Computational Physics
Chapter 3 Problems

Cristopher Cerda Puga

Exercises 3.2

Modify the programs 3.4 and 3.5 to find the roots of the function $f(x) = \sin x$.

Listing 1: Program 3.4 Newton-Rapson Method

```
# defines the function
def f(x):
    f = x**3-x**2-x+1
    return f

def df(x):
    der = 3*x**2-2*x-1
    return der

#########################################################

def nr(x0,nmax,tol):
    x = x0
    if abs(df(x)) < tol:
        dx = 0
    else:
        dx = f(x)/df(x)
    for i in range(nmax):
        n = i
        if abs(dx) < tol:
            x0 = x -dx
            break
        x = x0-dx
        if abs(df(x)) < tol:
            dx = 0
        else:
            dx = f(x)/df(x)
        x0 = x
    x = x0
    return (x,f(x),n)
```

Listing 2: Program 3.5 Newton-Rapson method with arbitrary precision.

```
import mpmath
from mpmath import *

mp.dps = 40; mp.pretty = True

#defines the function

def f(x):
    x2 = mpmath.fmul(x,x)
    x3 = mpmath.fmul(x,x2)
    p1 = mpmath.fadd(x3,1)
    p2 = mpmath.fsub(p1,x2)
    f = mpmath.fsub(p2,x)

    return f

# defines the derivative of the function

def df(x):
    x2 = mpmath.fmul(x,x)
    p1 = mpmath.fmul(3,x2)
    p2 = mpmath.fmul(2,x)
```

```
23      p3 = mpmath.fsub(p1,p2)
24      der = mpmath.fsub(p3,1)
25      return der
26
27  ##################################################
28
29  def nrmpmath(x0,nmax,tol):
30      x = x0
31      if mpmath.fabs(df(x))<tol:
32          dx = 0
33      else:
34          dx = f(x)/df(x)
35      for i in range(nmax):
36          nn = i
37          if mpmath.fabs(dx)<tol:
38              x0 = x - dx
39              break
40          x = x0-dx
41          if mpmath.fabs(df(x)) < tol:
42              dx = 0
43          else:
44              dx = f(x)/df(x)
45          x0 = x
46      x = x0
47
48      return(x,f(x),nn)
```

Modifying the previous codes, we get the following:

- Modification of program 3.4:

```
1   import mpmath as mp
2   from mpmath import *
3   import math
4
5   def f(x):
6       f = math.sin(x)
7       return f
8
9   def df(x):
10      df = math.cos(x)
11      return df
12
13  def nr(x0,nmax,tol):
14      x = x0
15      if abs(df(x)) < tol:
16          dx = 0
17      else:
18          dx = f(x)/df(x)
19      for i in range(nmax):
20          n = i
21          if abs(dx) < tol:
22              x0 = x -dx
23              break
24          x = x0-dx
25          if abs(df(x)) < tol:
26              dx = 0
27          else:
28              dx = f(x)/df(x)
29          x0 = x
30      x = x0
```

```
31      return (x,f(x),n)
32
33
34  print(nr(math.pi,100,1e-100))
35  >>> (3.141592653589793, 1.2246467991473532e-16, 99)
```

- Modification of the program 3.5:

```
1
2   import mpmath as mp
3   from mpmath import *
4   mp.dps = 40; mp.pretty = True
5
6   #defines the function
7   def g(x):
8       g = mp.sin(x)
9       return g
10
11  # defines the derivative of the function
12  def dg(x):
13      dg = mp.cos(x)
14      return dg
15
16  def nrmp(x0,nmax,tol):
17      x = x0
18      if mp.fabs(df(x))<tol:
19          dx = 0
20      else:
21          dx = f(x)/df(x)
22      for i in range(nmax):
23          nn = i
24          if mp.fabs(dx)<tol:
25              x0 = x - dx
26              break
27          x = x0-dx
28          if mp.fabs(df(x)) < tol:
29              dx = 0
30          else:
31              dx = f(x)/df(x)
32          x0 = x
33      x = x0
34
35      return(x,g(x),nn)
36
37
38  print(nrmp(mp.pi,100,1e-100))
39  >>> (3.14159265358979305043275905505763162387 1, ↩
        1.88029884328221871260326456831956502997 3e-16, 99)
```

- Try to calculate as many digits of π as you can ussing the Newton-Raphson method:

  To improve the code and get more digits of pi, you can modify the input parameters `x0`, `nmax`, and `tol`. Increasing `nmax` will allow for more iterations, which can lead to higher precision. We can also decrease `tol` to set a stricter tolerance for convergence.

```
1   import mpmath as mp
2   from mpmath import *
3   mp.dps = 40; mp.pretty = True
4
```

```
# Context from Function problems/Homework3/pidigits.py:df
def f(x):
    f = mp.sin(x)
    return f

def df(x):
    df = mp.cos(x)
    return df

#create a function that gives you an aproximation of pi using the Newton-Raphson↩
     method
def pi(x0,nmax,tol):
    x = x0
    if mp.fabs(df(x))<tol:
        dx = 0
    else:
        dx = f(x)/df(x)
    for i in range(nmax):
        nn = i
        if mp.fabs(dx)<tol:
            x0 = x - dx
            break
        x = x0-dx
        if mp.fabs(df(x)) < tol:
            dx = 0
        else:
            dx = f(x)/df(x)
        x0 = x
    x = x0

    return(x,f(x),nn)

print(pi(22/7,1000,0.0001))
>>> (3.14159265358979323846264338381538582752, ↩
    -1.020356985548841513566328914117877994006e-28, 1)
```

- The series obtained by Bailey-Borwein-Pluffe (BBP)

$$\sum_{n=0}^{\infty}\left(\frac{1}{16}\right)^n\left(\frac{4}{8n+1}-\frac{2}{8n+4}-\frac{1}{8n+5}-\frac{1}{8n+6}\right)$$

is known to converge to π. The partial sums converge very fast and they can be used to calculate π with high precision, with a finit number of terms. For example, with just the first 1000 terms of the sum, one obtains the first 1200 digits of π.
Write a python program that uses `mpmath` to calculate the partial sums with n therms (with n sufficiently large) and use the result obtained in this way to estimate how fast the Newton-Raphson method is convergent to π.

Listing 3: Aproximation of π using BBP series.

```
from mpmath import mp

# Set the decimal places to a high value
mp.dps = 1200

# Function to calculate the nth term for BBP series
def bbp_term(n):
    return mp.mpf(1)/(16**n) * (
        mp.mpf(4)/(8*n + 1) -
```

```
        mp.mpf(2)/(8*n + 4) -
        mp.mpf(1)/(8*n + 5) -
        mp.mpf(1)/(8*n + 6)
    )

# Calculate the value of pi using BBP series
def calculate_pi():
    pi = mp.mpf(0)
    for k in range(0, mp.dps):
        pi += bbp_term(k)
    return pi


# Calculate and print the value of pi
pi = calculate_pi()
print(pi)
```

This series has a pretty fast rate of convergence, as we can see in the next plot, the error between the series terms and the value of π decreases very rapidly:

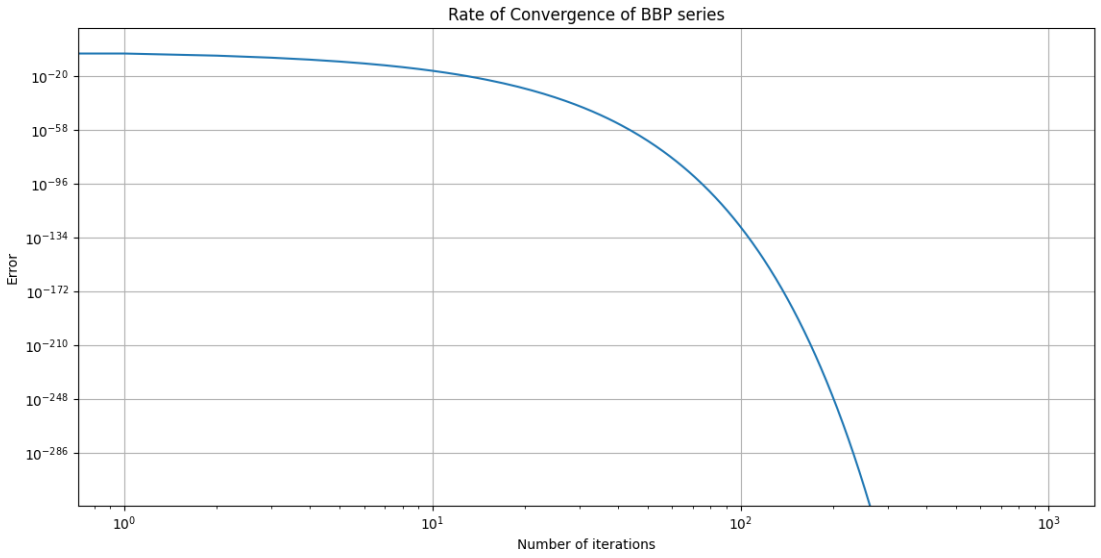

Figure 1: Rate of convergence of BBP series

Now, performing the same analysis to the Newton-Raphson method:

```
from mpmath import mp
import matplotlib.pyplot as plt
import mynrmethod as nr
import numpy as np

# Set the decimal places to a high value
mp.dps = 1200

# Helper function to calculate the nth term for BBP series
def bbp_term(n):
    return mp.mpf(1)/(16**n) * (
        mp.mpf(4)/(8*n + 1) -
        mp.mpf(2)/(8*n + 4) -
```

```
        mp.mpf(1)/(8*n + 5) -
        mp.mpf(1)/(8*n + 6)
    )

n_ite = 10**3
# Calculate the value of pi using BBP series
def calculate_pi():
    pi = mp.mpf(0)
    pi_value = []
    for k in range(0, n_ite):
        pi += bbp_term(k)
        pi_value.append(pi)
    return pi_value

# convergence of BBP series
pi_value = calculate_pi()
convergencebbp = []
for i in range(0,n_ite):
    convergencebbp.append(mp.fabs(mp.pi-pi_value[i]))



# convergence of NR method
values = []
ite = []
convergencenr = []

values = nr.pi(4,n_ite,1e-1200)[1]
ite = nr.pi(4,n_ite,1e-1200)[0]

for i in range(0,len(ite)):
    convergencenr.append(mp.fabs(mp.pi-values[i]))


# Plot the rate of convergence
plt.plot(range(0, n_ite), convergencebbp,label='BBP convergence rate')
plt.plot(range(0,n_ite), convergencenr,label='NR convergence rate')
plt.yscale("log")
plt.xscale("log")
plt.xlabel('Number of iterations')
plt.ylabel('Error')
plt.title('Rate of Convergence to $\pi$')
plt.grid(True)
plt.legend()
plt.show()
```

One can notice that the Newton-Raphson method converges even more rapidly than the BBP series

- Write a python program that uses mpmath to calculate the partial sums with n terms (with n sufficiently large) and use the result obtained in this way to estimate how fast the Netwon-Raphson method is converging to . You should find that your results converge quadratically, i.e. that the number of correct digits doubles at each iteration if you are close enough to the root. Compare this rate of convergence with the one you would have using the bisection method;

```
import mpmath as mp
from mpmath import *
import matplotlib.pyplot as plt
mp.dps = 1200; mp.pretty = True

# Context from Function problems/Homework3/pidigits.py:df
```

