homework4Final

November 30, 2021

1 About python imports

No imports are specified as in previous homeworks, please insert the necessary python code yourself.

---

2 Exercise 1 (3 points)

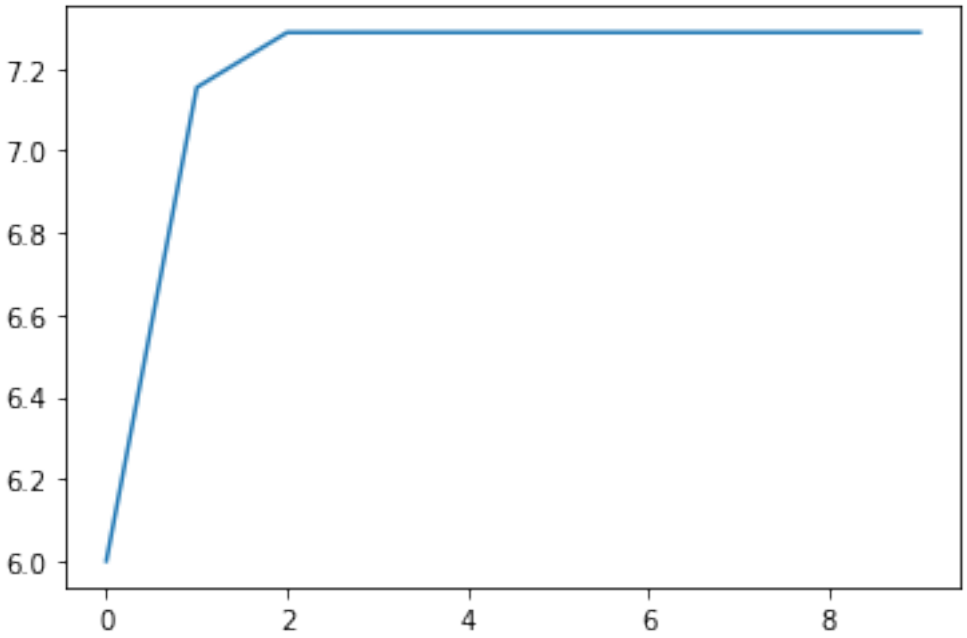
Write a program implementing Rayleigh quotient iteration for computing an eigenvalue and corresponding eigenvector of a matrix. Test your program on the matrix

$$A = \begin{bmatrix} 6 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

using a random starting vector. Let the program create output that shows the convergence behavior.

[2]:
```
import numpy as np
import matplotlib.pyplot as plt

def rayleigh(A, x, iterations):
    """
    Apply Rayleigh quotient iteration on a given matrix A and vector x.
    """
    x_ks = [x]
    sigma_ks = []
    for i in range(iterations):
        # Get sigma of current iteration
        sigma_k = x_ks[-1].T.dot(A).dot(x_ks[-1])/x_ks[-1].T.dot(x_ks[-1])
        sigma_ks.append(sigma_k)

        # Get y_k and normalize to find x_k
        y_k = np.linalg.solve(A - sigma_k*np.identity(len(A)), x_ks[-1])
        x_ks.append(y_k/np.linalg.norm(y_k))
    return x_ks, sigma_ks

# Apply Rayleigh quotient iteration for x = [1 1 1]^T
x = np.ones(3)
A = np.array(((6,2,1),(2,3,1),(1,1,1)))
```

```
x_k, sigma_k = rayleigh(A,x,10)

# Plot all iterations of Rayleigh to show convergence
plt.plot(sigma_k)
plt.show()
```



3 Automatic differentiation using JAX

In applications of rootfinding, computing the derivative is often a problematic step. For example, the function for which zeros are sought might be given by a complicated computer program.

Automatic differentiation is a set of techniques to evaluate the derivative of a function specified by a computer program. See the wikipedia page on this topic.

