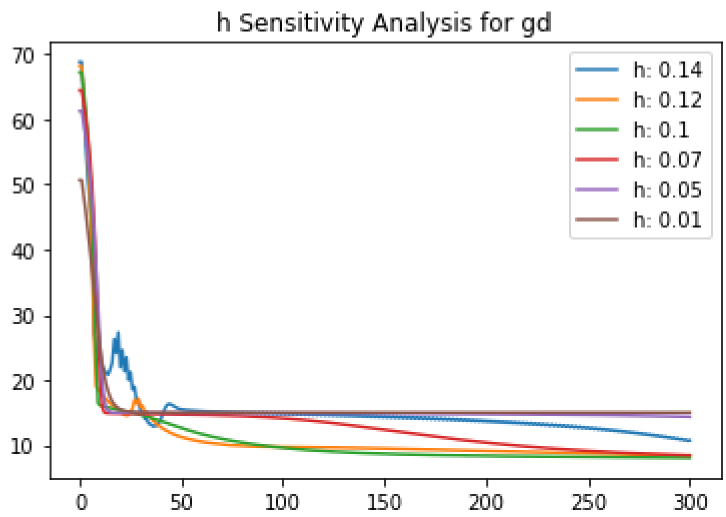


We discover that the rate and stability of the convergence for the gradient decent method for a given τ is dependent on the number of data points I. We also discover that by using a τ that is inversely proportional with I, the convergence is similar for different choices of I. By looking at the analysis we find that $\tau = 2/I$ has approximately the best rate and stability.

h Sensitivity

In [14]:

```
def h_sensitivity(method="gd"):
    K = 20
    d_0 = 2
    d = 4
    I = 600
    max_it = 300
    tauxI = 2

    b = generate_synthetic_batches(I)

    c, inv = scale(b["c"])
    Y = b["Y"]
    #Y = scale(b["Y"])
    d_0 = Y.shape[0]

    var = var = [ 0.14, 0.12, 0.1, 0.07, 0.05, 0.01]
    it = np.arange(0,max_it+1)

    for i in range(len(var)):
        h = var[i]
        th = initialize_weights(d_0, d, K)
        JJ,th = train(c, d, d_0, K, h, Y, th, tauxI/I, max_it=max_it, method=method)
        plt.plot(it, JJ, label="h: "+ str(var[i]))

    plt.title("h Sensitivity Analysis for " + method)
    plt.legend()
    plt.show()

h_sensitivity(method="gd")
```

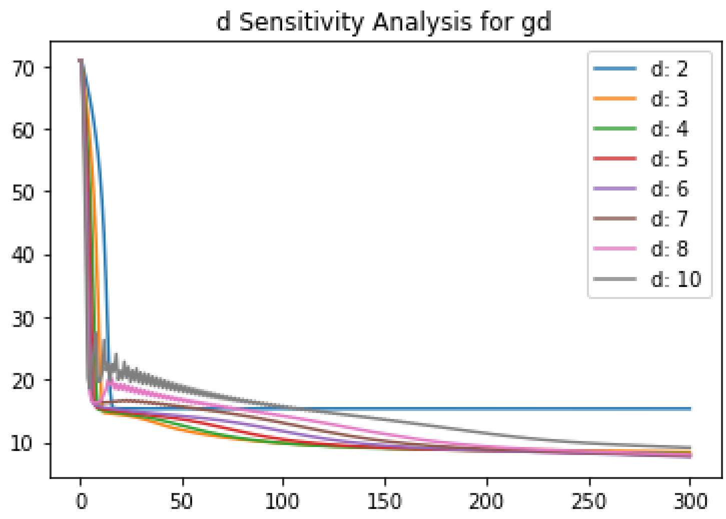


$h = 0.1$ seems to be a good choice.

d-sensitivity

In [15]:

```
def d_sensitivity(method="gd"):
    K = 20
    h = 0.1
    d_0 = 2
    I = 600
    max_it = 300
    tauxI = 2
    b = generate_synthetic_batches(I)

    c, inv = scale(b["c"])
    Y = b["Y"]
    #Y = scale(b["Y"])
    d_0 = Y.shape[0]

    var = var = [ 2, 3, 4, 5, 6, 7, 8, 10 ]
    it = np.arange(0,max_it+1)

    for i in range(len(var)):
        d = var[i]
        th = initialize_weights(d_0, d, K)
        JJ,th = train(c, d, d_0, K, h, Y, th, tauxI/I, max_it=max_it, method=method)
        plt.plot(it, JJ, label="d: "+ str(var[i]))

    plt.title("d Sensitivity Analysis for " + method)
    plt.legend()
    plt.show()
d_sensitivity(method="gd")
```

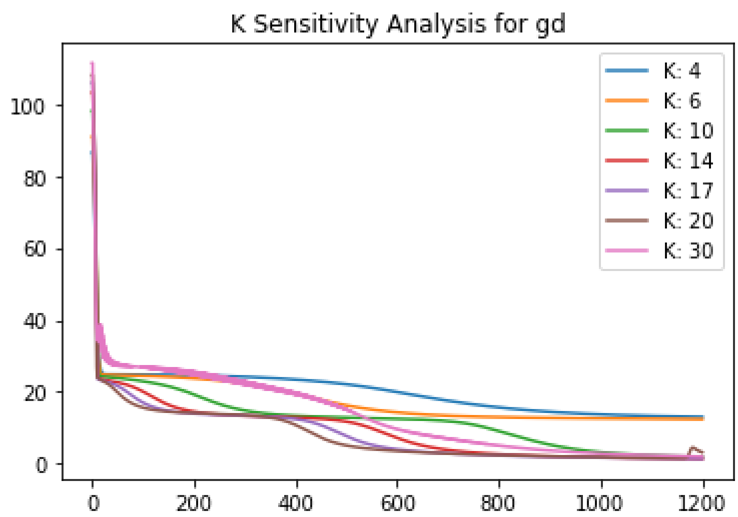


$d = 4$ seems to be a good choice.

K-Sensitivity

In [18]:

```
def K_sensitivity(method="gd"):
    h = 0.1
    d_0 = 2
    d = 4
    I = 1000
    max_it = 1200
    tauxI = 2

    b = generate_synthetic_batches(I)

    c, inv = scale(b["c"])
    Y = b["Y"]
    d_0 = Y.shape[0]

    var = [ 4, 6, 10, 14, 17, 20, 30]
    it = np.arange(0,max_it+1)

    for i in range(len(var)):
        K = var[i]
        th = initialize_weights(d_0, d, K)
        JJ,th = train(c, d, d_0, K, h, Y, th, tauxI/I, max_it, method)
        plt.plot(it, JJ, label="K: "+ str(var[i]))

    plt.title("K Sensitivity Analysis for " + method)
    plt.legend()
    plt.show()
K_sensitivity(method="gd")
```



$K = 20$ seems to be a good choice.

2. c)

Train the model for the case of data given (with unknown Hamiltonian function).

In [25]:

```
def train_unknown(pq):

    K = 20
    h = 0.1
    sifts = 10000
    Ihat = 2000
    tau = 2/Ihat

    batches = import_batches()
    batch1 = batches[0]
    antB = 40


    bigbatch = {}
    bigbatch["Y"] = np.array([[],[],[]])
    bigbatch["c"] = np.array([])

    for i in range(antB):
        batch = batches[i]
        bigbatch["Y"] = np.append(bigbatch["Y"],batch["Y_"+pq],1)
        bigbatch["c"] = np.append(bigbatch["c"],batch["c_"+pq])

    Y = bigbatch["Y"]
    c = bigbatch["c"][:,np.newaxis]

    sc,inv = scale(c)

    sparameters = scale(c,returnParameters = True)


    inv_file = open(pq + "_unknown_inv.pkl", "wb")
    pickle.dump(sparameters, inv_file)
    inv_file.close()

    d_0 = Y.shape[0]
    d = d_0*2



    th = initialize_weights(d_0, d, K)

    #JJ, th = stocgradient(sc, d, d_0, K, h, Y, th, tau, 1 , Ihat, sifts, True, pq + "_
unknown_w.pkl")
    JJ, th = stocgradient(sc, d, d_0, K, h, Y, th, tau, 1 , Ihat, sifts)

    plt.plot(JJ)
    plt.yscale("log")
    plt.show()


    th_file = open(pq + "_unknown_w.pkl", "wb")
    pickle.dump(th, th_file)
    th_file.close()

#train_unknown("p")
```

This code takes a very long time to run because it sifts through a large dataset very many times.