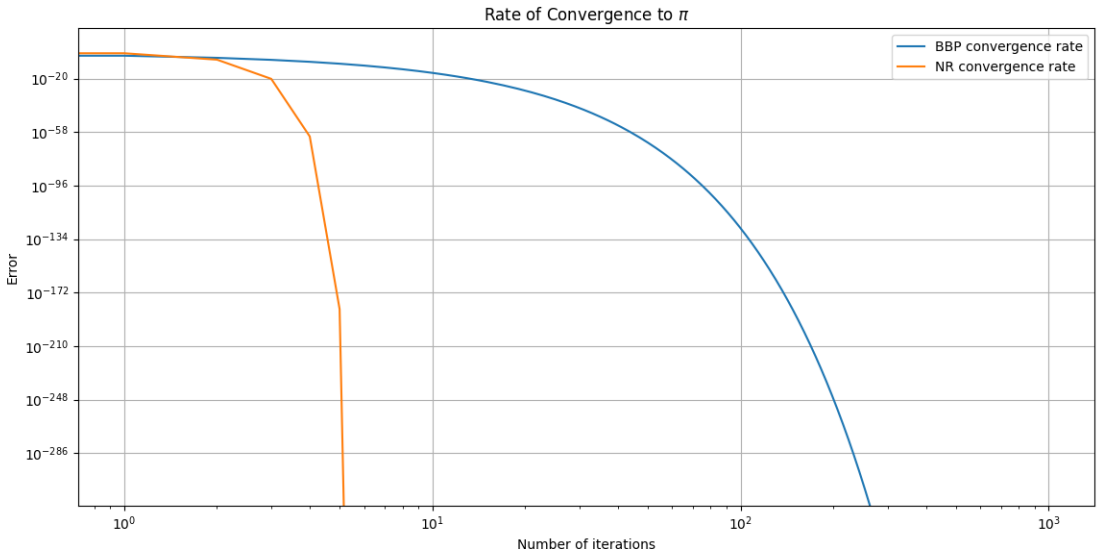

Figure 2: Comparison of convergence rate.

```
7  def f(x):
8      f = mp.sin(x)
9      return f
10
11 def df(x):
12     df = mp.cos(x)
13     return df
14
15 #create a function that gives you an aproximation of pi using the Newton-Raphson↩
       method
16 def pi(x0,nmax,tol):
17     value = []
18     ite = []
19     x = x0
20     if mp.fabs(df(x))<tol:
21         dx = 0
22     else:
23         dx = f(x)/df(x)
24     for i in range(nmax):
25         nn = i
26         if mp.fabs(dx)<tol:
27             x0 = x - dx
28             break
29         x = x0-dx
30         if mp.fabs(df(x)) < tol:
31             dx = 0
32         else:
33             dx = f(x)/df(x)
34         if(x0 == mp.pi):
35             break
36         x0 = x
37         value.append(x0)
38         ite.append(nn)
39     x = x0
40     return ite,value
41
```

```
42
43 n_ite = pi(22/7,10**3,1e-1200)[-1]
44 value = pi(22/7,10**3,1e-1200)[1]
45
46 l = len(value)-2
47
48 # calculate the order of convergence of the method
49
50 x_k = value[l]
51 x_kp1 = value[l+1]
52 x_km1 = value[l-1]
53 x_km2 = value[l-2]
54
55 q = mp.log(mp.fabs((x_kp1-x_k)/(x_k-x_km1)))/mp.log(mp.fabs((x_k - x_km1)/(x_km1↩
       -x_km2)))
56 print(q)
57 >>> 2.99999999999999999999999999999999999999999999...
```

It seems that the order of convergence of this method is cubic!

Exercise 3.3

Write a program to calculate $\sqrt{2}$ using the Newton-Raphson method. Obtain the first 100 digits of $\sqrt{2}$ using the `mpmath` module. How many iterations are needed to reach this precision?

Listing 4: Approximating $\sqrt{2}$ with Newthon-Raphson method.

```
1  import mpmath as mp
2  from mpmath import *
3  import numpy as np
4
5  mp.dps = 100; mp.pretty = True;
6
7  # construct a function using mpmath whose roots is the square root of 2
8  def f(x):
9      x1 = mp.fmul(x,x)
10     f = mp.fsub(x1,2)
11     return f
12
13 def df(x):
14     df = mp.fmul(x,2)
15     return df
16
17 # Newton-Raphson method to find the roots of f(x):
18 def nr(x0,nmax,tol):
19     x = x0
20     if mp.fabs(df(x))<tol:
21         dx = 0
22     else:
23         dx = f(x)/df(x)
24     for i in range(nmax):
25         nn = i
26         if mp.fabs(dx)<tol:
27             x0 = x - dx
28             break
29         x = x0-dx
30         if mp.fabs(df(x)) < tol:
31             dx = 0
32         else:
33             dx = f(x)/df(x)
```

```
34          x0 = x
35      x = x0
36      return (x,f(x),nn)
37
38  root = nr(2,100,1e-100)
39  print(root)
40  >>> ↩
        (1.414213562373095048801688724209698078569671875376948073176679737990732478462107038850387534327
         0.0, 8) #It takes 8 iterations to reach such precision
41
42  # Check if the root corresponds to the first 100 digits of sqrt(2)
43  difference = mp.fabs(root[0] -mp.sqrt(2))
44  print(difference)
45  >>> 0.0
```

Exercise 3.4

You may wonder whether using the fancy `odeint` package in the example above is really needed: `odeint` performs an accurate numerical integration of the differential equation for $0 \leqslant t \leqslant T$, but the purpose of the calculation is just to obtain $x(T)$, without really caring about the accuracy of the trajectory.

Assuming that we proceed by samll time steps, $0 < dt \ll T$, we can write:

$$x(t+dt) \approx x(t) + \dot{x}(t)dt$$
$$\dot{x}(t+dt) \approx \dot{x}(t) + \ddot{x}(t) = \dot{x}(t) - (V'(x(t)) - k\dot{x}(t))dt$$

In this way, knowing the position and velocity of the particle at $t = 0$, we can obtain the position and velocity at $t = dt$:

$$x(dt) \approx x(0) + \dot{x}(0)dt$$
$$\dot{x}(dt) \approx \dot{x}(0) - (V'(x(t)) - k\dot{x}(t))_{t=0}dt$$

If we iterate this procedure we can obtain both the (approximate) position and the velocity of the particle at times $2dt, 3dt, \ldots$. This is known as the **Euler method**.
Modify the previous program to use the Euler method rather than `odeint`. Be careful in choosing dt not too large, because you colud end uo in the unbounded region. Monitor the total mechanical energy of the particle as function of time and do the same for the previous program. Compare the results, for similar values of dt and T, and same initial conditions.