JAX is a software package that implements automatic differentiation as well as other functionality. The documentation is here. The idea of this exercise is to use JAX for obtaining derivative functions.

To use JAX, there are two options: - install JAX. The installation of JAX is described on the Github page, see https://github.com/google/jax#installation. **It appears that installation under Windows is not supported. According to the internet, one may use the Windows Subsystem for Linux, but I haven't tested this.** - run your python notebook on the google colab environment. The google colab environment is at https://colab.research.google.com. In the google colab environment, the JAX package is available.

There is some material online about JAX, see for example https://medium.com/swlh/solving-optimization-problems-with-jax-98376508bd4f (LATEX-pdf version here https://github.com/mazy1998/Solving-Optimization-Problems-with-

JAX/blob/master/Opitimization_with_jax.pdf) or https://www.kaggle.com/aakashnain/tf-jax-tutorials-part1.

The result is that for many functions, the derivative can be automatically computed. We will show this for a vector valued function $\mathbb{R}^2 \to \mathbb{R}^2$.

The first step is to import some functions from the package JAX. Notice that JAX has its own version of numpy. Here we import it as `jnp`.

[3]:
```
import jax.numpy as jnp
from jax import grad, jit, vmap
from jax import jacfwd, jacrev

# depending on the application, more imports are needed
```

JAX implements forward and reverse mode automatic differentiation. The commands are `jacfwd` and `jacrev` (jac stands for Jacobian). The wikipedia page on automatic differentiation briefly introduces forward and reverse mode automatic differentiation. Here we just mention that forward accumulation is more efficient than reverse accumulation for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \gg n$ as only n sweeps are necessary, compared to m sweeps for reverse accumulation and that reverse accumulation is more efficient than forward accumulation for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \ll n$ as only n sweeps are necessary.

In the following cell a simple function is defined and differentiated. Note that JAX has its own array type, `jax.numpy.array` (because of the line `import jax.numpy as jnp` we write this as `jnp.array`), that output is in 32 bits floating point format in a different array type `DeviceArray` (JAX has a preference for the 32 bits floating point format and you are allowed to use it in this exercise).

[4]:
```
def circle(x): return x[0]**2 + x[1]**2
J = jacfwd(circle)
J(jnp.array([1.0 ,2.0]))
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set TF_CPP_MIN_LOG_LEVEL=0
and rerun for more info.)
```

[4]:
```
DeviceArray([2., 4.], dtype=float32)
```

It is allowed to take derivatives of derivatives, so that a second derivative matrix can be obtained.

[5]:
```
def hessian(f): return jacfwd(jacrev(f))
H = hessian(circle)
myMatrix = H(jnp. array ([1.0 ,2.0]) )
print(myMatrix)
```

```
[[2. 0.]
 [0. 2.]]
```

Although this is not the standard `numpy.ndarray` format, it appears however that this format can be used in linear algebra operations such as solve.

[6]:
```
myVector = jnp.array([0.5, 2.0])

import scipy.linalg as la
la.solve(myMatrix, myVector)
```

[6]:
```
array([0.25, 1.  ], dtype=float32)
```

It is easy to define vector valued functions, by returning an `jax.numpy.array` object.

When using functions such as sin and cos and exp one must be careful. One must use the functions `jax.numpy.sin`, `jax.numpy.cos`, etc. (and not `math.sin` etc.). We define a test function $f(x_1, x_2) = [x_1 \exp(x_2), x_1 + x_2]$ using exp and show that it can be differentiated. Note that the derivative matrix is $\begin{bmatrix} \exp(x_2) & x_1 \exp(x_2) \\ 1 & 1 \end{bmatrix}$.