2. d)

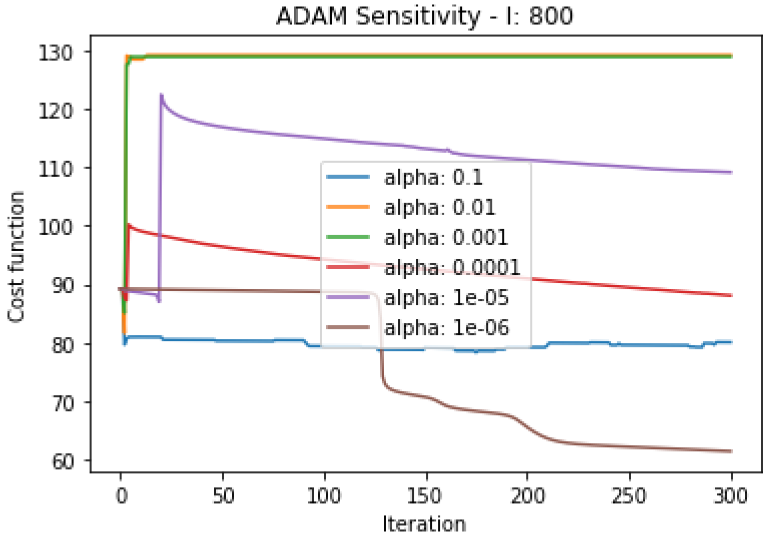
Try other alternatives for optimisation, such a the Adam method

In [30]:

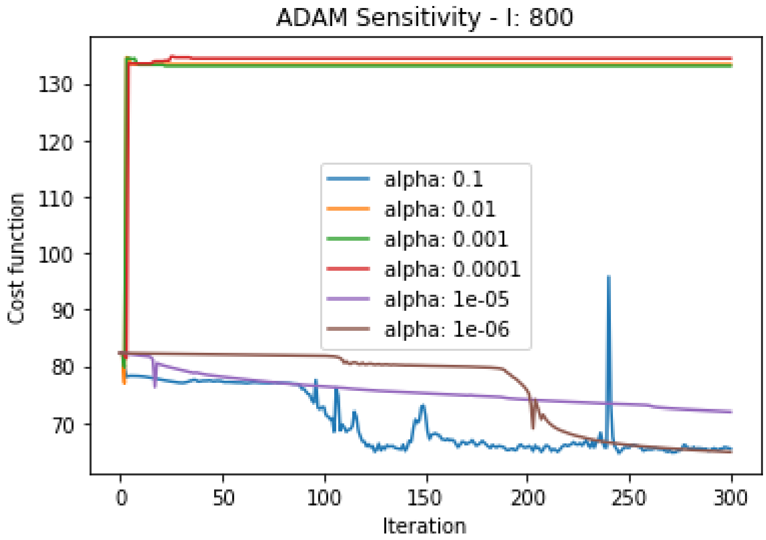
```
def alpha_sensitivity(method="adam"):

    K = 20
    h = 0.1
    d_0 = 2
    d = 4
    I = 800
    max_it = 300
    tauxI = 2

    b = generate_synthetic_batches(I)

    c, inv = scale(b["c"])
    Y = b["Y"]
    #Y = scale(b["Y"])
    d_0 = Y.shape[0]

    var = [0.75*10**-4, 0.5*10**-4, 0.35*10**-4, 0.75*10**-5, 0.5*10**-5, 0.25*10**-6]
    it = np.arange(0,max_it+1)

    for i in range(len(var)):
        alpha = var[i]
        th = initialize_weights(d_0, d, K)
        JJ,th = train(c, d, d_0, K, h, Y, th, tauxI/I, max_it=max_it, method=method, al
pha=alpha)
        plt.plot(it, JJ, label="alpha: "+ str(alpha))

    plt.title("Alpha Sensitivity Analysis for " + method)
    plt.legend()
    plt.show()

def alphaI_sensitivity(I, method="adam"):

    K = 20
    h = 0.1
    d_0 = 2
    d = 4
    I = 800
    max_it = 300
    tauxI = 2

    b = generate_synthetic_batches(I)

    c, inv = scale(b["c"])
    Y = b["Y"]
    d_0 = Y.shape[0]

    var = np.array( [10**-1, 10**-2, 10**-3, 10**-4, 10**-5, 10**-6 ])#/I
    it = np.arange(0,max_it+1)

    for i in range(len(var)):
        alpha = var[i]
        th = initialize_weights(d_0, d, K)
        JJ,th = train(c, d, d_0, K, h, Y, th, tauxI/I, max_it=max_it, method=method, al
pha=alpha)
        plt.plot(it, JJ, label="alpha: "+ str(alpha) )

    plt.title("ADAM Sensitivity - " + "I: "+str(I))
    plt.xlabel("Iteration")
```

```
    plt.ylabel("Cost function")
    plt.legend()
    plt.show()




I = [160, 320, 720, 1440, 2880, 5760]
for i in I:
    print("I: ", i)
    alphaI_sensitivity(i)
```

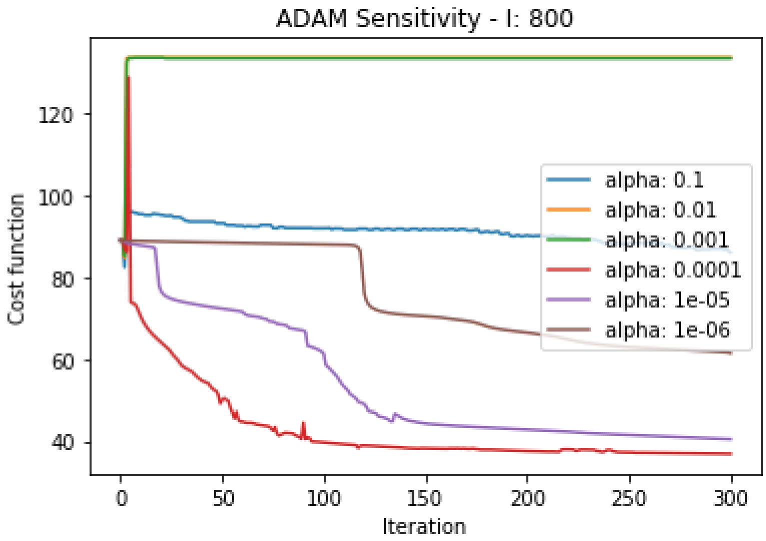
```
I:  160

C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:35: Runti
meWarning: overflow encountered in cosh
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:35: Runti
meWarning: overflow encountered in square
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:45: Runti
meWarning: overflow encountered in cosh
```



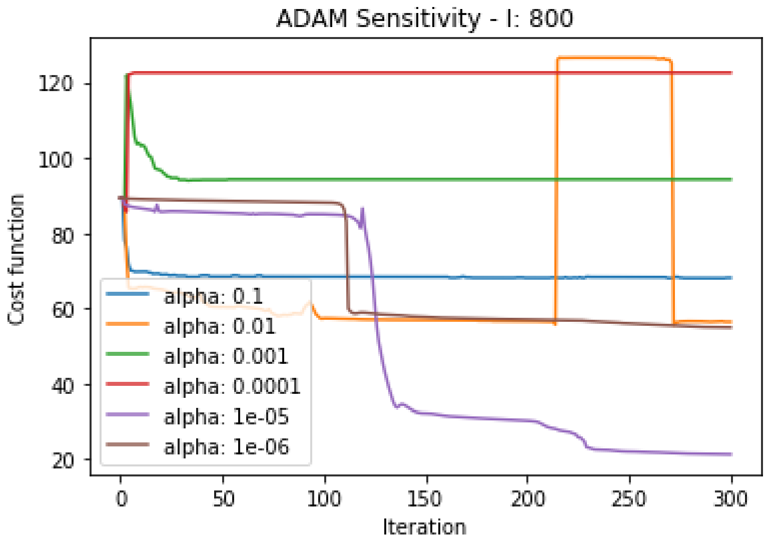

```
I:  320
```



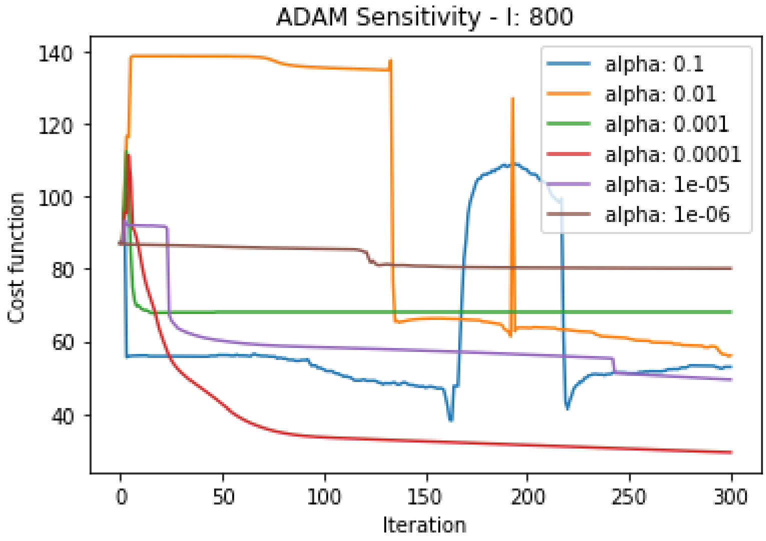
I: 720



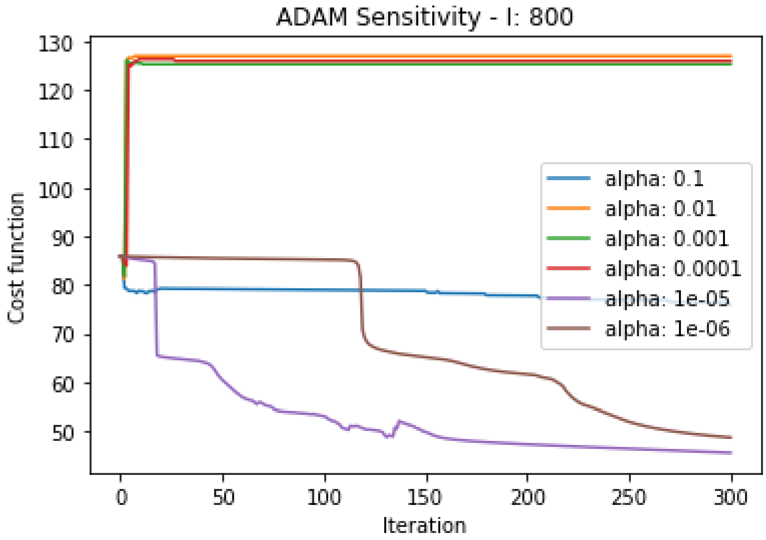
I: 1440



I: 2880




```
I:  5760
```



In general, we realized that alpha was the most critical parameter for the convergence rate. However, we also observed that the convergence was quite inconsistent depending on the number of datapoints I. It is easy to see that for a large portion of the chosen alpha-values the convergence stagnates before reaching any reasonable value for the objective function. By comparing to the convergence of gradient descent in previous analysis, we find that our implementation of gradient descent is superior to our implementation of Adams method.

2. e)

Make use of convergence plots for getting an indication of the efficiency of your choices

We used convergence plots to find what choices would be the most efficient in the earier analysis.

2. g/f)

Include an evaluation, mostly through experiments, on how well your trained model approximates the function you started with on the training data.