First of all, modifying the previous program to get the $\sqrt{2}$ approximation using the Euler method:

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3
4   # Define the differential equation
5   def equation(x, x_dot):
6       V_prime = -x**2 + 2
7       kappa = 1
8       return V_prime - kappa*x_dot
9
10  # Set the initial conditions
11  t0 = 0
12  x0 = 2
13  x_dot0 = 0
14
15  # Set the time step size and the number of iterations
16  dt = 0.01
```

```
num_iterations = 10**4

# Initialize arrays to store the solution
x = np.zeros(num_iterations)
x_dot = np.zeros(num_iterations)

# Perform Euler's method
x[0] = x0
x_dot[0] = x_dot0
for i in range(1, num_iterations):
    x_dot[i] = x_dot[i-1] + equation(x[i-1], x_dot[i-1])*dt
    x[i] = x[i-1] + x_dot[i-1]*dt

# Approximate the value of sqrt(2)
approx_sqrt2 = x[-1]
print(approx_sqrt2)
>>> 1.414213562373098
```

And producing the phase plane plot:

```
#Plot the results
plt.plot(x,x_dot)
plt.xlabel('x')
plt.ylabel('v')
plt.title('Euler method solution')
plt.grid(True)
plt.show()
```

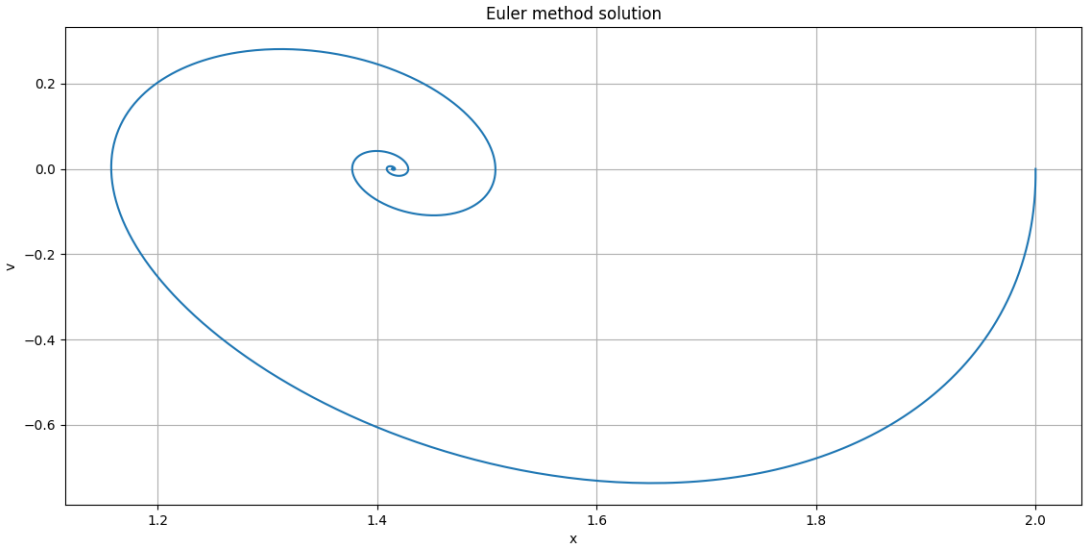

Figure 3: $\dot{x}$ vs x

And finally the mechanical energy as function of time:

```
energy = (x_dot**2)/2 - (x**3)/3 + 2*x

plt.plot(t, energy)
plt.xlabel('t')
plt.ylabel('Energy')
plt.title('Mechanical energy')
plt.grid(True)
plt.show()
```

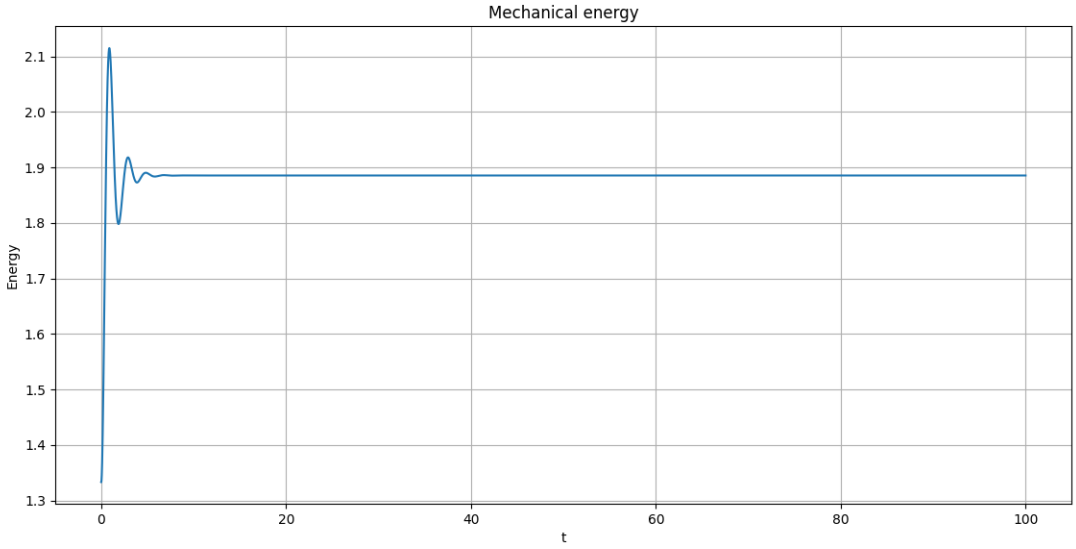

Figure 4: Mechanical Energy

Now, the same mechanical energy for the `odeint` method:

```
...
sol = odeint(ode, y0, t, args=(b, c))

# Calculate the mechanical energy
v = sol[:,1]
x = sol[:,0]

energy = (v**2)/2 - (x**3)/3 + 2*x

# plot the mechanical energy of the system

#plt.plot(t, sol[:, 0], 'b', label='theta(t)')
plt.plot(t, energy, label='Mechanica Energy')
plt.legend(loc='best')
plt.xlabel('t')
plt.grid()
plt.show()
```

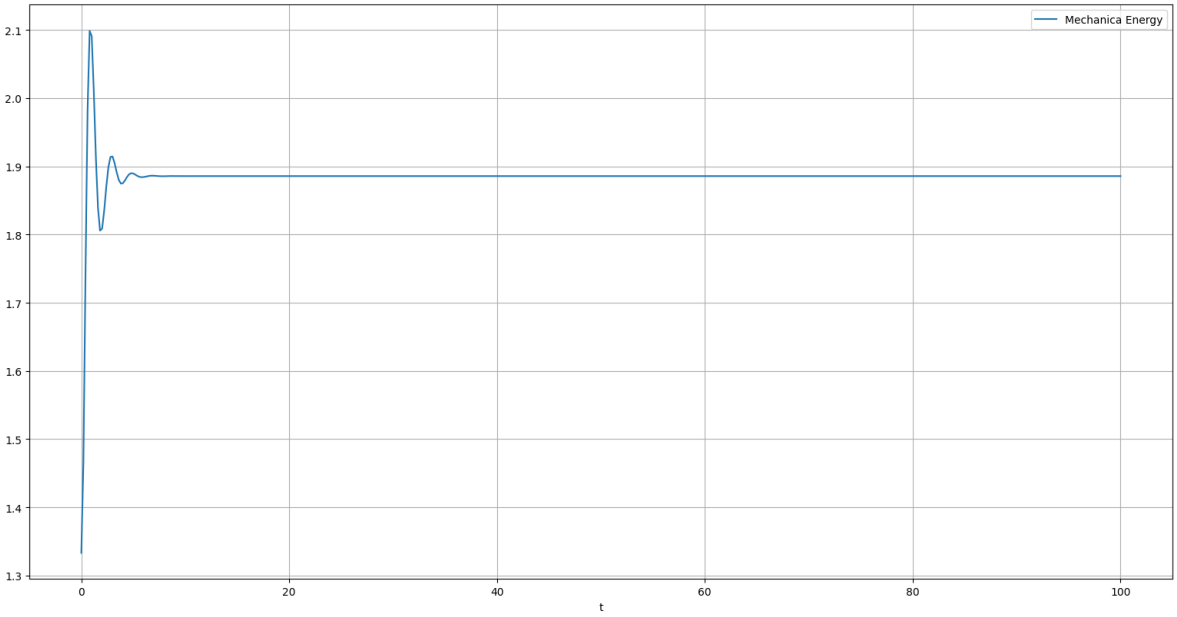

Figure 5: Mechanical Energy