[7]:
```
def f(x):
    return jnp.array([x[0] * jnp.exp(x[1]), x[0] + x[1]])

print("values:")
print(f(jnp.array([2.0,1.0])))

Df = jacfwd(f)
print("jacobian matrix:")
print(Df(jnp.array([2.0,1.0])))
```

```
values:
[5.4365635 3.       ]
jacobian matrix:
[[2.7182817 5.4365635]
 [1.        1.       ]]
```

4 Exercise 2 (rootfinding with automatic differentiation, 3 points)

4.1 (a)

Create a Python function to apply Newton's method in multiple dimensions. Create a stopping criterion, such that your method automatically stops when one of the following conditions is satisfied: (i) the size of the function is below a specified tolerance; (ii) the difference in two subsequent iterates $\mathbf{x}_k$ is below a specified tolerance; (iii) the number of iterations reaches a specified limit.

[8]:
```
def Newton(fun, x, iterations, *args):
  """
  Apply Newtons method on a given function.
  """
  # Do Base cases
  x_ks = [x]
  Df = jacfwd(fun)
  for i in range(iterations):
    # Do iterative steps of Newton method
```

```
    s_k = la.solve(Df(x_ks[-1], *args), -fun(x_ks[-1], *args))
    x_ks.append(x_ks[-1] + s_k)

    # Check the length of f(x_{k-1}), stopping condition i
    if la.norm(fun(x_ks[-1], *args)) < 0.000001:
      print("Size of function is below tolerance.")
      break

    # Checks the difference in all positions seperately, stopping condition ii
    if la.norm(x_ks[-1] - x_ks[-2]) < 0.000001:#[diff for diff in x_difference
↪if diff < 0.000001]:
      print("The difference in two subsequent iterations is below tolerance.")
      break

  # Stopping condition iii
  if len(x_ks) == iterations + 1:
    print("The number of iterations was reached.")

  # Print solution
  print("Solutions of x are:")
  for i in range(len(x_ks[-1])):
    print(f"x{i} = {round(x_ks[-1][i],4)}")
  return x_ks

# x = jnp.array([1.,2.])
# plt.plot(Newton(f, x, 10))
# plt.show()
```

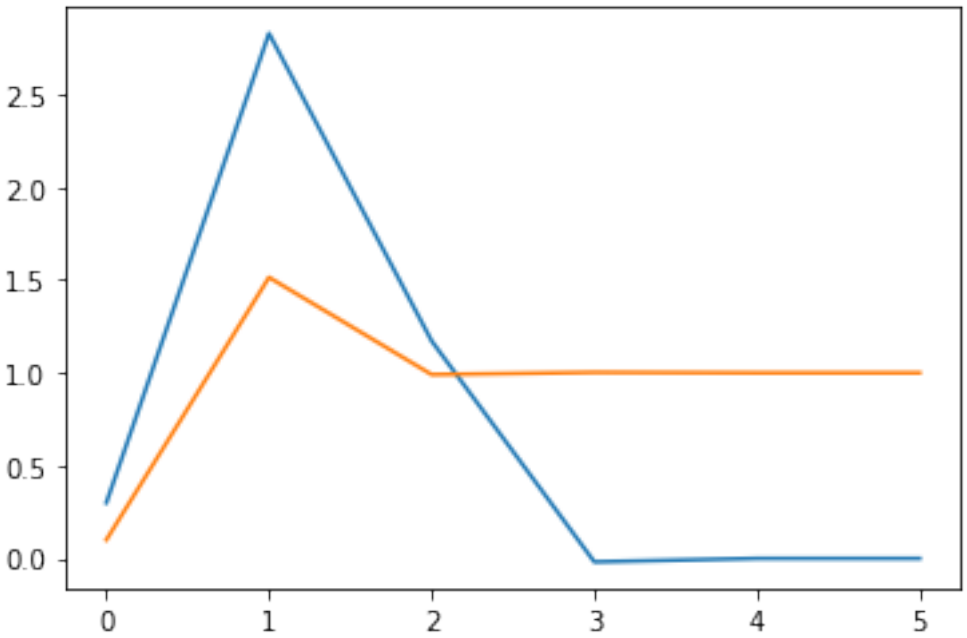
4.2 (b)

Solve Computer Exercise 5.19 using Newton's method and automatic differentiation. (N.B. Do not choose the starting point equal to a solution.)