Do a similar evaluation on test data (that were not used in the training phase)

In [34]:

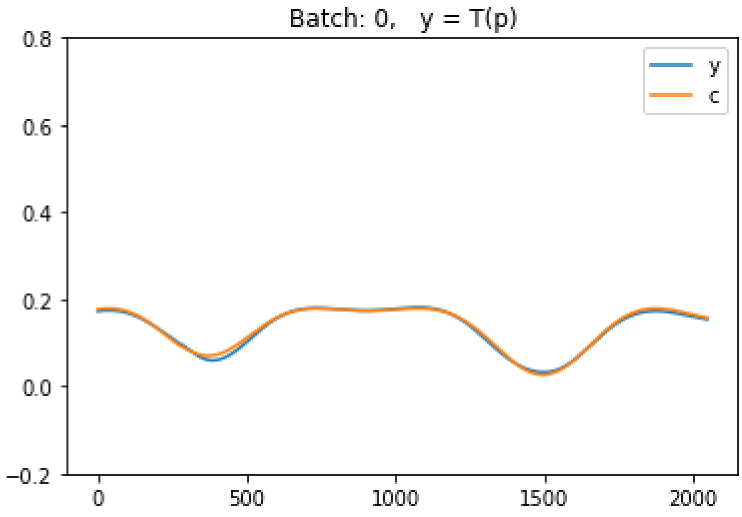
```
def test_unknown(pq, low = 0 , high = 1):

    K = 20
    h = 0.1


    inv_file = open( pq + "_unknown_inv.pkl", "rb")
    inv = pickle.load(inv_file)
    inv_file.close()

    w_file = open(pq + "_unknown_w.pkl", "rb")
    th = pickle.load(w_file)
    w_file.close()

    batches = import_batches()
    batch1 = batches[0]
    antB = 50

    Y = batch1["Y_q"]
    d_0 = Y.shape[0]
    d = d_0*2

    b_indexes = np.arange(low,high+1)
    if pq == "p":
        F = "T(p)"
    elif pq == "q":
        F = "V(q)"
    else:
        print("p or q")

    #for i in range(antB):
    for i in b_indexes:
        plt.title("Batch: " + str(i) + ",    y = " + F)
        testbatch = batches[i]

        tY = testbatch["Y_"+pq]

        z, yhat = F_tilde(tY, th, d_0, d, K, h)


        y = invscaleparameter(yhat, inv[0], inv[1], inv[2], inv[3])
        c = testbatch["c_"+pq]


        print("Mean error:", np.average(np.abs(y-c)))

        plt.plot(y,label ="y")
        plt.plot(c,label ="c")
        plt.ylim(-0.2,0.8)
        plt.legend()
        plt.show()
```

You can choose what batches you want to look at by selecting a range as the last two parameters in the test_unknown function, "p" corresponds to T(p), and "q" to V(q).

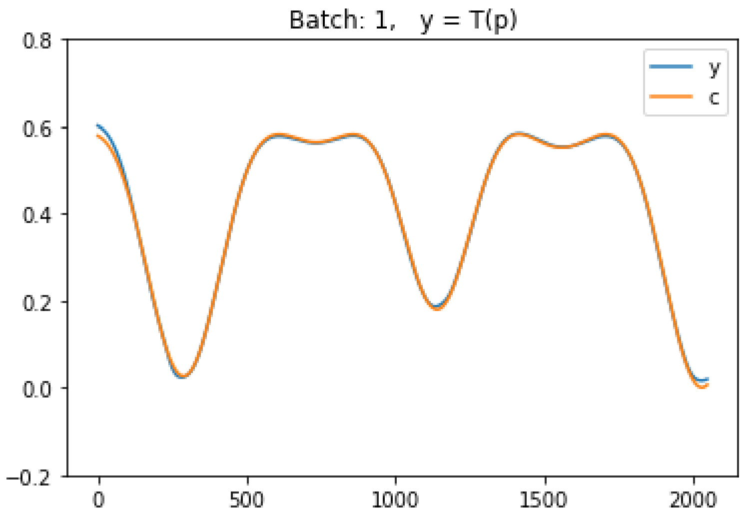
In [35]:

```
test_unknown("p",0,1)
test_unknown("q",0,1)
test_unknown("p",45,46)
test_unknown("q",45,46)
```

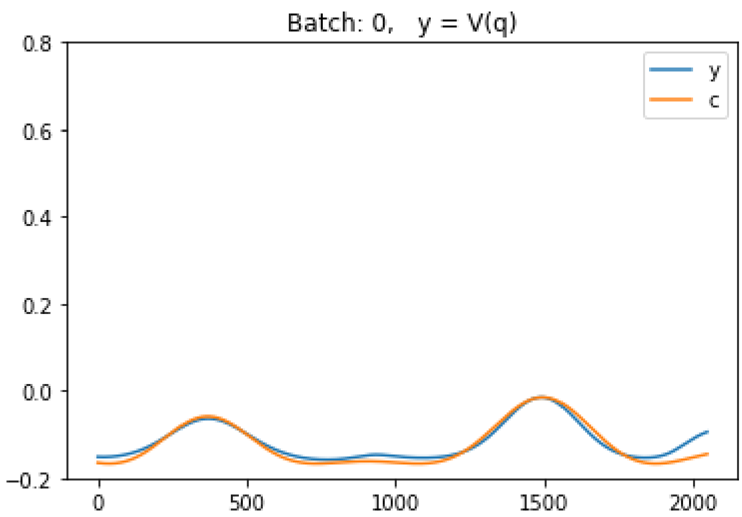
```
Mean error: 0.0035752891941011455
```



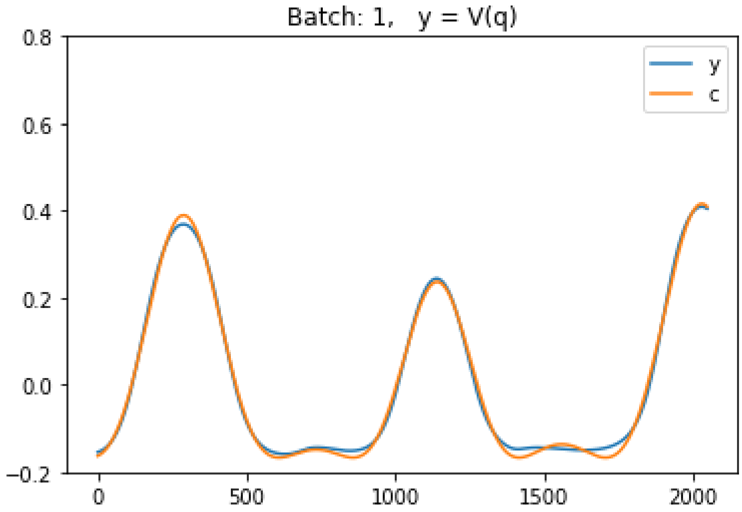

```
Mean error: 0.003605999106927585
```



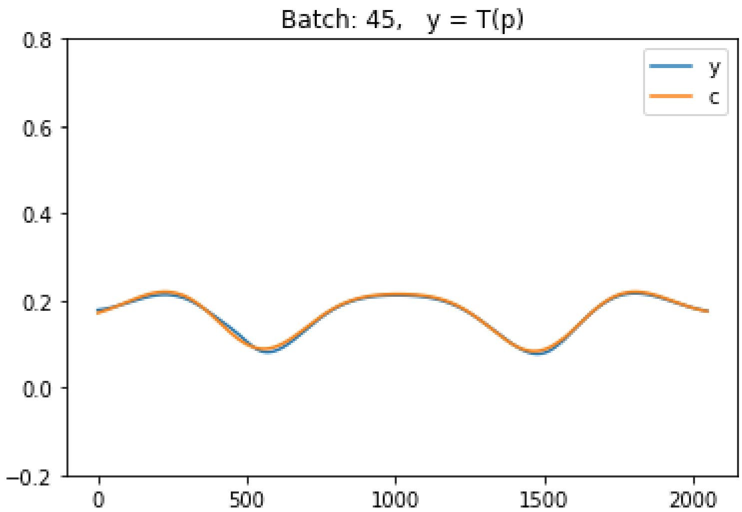

```
Mean error: 0.010534481038605248
```



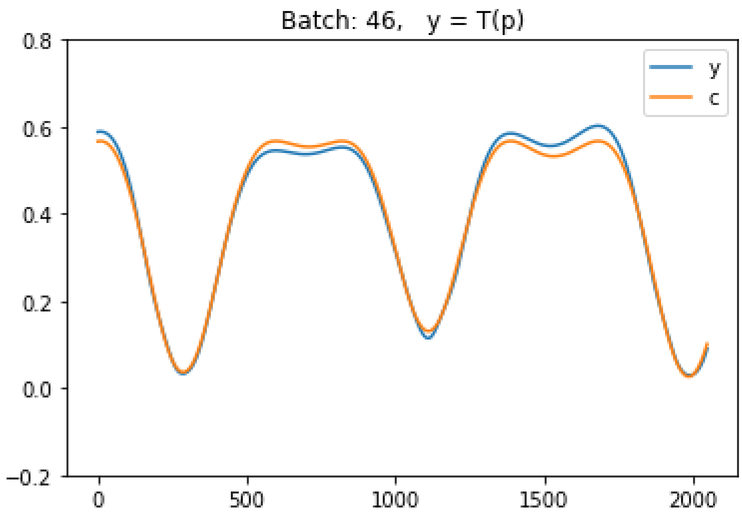

```
Mean error: 0.009004505360867445
```