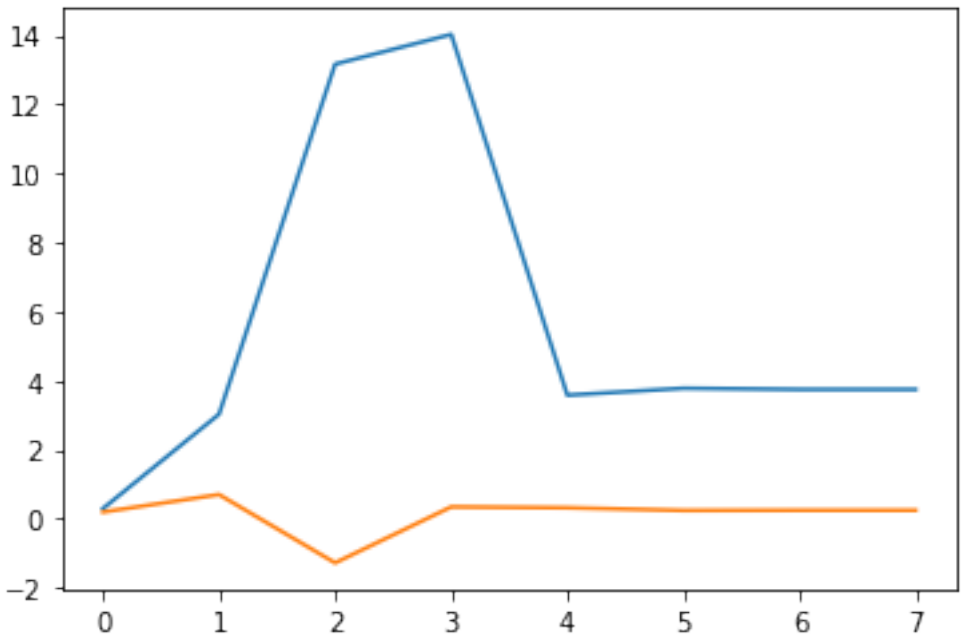
[9]:
```
def bio_fun(x, gamma, delta):
  """
  Return function values for given nonlinear equation.
  """
  return jnp.array([gamma * x[0] * x[1] - x[0] * (1 + x[1]),
                    -x[0] * x[1] + (delta - x[1]) * (1 + x[1])])

# Set parameters
gamma = 5
delta = 1

# Find a solution that ends with a bacterial density of 0
print("--Bacterial density x0 ends at zero--")
x = jnp.array([.3,.1])
```

```
plt.plot(Newton(bio_fun, x, 10, gamma, delta))
plt.show()

# Find a solution that ends with a bacterial density of not 0
print("\n--Bacterial density x0 does not end at zero--")
x = jnp.array([.3,.2])
plt.plot(Newton(bio_fun, x, 10, gamma, delta))
plt.show()
```

```
--Bacterial density x0 ends at zero--
Size of function is below tolerance.
Solutions of x are:
x0 = 0.0
x1 = 1.0
```




```
--Bacterial density x0 does not end at zero--
Size of function is below tolerance.
Solutions of x are:
x0 = 3.75
x1 = 0.25
```



With the values for $\gamma = 5$ and $\delta = 1$ the Newton method finds an $x_k = [x_0, x_1]$ of $[0, -1]$ for starting value $[0.3, 0.1]$, i.e. the bacterial population dies out. The Newton method finds an $x_k = [x_0, x_1]$ of $[3.75, 0.25]$ for starting value $[0.3, 0.2]$, i.e. the bacterial population does not die out.

5 Exercise 3 (3 points)

5.1 (a)

Consider the system

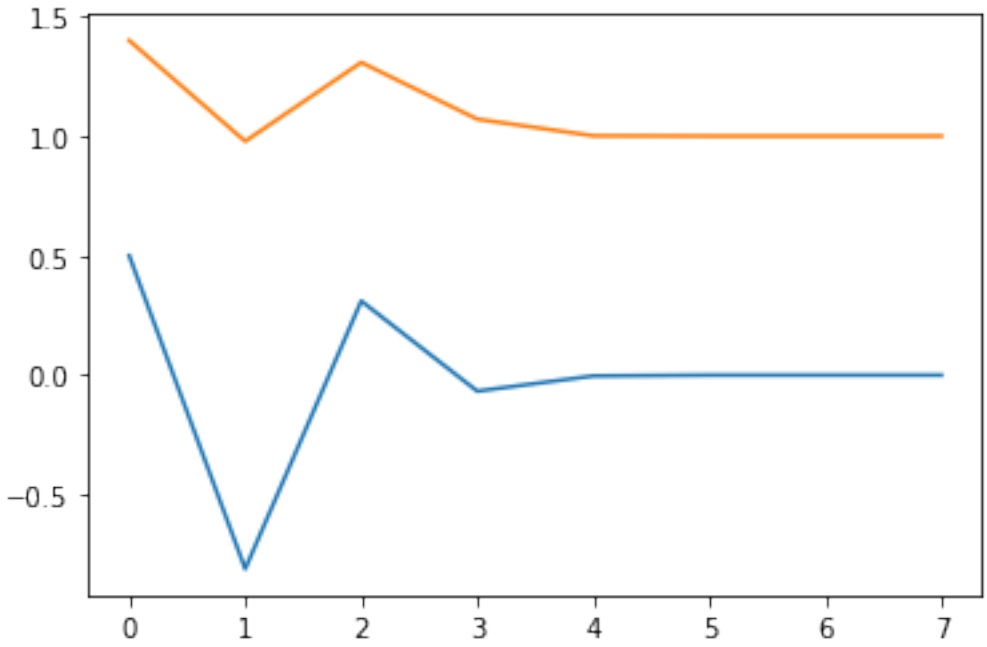
$$(x_1 + 3)(x_2^3 - 7) + 18 = 0$$
$$\sin(x_2 e^{x_1} - 1) = 0.$$

Solve this system using Newton's method with starting point $\mathbf{x}_0 = [0.5 \;\; 1.4]^T$.

[10]:
```
# your answer here
def fun3(x):
  return jnp.array([(x[0] + 3)*(x[1]**3 - 7) + 18, jnp.sin(x[1]*jnp.exp(x[0]) -␣
 ↪1)])

x = jnp.array([0.5, 1.4])
newton = Newton(fun3, x, 10)
plt.plot(newton)
plt.show()
```

```
Size of function is below tolerance.
Solutions of x are:
```

```
x0 = 0.0
x1 = 1.0
```



5.2 (b)

Write a program based on Broyden's method to solve the same system with the same starting point.

[11]:
```
def Broyden(fun,x0,iterations):
  b0 = jacfwd(fun)
  x_ks = [x0]
  y_ks = [fun(x0)]
  B_ks = [b0(x_ks[-1])]
  for k in range(iterations):
    sk = la.solve(B_ks[-1], -fun(x_ks[-1]))
    x_k = x_ks[-1] + sk
    y_k = fun(x_k) - fun(x_ks[-1])
    B_k = B_ks[-1] + (np.outer((y_k - B_ks[-1].dot(sk)),(sk.T)) / (sk.T.
↪dot(sk)))

    # Check the length of f(x_{k-1})
    if la.norm(fun(x_k)) < 0.000001:
      print("Size of function is below tolerance.")
      break

    # How to calculate difference between vectors?
    # Checks the difference in all positions seperately
```

```
    x_difference = np.abs(x_k - x_ks[-1])
    if [diff for diff in x_difference if diff < 0.000001]:
      print("The difference in two subsequent iterations is below tolerance.")
      break

    B_ks.append(B_k)
    x_ks.append(x_k)

  if len(x_ks) == iterations + 1:
    print("The number of iterations was reached.")

  print('x_k =', x_ks[-1])
  return x_ks
```