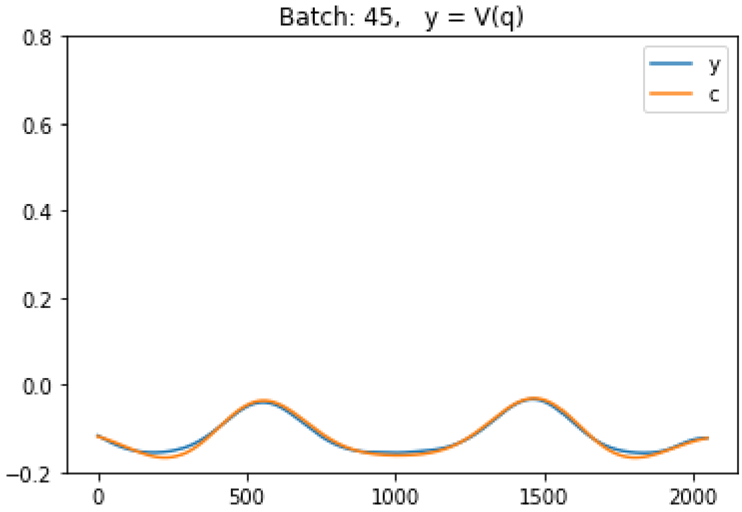
Mean error: 0.003488720078072171



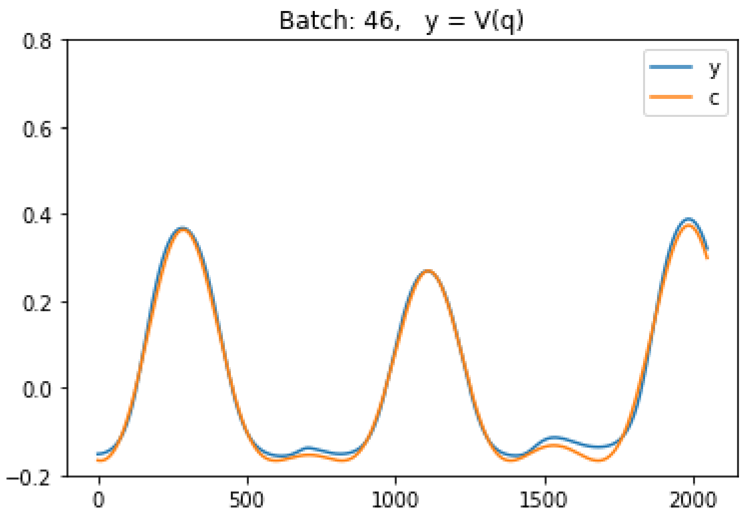
Mean error: 0.0137073346055613



Mean error: 0.0050932947488626056




```
Mean error: 0.011918062202387777
```



Here y is the numerical values and c is the analytical values. Looking at the plots we observe that the numerical vaues approximates the analytical ones quite closely. The training data was the batches from 0 to 39, we observe that the trained function generalized nicely to the test data (batches 40 to 49) that was not included in the training data.

3.

Derive the formulas for computing the gradient of the trained function

We want to derive the gradient,

$$(\nabla_y \tilde{F}(y))$$

.

From the supplementary hint note we have that

$$A_K = \nabla G(Z^K) = \eta'(w^T Z^{(K)} + \mu)w$$

For k = K, $K-1$, ..., 1, 0 we have that

$$\begin{aligned} A_{k-1} &= (D\Phi_{k-1}(Z^{k-1}))^T A_k \\ &= A_k + W_{k-1}^T(h\sigma'(W_{k-1}Z^{(k-1)} + b_k) \circ A_k). \end{aligned}$$

We can define the result as

$$\nabla_y \tilde{F} = A_K.$$

4.

Implement these formulas for computing the gradient

In [36]:

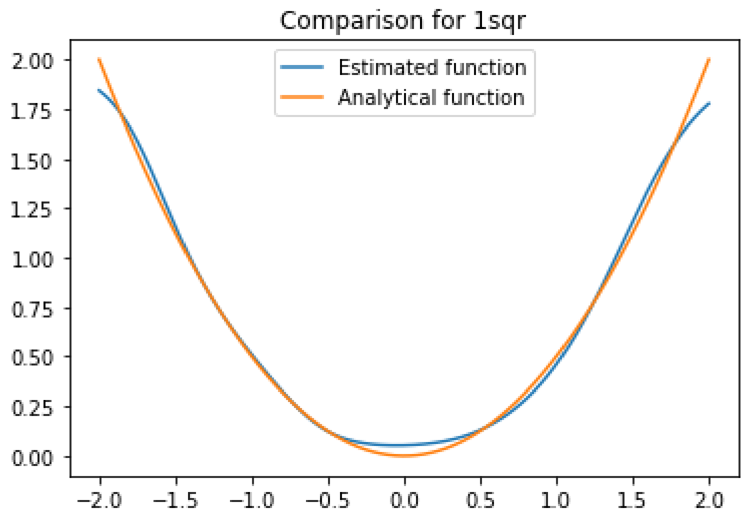
```
def dF_tilde_y(y, h, th, d_0, d, K):

    Z, Upsilon = F_tilde(y, th, d_0, d, K, h)

    dGZK = eta(th["w"].T@Z[K]+ th["mu"], derivative = True) * th["w"]

    A = dGZK
    for k in range(K,0,-1):
        A = A + th["W"][k-1].T@(h*sigma((th["W"][k-1]@Z[k-1] + th["b"][k-1]), derivativ
e = True)*A)

    return A[:d_0]
```

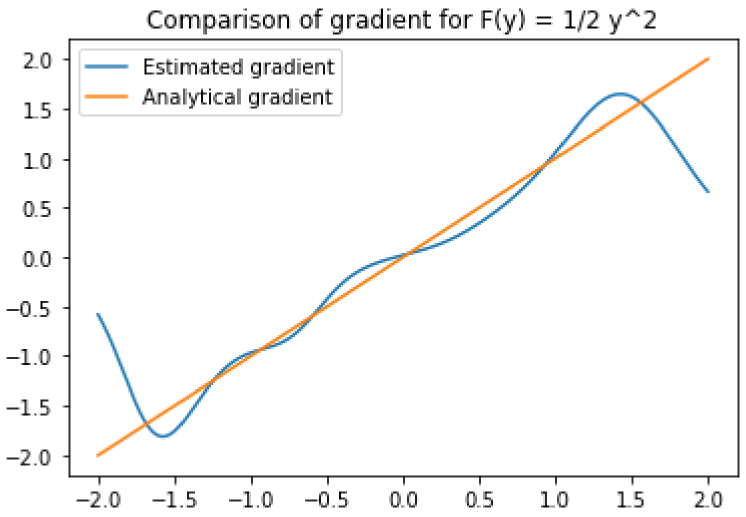
In [42]:

```
def test_grad_1sqr():

    func = "1sqr"

    numData = 2000
    K = 20
    h = 0.1


    Y  = np.linspace(-2,2,numData)
    Y = Y[:,np.newaxis].T
    c = Y
    c = c.T

    d_0 = Y.shape[0]
    d = d_0*2


    inv_file = open( func+"_inv.pkl", "rb")
    inv = pickle.load(inv_file)
    inv_file.close()

    th_file = open(func + "_th.pkl", "rb")
    th = pickle.load(th_file)
    th_file.close()

    yhat = dF_tilde_y(Y, h, th, d_0, d, K)

    y = invscaleparameter_no_shift(yhat, inv[0], inv[1], inv[2], inv[3])

    plt.plot(Y.T, y.T, label ="Estimated gradient")
    plt.plot(Y.T, c, label ="Analytical gradient")
    plt.title("Comparison of gradient for F(y) = 1/2 y^2")
    plt.legend()
    plt.show()
```

In [41]:

```
test_func(func="1sqr")
test_grad_1sqr()
```





As we can see the impementation does a good job of calulating the gradient of the numerical function. We see that the numerical function diverges away from the analytical one. So it makes sence for the numerical gradient to also diverge from the numerical one.

5.

Implement symplectic Euler and the Størmer-Verlet method for the Hamiltonian

Symplectic Euler is formulated as

$$q_{n+1} = q_n + \Delta t \frac{\partial T}{\partial p}(p_n)$$

$$p_{n+1} = p_n - \Delta t \frac{\partial V}{\partial q}(q_n)$$

In [43]:

```
def s_euler(p0, q0, thp, thq, hF, K, N, T, invp, invq):

    h = T/N

    d_0 = p0.shape[0]
    d = d_0*2

    p = np.zeros((N+1,1,1))
    p[0] = p0

    q = np.zeros((N+1,1,1))
    q[0] = q0

    for n in range(N):

        dTs = dF_tilde_y(p[n], hF, thp, d_0, d, K)
        dT = invscaleparameter_no_shift(dTs, invp[0], invp[1], invp[2], invp[3])
        dVs = dF_tilde_y(q[n], hF, thq, d_0, d, K)
        dV = invscaleparameter_no_shift(dVs, invq[0], invq[1], invq[2], invq[3])

        q[n+1] = q[n] + h*dT[:d_0]
        p[n+1] = p[n] - h*dV[:d_0]

    p = np.reshape(p,N+1)
    q = np.reshape(q,N+1)
    return p,q
```

Størmer-Verlet is formulated as

$$\begin{aligned} p_{n+1/2} &= p_n - \frac{\Delta t}{2}\frac{\partial V}{\partial q}(q_n) \\ q_{n+1} &= q_n + \Delta t \frac{\partial T}{\partial p}(p_{n+1/2}) \\ p_{n+1} &= p_{n+1/2} - \frac{\Delta t}{2}\frac{\partial V}{\partial q}(q_{n+1}) \end{aligned}$$

In [44]:

```
def stormer_verlet(p0, q0, thp, thq, hF, K, N, T, invp, invq):

    h = T/N

    d_0 = p0.shape[0]
    d = d_0*2

    p = np.zeros((N+1,d_0,1))
    p[0] = p0

    q = np.zeros((N+1,d_0,1))
    q[0] = q0

    for n in range(N):

        dVs = dF_tilde_y(q[n], hF, thq, d_0, d, K)
        dV = invscaleparameter_no_shift(dVs, invq[0], invq[1], invq[2], invq[3])
        p_hat = p[n] - (h/2)*dV

        dTs = dF_tilde_y(p_hat, hF, thp, d_0, d, K)
        dT = invscaleparameter_no_shift(dTs, invp[0], invp[1], invp[2], invp[3])
        q[n+1] = q[n] + h*dT

        dVs = dF_tilde_y( q[n+1], hF, thq, d_0, d, K)
        dV1 = invscaleparameter_no_shift(dVs, invq[0], invq[1], invq[2], invq[3])
        p[n+1] = p_hat - (h/2)*dV1


    p = np.reshape(p,(N+1,d_0))
    q = np.reshape(q,(N+1,d_0))
    return p,q


def stormer_verlet_analytical(p0, q0, N, T,  dT, dV):

    h = T/N

    d_0 = p0.shape[0]
    d = d_0*2

    p = np.zeros((N+1,d_0,1))
    p[0] = p0

    q = np.zeros((N+1,d_0,1))
    q[0] = q0

    for n in range(N):
        # 1
        dVq = dV(q[n])
        phat = p[n] - h/2*dVq
        dTph = dT(phat)
        q[n+1] = q[n] + h*dTph
        dVq = dV(q[n+1])
        p[n+1] = phat - h/2*dVq


    p = np.reshape(p,(N+1,d_0))
    q = np.reshape(q,(N+1,d_0))
```

```
    return p,q
```

5. a/b)

Try in particular to test it on the given Hamiltonians

Test to which extent the numerical solution preserves the Hamiltonian along the trajectories

In [59]:

```
def train_two_body(pq, continue_training = False):

    if pq == "p":
        func = "2sqr"
    elif pq == "q":
        func = "2norm-1"
    else:
        raise Exception("p or q")


    I = 8000
    K = 20
    h = 0.1
    sifts = 1000
    Ihat = 400
    tau = 2/Ihat

    qdata = generate_synthetic_batches(I, func=func)

    q =qdata["Y"]
    cq = qdata["c"]
    scq,invqc = scale(cq)

    parametersq = scale(cq, returnParameters = True)

    invq_file = open( pq + "_tb_inv.pkl", "wb")
    pickle.dump(parametersq, invq_file)
    invq_file.close()

    d_0 = q.shape[0]
    d = d_0*2

    if continue_training:
        qw_file = open( pq + "_tb_w.pkl", "rb")
        thq = pickle.load(qw_file)
        qw_file.close()

    else:
        thq = initialize_weights(d_0, d, K)


    JJq, thq = stocgradient(scq, d, d_0, K, h, q, thq, tau, 1 , Ihat, sifts)

    plt.plot(JJq)
    plt.yscale("log")
    plt.show()

    thq_file = open(pq + "_tb_w.pkl", "wb")
    pickle.dump(thq, thq_file)
    thq_file.close()


def model_two_body():

    K = 20
    hF = 0.1
    d_0 = 1
    d = d_0*2
```

```
T = 6
dt = 1e-2
N = int(T/dt)

pp_file = open("p_tb_inv.pkl", "rb")
invp = pickle.load(pp_file)
pp_file.close()

qp_file = open("q_tb_inv.pkl", "rb")
invq = pickle.load(qp_file)
qp_file.close()

pw_file = open("p_tb_w.pkl", "rb")
thp = pickle.load(pw_file)
pw_file.close()

qw_file = open("q_tb_w.pkl", "rb")
thq = pickle.load(qw_file)
qw_file.close()

p0 = np.array([0.8,0])[:,np.newaxis]
q0 = np.array([0,-1.2])[:,np.newaxis]


def dT(p):
    derr = np.array([p[0],p[1]])
    return derr

def dV(q):
    derr = np.array([q[0]/(q[0]**2+q[1]**2)**(3/2), q[1]/(q[0]**2+q[1]**2)**(3/2)])
    return derr


#p,q = s_euler(p0, q0, thp, thq, hF, K, N, T, invp, invq)
p,q = stormer_verlet(p0, q0, thp, thq, hF, K, N, T, invp, invq)
pa,qa = stormer_verlet_analytical(p0, q0, N, T,  dT, dV)

plt.title("Numerical and analytical values of q")
plt.plot(q[:,0],q[:,1],label="q")
plt.plot(qa[:,0],qa[:,1],label="q analytical")
plt.legend()
plt.show()

Tpa = 1/2*pa[:,0]**2 + 1/2*pa[:,1]**2
Vqa = -1/np.sqrt(qa[:,0]**2 + qa[:,1]**2)

plt.title("Analytical T(p) and V(q) values")
plt.plot(Tpa, label="Ta")
plt.plot(Vqa,  label ="Va")
plt.plot(Tpa+Vqa, label ="Ta + Va")
plt.legend()
plt.show()

Tp = 1/2*p[:,0]**2 + 1/2*p[:,1]**2
Vq = -1/np.sqrt(q[:,0]**2 + q[:,1]**2)

plt.title("Numerical T(p) and V(q) values")
plt.plot(Tp, label="T")
plt.plot(Vq,  label ="V")
plt.plot(Tp+Vq, label ="T + V")
plt.legend()
```

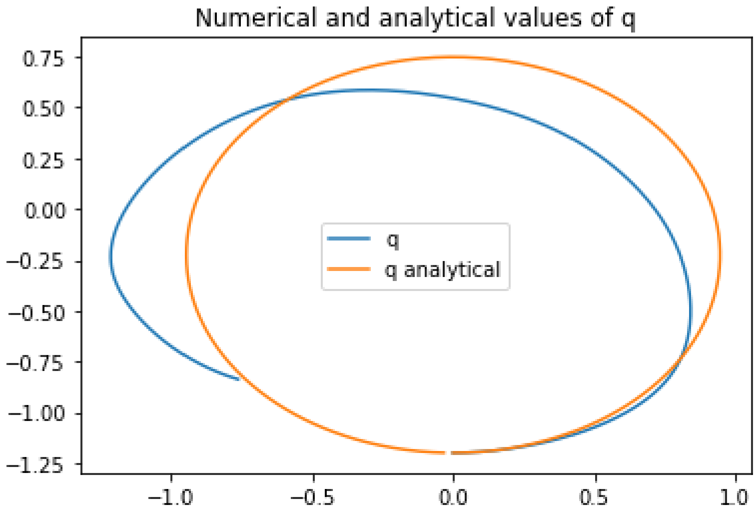
```
    plt.show()


def train_nlp(pq):

    if pq == "p":
        func = "1sqr"
    elif pq == "q":
        func = "1cos"
    else:
        raise Exception("p or q")


    I = 8000
    K = 20
    h = 0.1
    sifts = 2400
    Ihat = 320
    tau = 3/Ihat

    data = generate_synthetic_batches(I, func)

    Y =data["Y"]
    c = data["c"]
    sc , invc = scale(c)
    sparameters = scale(c,returnParameters = True)

    inv_file = open( pq + "_nlp_inv.pkl", "wb")
    pickle.dump(sparameters, inv_file)
    inv_file.close()

    d_0 = Y.shape[0]
    d = d_0*2



    th = initialize_weights(d_0, d, K)

    JJ, th = stocgradient(sc, d, d_0, K, h, Y, th, tau, 1 , Ihat, sifts)

    plt.plot(JJ)
    plt.yscale("log")
    plt.show()


    th_file = open(pq + "_nlp_w.pkl", "wb")
    pickle.dump(th, th_file)
    th_file.close()

def model_nlp():

    K = 20
    hF = 0.1
    d_0 = 1
    d = d_0*2

    T = 12
    dt = 1e-2
    N = int(T/dt)

    invp_file = open("p_nlp_inv.pkl", "rb")
```

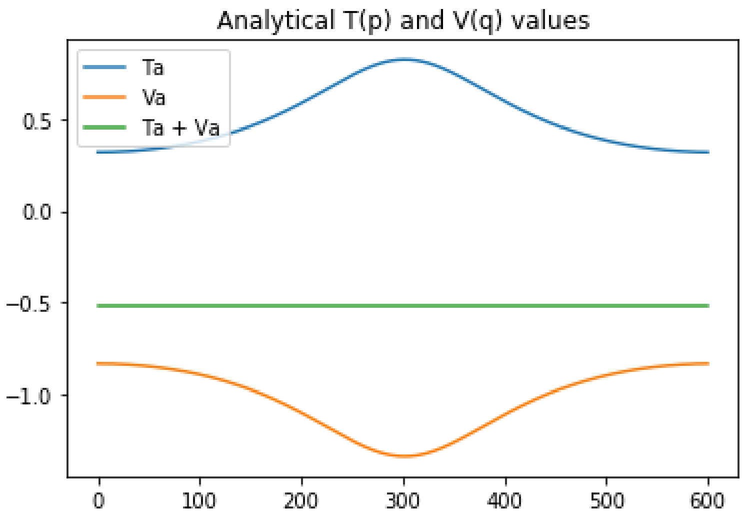
```
invp = pickle.load(invp_file)
invp_file.close()

invq_file = open("q_nlp_inv.pkl", "rb")
invq = pickle.load(invq_file)
invq_file.close()

wp_file = open("p_nlp_w.pkl", "rb")
thp = pickle.load(wp_file)
wp_file.close()

wq_file = open("q_nlp_w.pkl", "rb")
thq = pickle.load(wq_file)
wq_file.close()


p0 = np.array([1])[:,np.newaxis]
q0 = np.array([0])[:,np.newaxis]


def dT(p):
    return p

def dV(q):
    return np.sin(q)

#p,q = s_euler(p0, q0, thp, thq, hF, K, N, T, invp, invq)
p,q = stormer_verlet(p0, q0, thp, thq, hF, K, N, T, invp, invq)
pa,qa = stormer_verlet_analytical(p0, q0, N, T,  dT, dV)

plt.title("Numerical p and q values")
plt.plot(p, label="p")
plt.plot(q, label="q")
plt.legend()
plt.show()

plt.title("Analytical p and q values")
plt.plot(pa, label="p")
plt.plot(qa, label="q")
plt.legend()
plt.show()

Tp = 1/2*p**2
Vq = 1-np.cos(q)

Tpa = 1/2*pa**2
Vqa = 1-np.cos(qa)

plt.title("Numerical T(p) and V(q) values")
plt.plot(np.reshape(Tp,len(Tp)), label="T")
plt.plot(np.reshape(Vq,len(Vq)),  label ="V")
plt.plot(Tp+Vq, label ="T + V")
plt.legend()
plt.show()

plt.title("Analytical T(p) and V(q) values")
plt.plot(np.reshape(Tpa,len(Tpa)), label="T")
plt.plot(np.reshape(Vqa,len(Vqa)),  label ="V")
plt.plot(Tpa+Vqa, label ="T + V")
plt.legend()
plt.show()
```

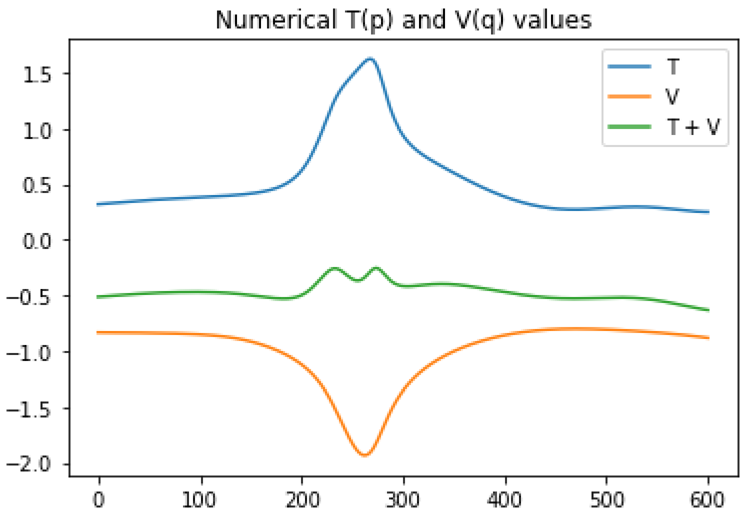
In [57]:

```
print("Kepler two-body:")
model_two_body()
```

```
Kepler two-body:
```





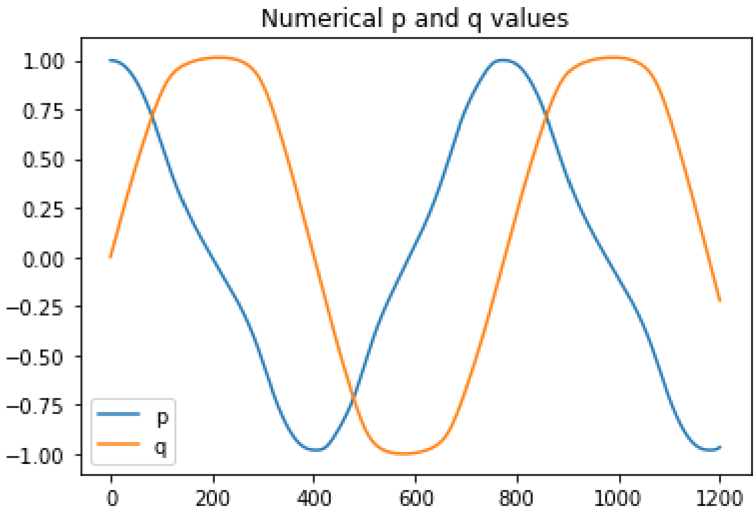


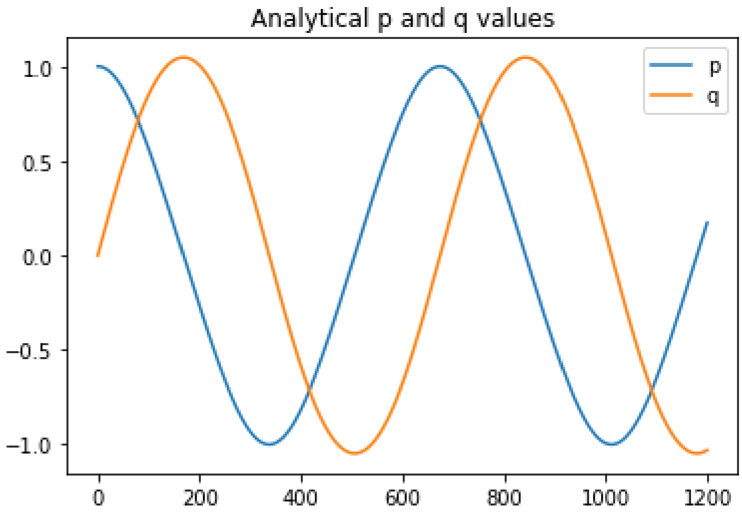
In the kepler two-body problem we observe that the numerical path starts off quite close to the analytical one before diverging, this is probably caused by the the accumulative error of the gradients. The hameltonian is somewhat conserved along the trajectory, however it does fluctuate sligthly.

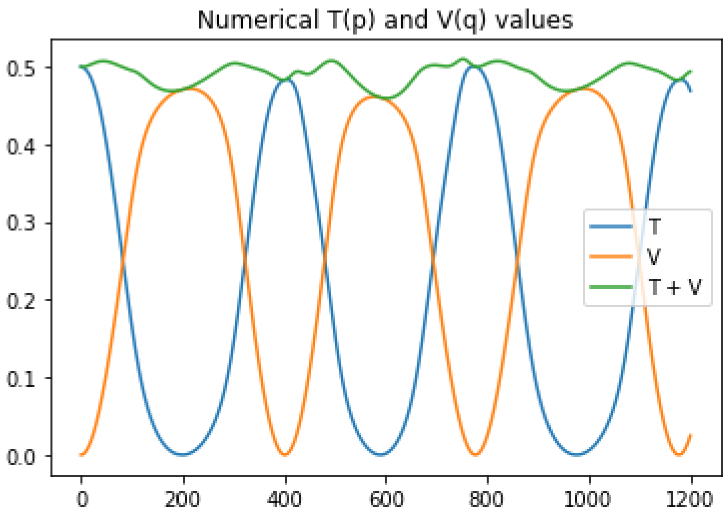
In [60]:

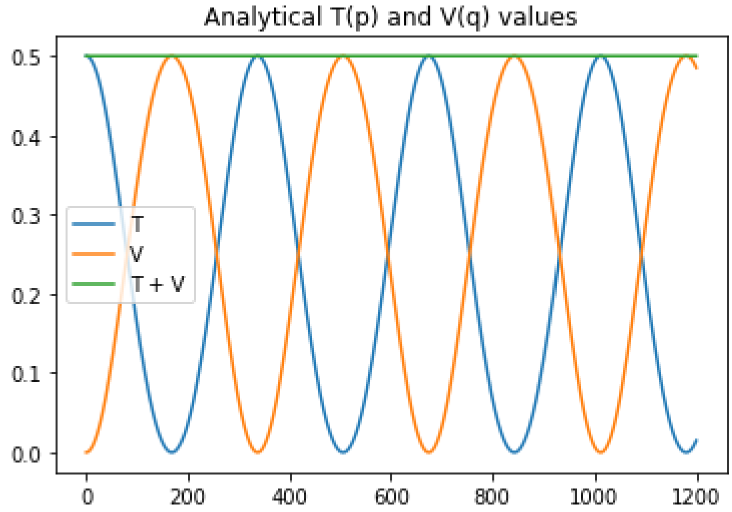
```
print("Nonlinear pendulum:")
model_nlp()
```

Nonlinear pendulum:









For the nonlinear pendulum we observe that the frequency is consistent for the numerical solution, however the frequency of the numerical solution is sligthly lower than the analytical one. The hameltonian does fluctuate somewhat but does not diverge away from the true value.

5. c)

Then try it on the given data with unknown Hamiltonian References

In [68]:

```
def model_unknown(low = 0,high = 49):

    K = 20
    h = 0.1


    invp_file = open("p_unknown_inv.pkl", "rb")
    invp = pickle.load(invp_file)
    invp_file.close()

    invq_file = open("q_unknown_inv.pkl", "rb")
    invq = pickle.load(invq_file)
    invq_file.close()

    wp_file = open("p_unknown_w.pkl", "rb")
    thp = pickle.load(wp_file)
    wp_file.close()

    wq_file = open("q_unknown_w.pkl", "rb")
    thq = pickle.load(wq_file)
    wq_file.close()

    batches = import_batches()
    batch1 = batches[0]
    antB = 50

    Y = batch1["Y_q"]
    d_0 = Y.shape[0]
    d = d_0*2

    N = Y.shape[1]
    b_indexes = np.arange(low,high+1)


    #for i in range(antB):
    for i in b_indexes:
        print("Batch: " + str(i))
        testbatch = batches[i]

        pa = testbatch["Y_p"]
        Ta = testbatch["c_p"]
        qa = testbatch["Y_q"]
        Va = testbatch["c_q"]

        plt.title("Given T(p) and V(q) values")
        plt.plot(np.reshape(Ta,len(Ta)), label="T(p)")
        plt.plot(np.reshape(Va,len(Va)),  label ="V(q)")
        plt.plot(Ta+Va, label ="T(p) + V(q)")
        plt.legend()
        plt.show()


        z, yhatp = F_tilde(pa, thp, d_0, d, K, h)
        Tpa = invscaleparameter(yhatp, invp[0], invp[1], invp[2], invp[3])
        z, yhatq = F_tilde(qa, thq, d_0, d, K, h)
        Vqa = invscaleparameter(yhatq, invq[0], invq[1], invq[2], invq[3])

        plt.title("Numerical T(p) and V(q) values, with given p and q values")
        plt.plot(np.reshape(Tpa,len(Tpa)), label="T(p)")
```

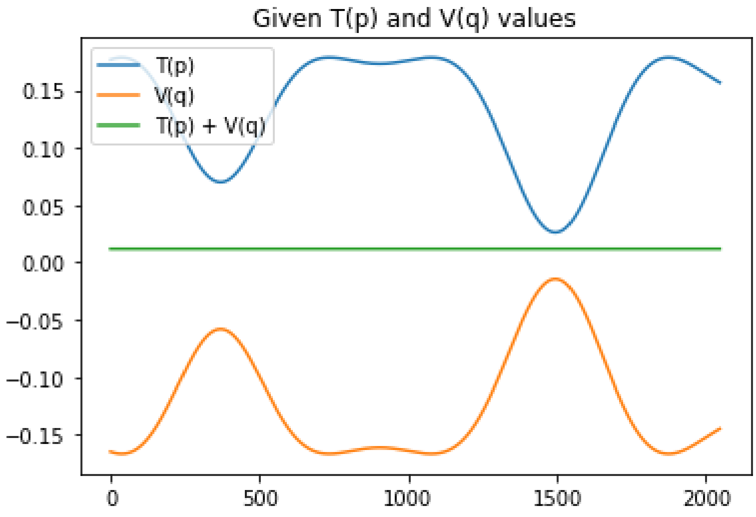
```
    plt.plot(np.reshape(Vqa,len(Vqa)),  label ="V(q)")
    plt.plot(Tpa+Vqa, label ="T(p) + V(q)")
    plt.legend()
    plt.show()


    dt = testbatch["t"][1]
    T = dt*N
    p0 = pa[:,0,np.newaxis]
    q0 = qa[:,0,np.newaxis]
    p,q = stormer_verlet(p0, q0, thp, thq, h, K, N, T, invp, invq)

    z, yhatpn = F_tilde(p.T, thp, d_0, d, K, h)
    Tp = invscaleparameter(yhatpn, invp[0], invp[1], invp[2], invp[3])
    z, yhatqn = F_tilde(q.T, thq, d_0, d, K, h)
    Vq = invscaleparameter(yhatqn, invq[0], invq[1], invq[2], invq[3])

    plt.title("Numerical T(p) and V(q) values, with numerical p and q values")
    plt.plot(np.reshape(Tp,len(Tp)), label="T(p)")
    plt.plot(np.reshape(Vq,len(Vq)),  label ="V(q)")
    plt.plot(Tp+Vq, label ="T(p) + V(q)")
    plt.legend()
    plt.show()
```

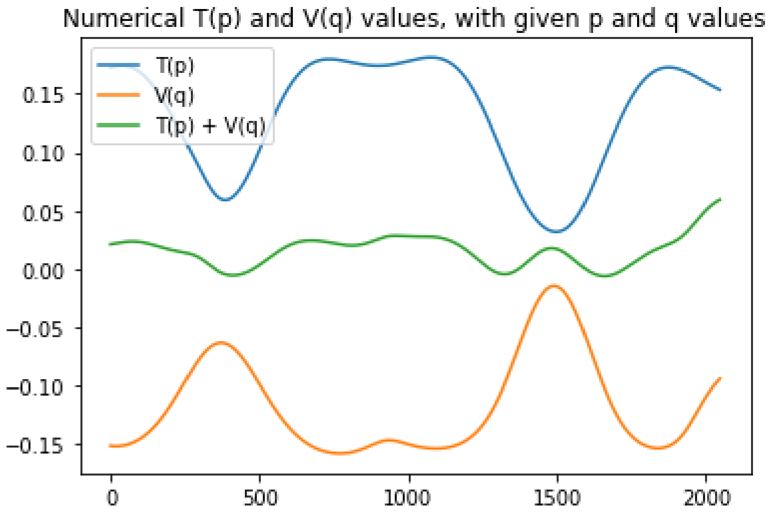
Choose what bathces to look at by seting a range as the parameters in model_unknown.

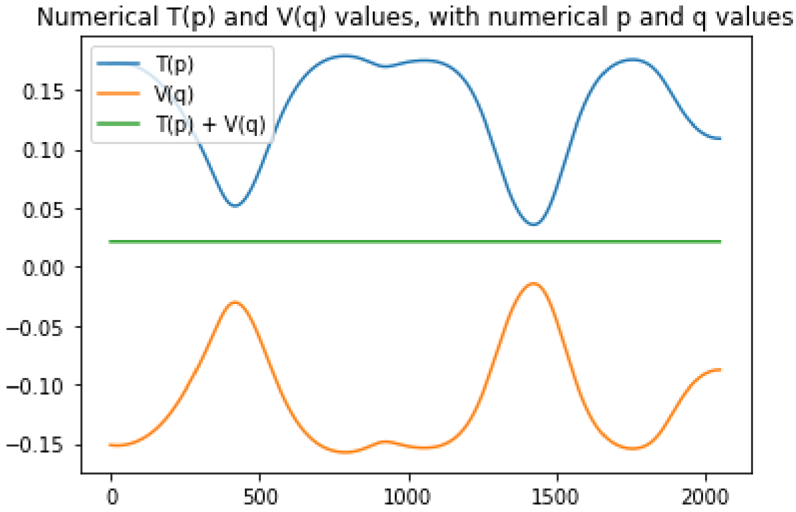
In [71]:

```
model_unknown(0, 1)
model_unknown(45,46)
```

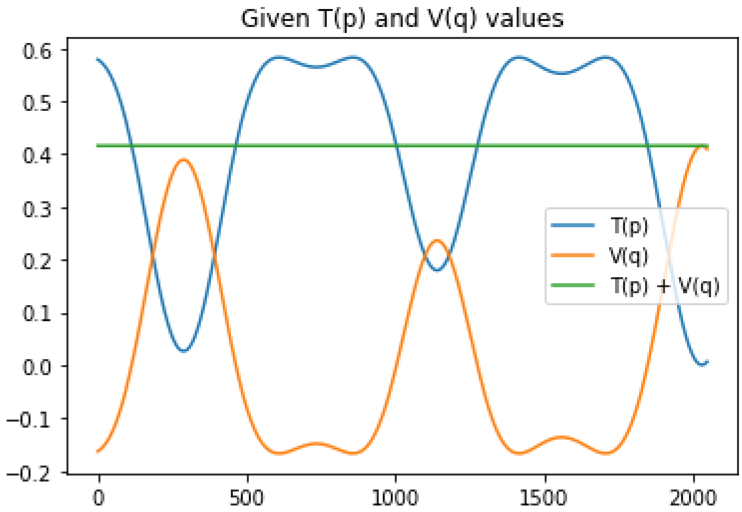
Batch: 0

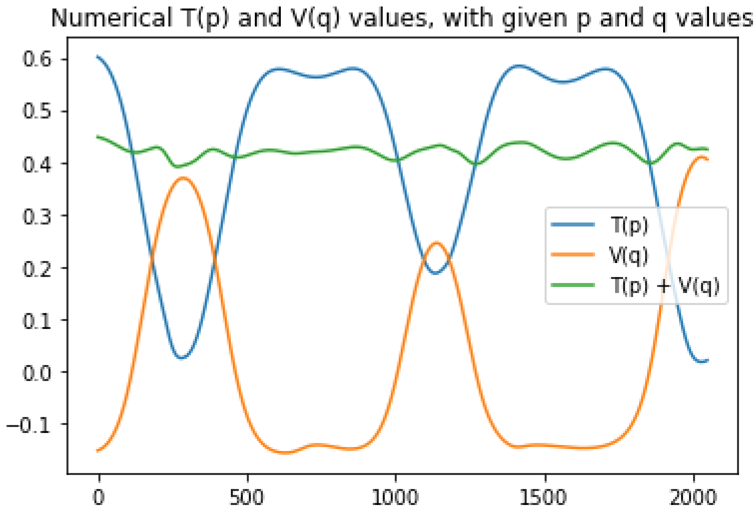

Given T(p) and V(q) values
T(p)
V(q)
T(p) + V(q)
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
500
1000
1500
2000

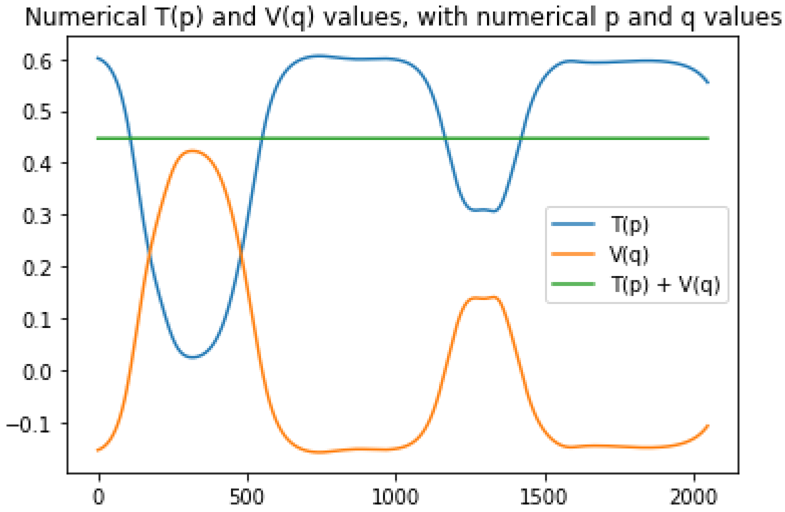

Numerical T(p) and V(q) values, with given p and q values
T(p)
V(q)
T(p) + V(q)
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
500
1000
1500
2000


Numerical T(p) and V(q) values, with numerical p and q values
T(p)
V(q)
T(p) + V(q)
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
500
1000
1500
2000

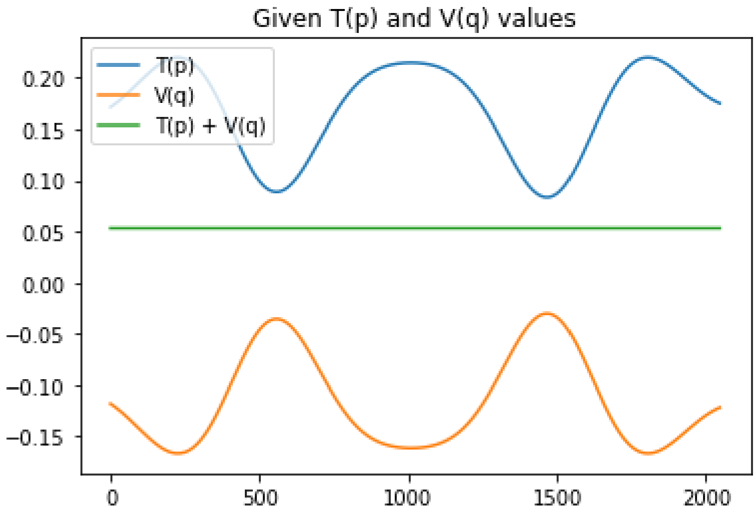
```
Batch: 1
```

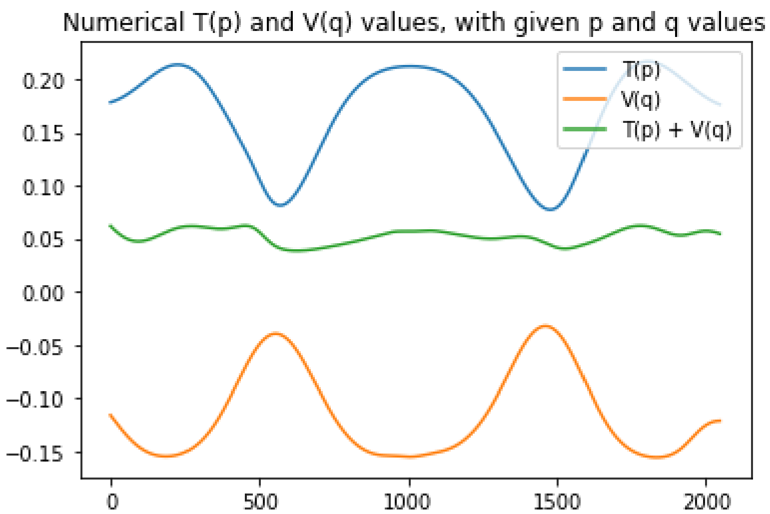

Given T(p) and V(q) values
T(p)
V(q)
T(p) + V(q)
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
0
500
1000
1500
2000

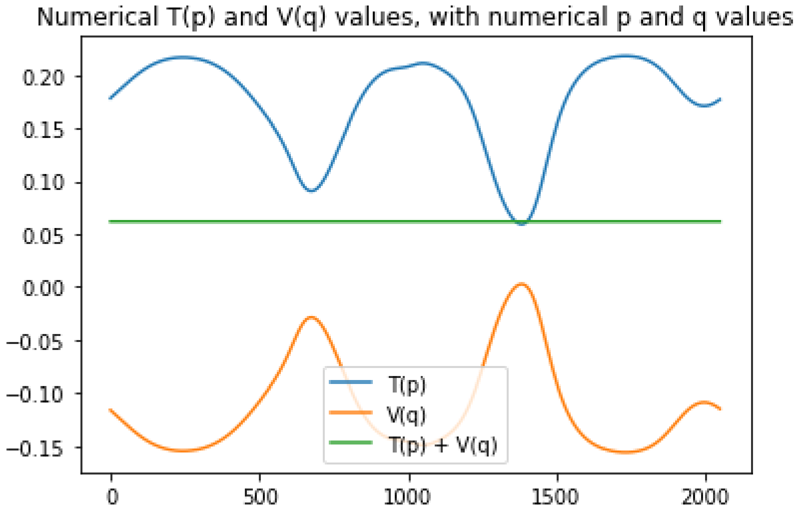

Numerical T(p) and V(q) values, with given p and q values
T(p)
V(q)
T(p) + V(q)
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
0
500
1000
1500
2000


Numerical T(p) and V(q) values, with numerical p and q values
T(p)
V(q)
T(p) + V(q)

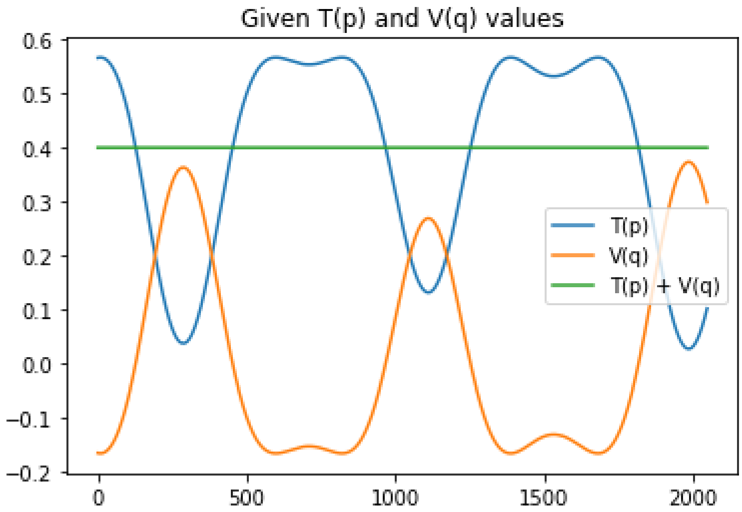
```
Batch: 45
```

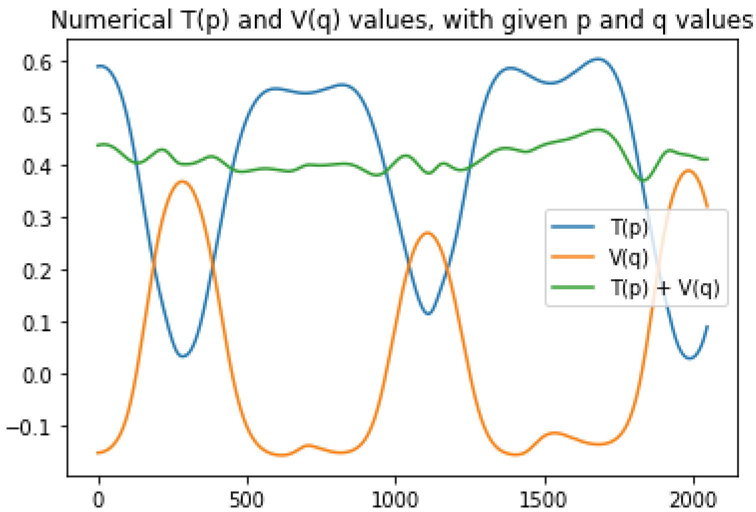

Given T(p) and V(q) values
T(p)
V(q)
T(p) + V(q)

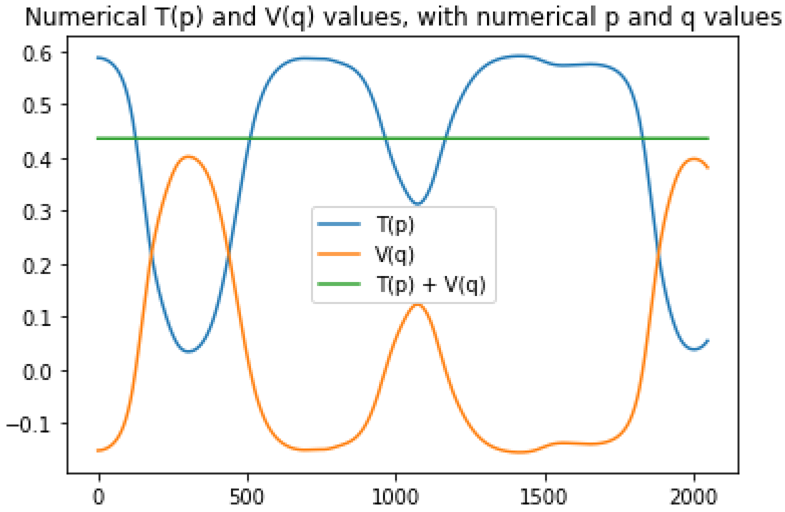

Numerical T(p) and V(q) values, with given p and q values
T(p)
V(q)
T(p) + V(q)


Numerical T(p) and V(q) values, with numerical p and q values
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
0
500
1000
1500
2000
T(p)
V(q)
T(p) + V(q)

```
Batch: 46
```


Given T(p) and V(q) values
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
−0.2
0
500
1000
1500
2000
T(p)
V(q)
T(p) + V(q)


Numerical T(p) and V(q) values, with given p and q values
0.6
0.5
0.4
0.3
0.2
0.1
0.0
−0.1
0
500
1000
1500
2000
T(p)
V(q)
T(p) + V(q)



The first plot is the given hamiltonian along the given trajectory, the second the numerical hamiltionian along the given trajectory, and the third the numerical hamiltionan along the numerically calculated trajectory. As expected in the first plot, the hamiltionian is constant. In the second the hamiltionian fluctuates slightly due to the difference between the numerical and analytical functions. Suprisingly, in the third plot the hamiltinian is constant despite the numerical trajectory being different from the analytical one. We get the same result, regardless of if the trajectory was part of the training-set or not.

In [ ]:

```

```

Table of contents

1.

Implement functions for generating synthetic input data.

2.

Implement the neural network for training approximation of Hamiltonian function.

2. a)

Test the model by using the suggested functions

2. b)

Investigate systematically what are optimal choices for K, τ, d, h and any other choices you need to make. Balance performance in the generalisation phase with time consumption of training.

2. c)

Train the model for the case of data given (with unknown Hamiltonian function)

2. d)

Try other alternatives for optimisation, such a the Adam method

2. e)

Make use of convergence plots for getting an indication of the efficiency of your choices.

2. g/f)

Make use of convergence plots for getting an indication of the efficiency of your choices.

Do a similar evaluation on test data (that were not used in the training phase).

3.

Derive the formulas for computing the gradient of the trained function ($\nabla_y F(y)$)

4.

Implement these formulas for computing the gradient

5.

Implement symplectic Euler and the Størmer-Verlet method for the Hamiltonian function

5. a/b)

Try in particular to test it on the given Hamiltonians

HamiltonianTest to which extent the numerical solution preserves the Hamiltonian alongtent the numerical solution preserves the Hamiltonian along

5. c)

In [ ]:

```

```