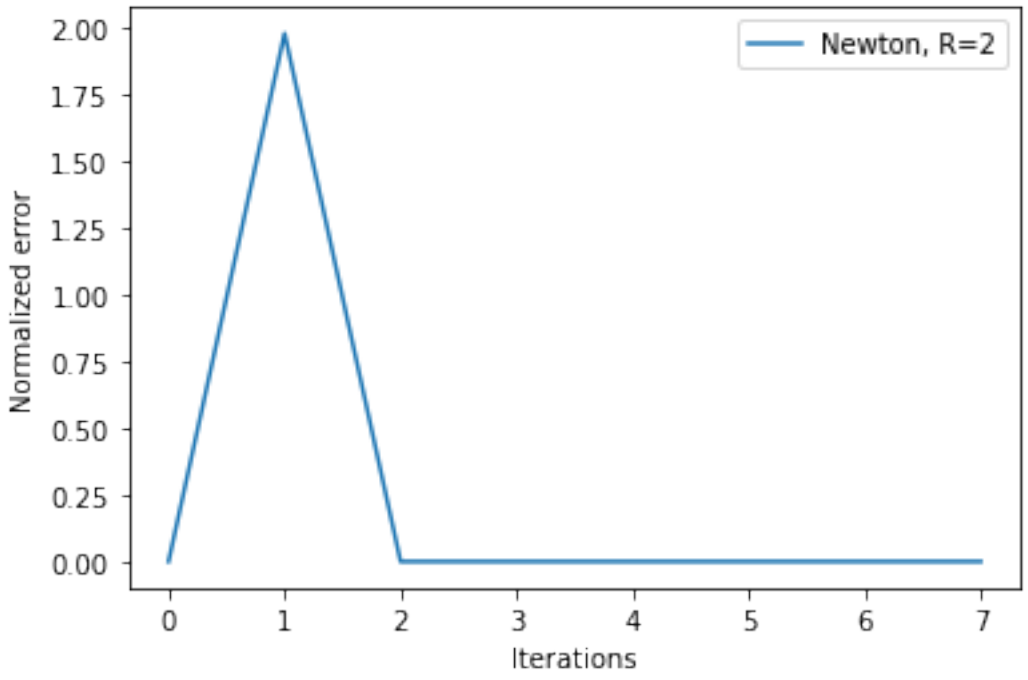
5.3 (c)

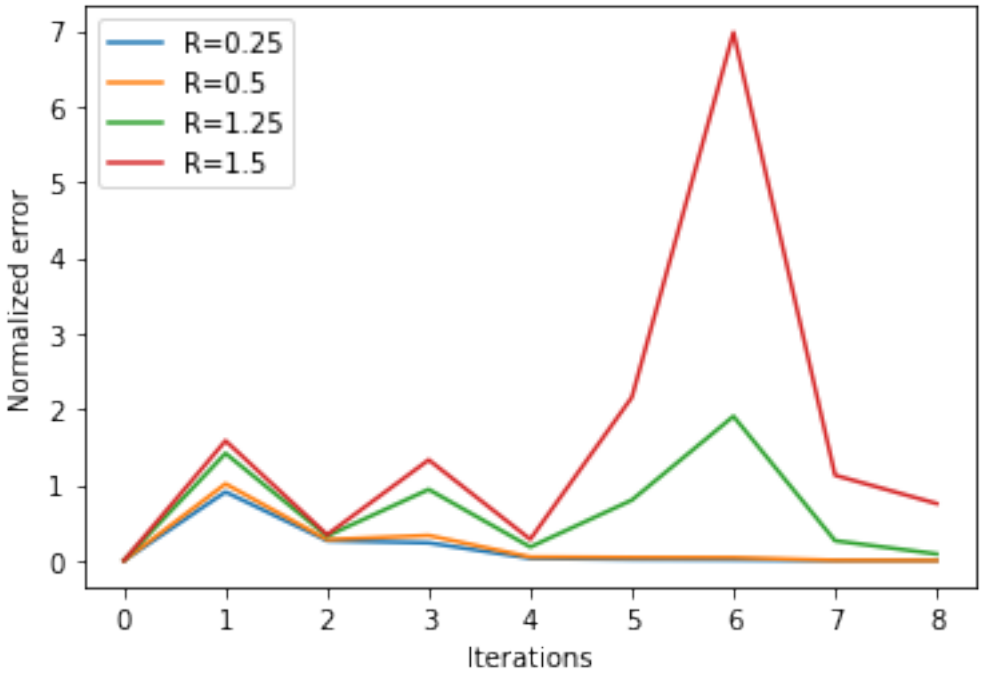
Compare the convergence rates of the two methods by computing the error at each iteration and appropriately analysing these errors, given that the exact solution is $\mathbf{x}^* = [0 \;\; 1]^T$.

[13]:
```
x = jnp.array([.5,1.4])
broyden = Broyden(fun3,x,20)
# plt.plot(broyden)
# plt.show()


def difference_N():
  difference_newton = np.zeros((len(newton), 1))
  x_star = np.array([0,1])
  for i in range(1,len(newton)):
    d_n0 = newton[i-1] - x_star
    d_n1 = newton[i] - x_star
    difference_newton[i] = np.linalg.norm(d_n1)/ (np.linalg.norm(d_n0)**2)
    return difference_newton

def difference_B():
  difference_broyden_1 = np.zeros((len(broyden), 1))
  difference_broyden_2 = np.zeros((len(broyden), 1))
  difference_broyden_3 = np.zeros((len(broyden), 1))
  difference_broyden_4 = np.zeros((len(broyden), 1))
  x_star = np.array([0,1])
  for i in range(1,len(broyden)):
    d_b0 = broyden[i-1] - x_star
    d_b1 = broyden[i] - x_star
    difference_broyden_1[i] = np.linalg.norm(d_b1)/ np.linalg.norm(d_b0)**0.25
    difference_broyden_2[i] = np.linalg.norm(d_b1)/ np.linalg.norm(d_b0)**0.5
    difference_broyden_3[i] = np.linalg.norm(d_b1)/ np.linalg.norm(d_b0)**1.25
    difference_broyden_4[i] = np.linalg.norm(d_b1)/ np.linalg.norm(d_b0)**1.5
```

```
  return␣
↪difference_broyden_1,difference_broyden_2,difference_broyden_3,difference_broyden_4

plt.figure()
plt.plot(difference_N(),label='Newton, R=2')
plt.legend()
plt.xlabel('Iterations')
plt.ylabel('Normalized error')
plt.show()
plt.plot(difference_B()[0],label='R=0.25')
plt.plot(difference_B()[1],label='R=0.5')
plt.plot(difference_B()[2],label='R=1.25')
plt.plot(difference_B()[3],label='R=1.5')
plt.xlabel('Iterations')
plt.ylabel('Normalized error')
plt.legend()
plt.show()
# plt.plot(difference_newton - difference_broyden)
# # plt.yscale("symlog")
# plt.show()
```

```
Size of function is below tolerance.
x_k = [1.5986298e-06 9.9999928e-01]
```





We notice that the newton function does converge to zero for a quadratic normalization. The broyden function however is not quadratic and therefore requires a different value of R. If we plot the function for 4 different values we observe that the value of R must be somewhere between $0.25 < R < 1.25$

[12]: