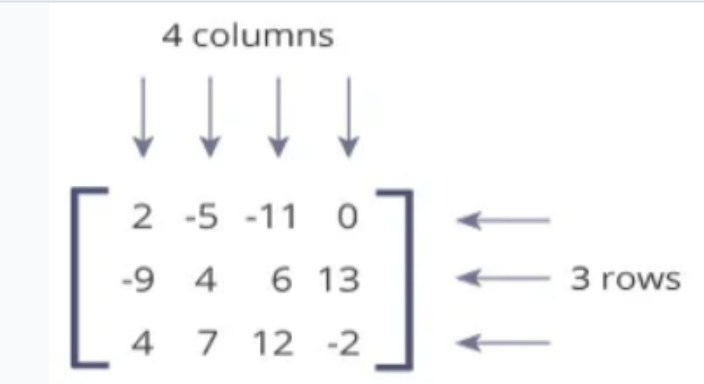
NumPy array handling for 3rd Semester B.Sc. python

Introduction

A matrix is a two-dimensional data structure where numbers are arranged into rows and columns. For example:



This matrix is a 3×4 (pronounced “three by four”) matrix because it has 3 rows and 4 columns.

All the codes, books, and documents related to this document are stored in the following GitHub repository.

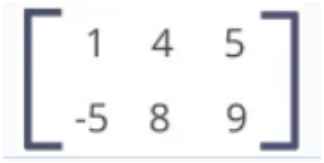
https://github.com/paramphy/3rd-Semester

Python Matrix

Python doesn't have a built-in type for matrices. However, we can treat a list of a list as a matrix. For example:

```
A = [[1, 4, 5],
    [-5, 8, 9]]
```

We can treat this list of a list as a matrix having 2 rows and 3 columns.



Let's see how to work with a nested list.

```
A = [[1, 4, 5, 12],
    [-5, 8, 9, 0],
    [-6, 7, 11, 19]]

print("A =", A)
print("A[1] =", A[1])      # 2nd row
print("A[1][2] =", A[1][2])   # 3rd element of 2nd row
print("A[0][-1] =", A[0][-1])   # Last element of 1st Row

column = [];        # empty list
for row in A:
  column.append(row[2])

print("3rd column =", column)
```

When we run the program, the output will be:

```
A = [[1, 4, 5, 12], [-5, 8, 9, 0], [-6, 7, 11, 19]]
A[1] = [-5, 8, 9, 0]
A[1][2] = 9
A[0][-1] = 12
3rd column = [5, 9, 11]
```

NumPy Array

NumPy is a package for scientific computing which has support for a powerful N-dimensional array object. Before you can use NumPy, you need to install it. For more info,

- Visit: How to install NumPy?
- If you are on Windows, download and install anaconda distribution of Python. It comes with NumPy and other several packages related to data science and machine learning.

Once NumPy is installed, you can import and use it.

NumPy provides a multidimensional array of numbers (which is actually an object). Let's take an example:

```
import numpy as np
a = np.array([1, 2, 3])
print(a)               # Output: [1, 2, 3]
print(type(a))         # Output: <class 'numpy.ndarray'>
```

As you can see, NumPy's array class is called **ndarray**.

How to create a NumPy array?

There are various ways to create NumPy arrays.

Array of integers, floats, and complex Numbers

```
import numpy as np

A = np.array([[1, 2, 3], [3, 4, 5]])
print(A)

A = np.array([[1.1, 2, 3], [3, 4, 5]]) # Array of floats
print(A)

A = np.array([[1, 2, 3], [3, 4, 5]], dtype = complex) # Array of complex numbers
print(A)
```

When you run the program, the output will be:

```
[[1 2 3]
 [3 4 5]]

[[1.1 2.  3. ]
[3.  4.  5. ]]

[[1.+0.j 2.+0.j 3.+0.j]
 [3.+0.j 4.+0.j 5.+0.j]]
```

Array of zeros and ones

```
import numpy as np

zeors_array = np.zeros( (2, 3) )
print(zeors_array)

'''
 Output:
 [[0. 0. 0.]
  [0. 0. 0.]]
'''

ones_array = np.ones( (1, 5), dtype=np.int32 ) // specifying dtype
print(ones_array)      # Output: [[1 1 1 1 1]]
```

Here, we have specified dtype to 32 bits (4 bytes). Hence, this array can take values from -2^{31} to $2^{31} - 1$.

Using arange() and shape()

```
import numpy as np

A = np.arange(4)
print('A =', A)

B = np.arange(12).reshape(2, 6)
print('B =', B)
```

Output of the code

```
A = [0 1 2 3]
B = [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```

Learn more about other ways of creating a NumPy array.

Matrix Operations

Above, we gave you 3 examples: addition of two matrices, multiplication of two matrices and transpose of a matrix. We used nested lists before to write those programs. Let's see how we can do the same task using NumPy array.

Addition of Two Matrices

We use **+** operator to add corresponding elements of two NumPy matrices.

```
import numpy as np

A = np.array([[2, 4], [5, -6]])
B = np.array([[9, -3], [3, 6]])
C = A + B      # element wise addition
print(C)
```

Output of the code

```
[[11  1]
 [ 8  0]]
```

Multiplication of Two Matrices

To multiply two matrices, we use dot() method. Learn more about how numpy.dot works.

Note: `*` is used for array multiplication (multiplication of corresponding elements of two arrays) not matrix multiplication.

```
import numpy as np

A = np.array([[3, 6, 7], [5, -3, 0]])
B = np.array([[1, 1], [2, 1], [3, -3]])
C = A.dot(B)
print(C)
```

Output of the code

```
[[ 36 -12]
 [ -1   2]]
```

Transpose of a Matrix

We use numpy.transpose to compute transpose of a matrix.

```
import numpy as np

A = np.array([[1, 1], [2, 1], [3, -3]])
print(A.transpose())
```

Output of the code

```
[[ 1  2  3]
 [ 1  1 -3]]
```

As you can see, NumPy made our task much easier.

Access matrix elements, rows, and columns

Access matrix elements

Similar like lists, we can access matrix elements using index. Let's start with a one-dimensional NumPy array.

```
import numpy as np
A = np.array([2, 4, 6, 8, 10])

print("A[0] =", A[0])     # First element
```

```
print("A[2] =", A[2])     # Third element
print("A[-1] =", A[-1])   # Last element
```

When you run the program, the output will be:

```
A[0] = 2
A[2] = 6
A[-1] = 10
```

Now, let's see how we can access elements of a two-dimensional array (which is basically a matrix).

```
import numpy as np

A = np.array([[1, 4, 5, 12],
    [-5, 8, 9, 0],
    [-6, 7, 11, 19]])

#  First element of first row
print("A[0][0] =", A[0][0])

# Third element of second row
print("A[1][2] =", A[1][2])

# Last element of last row
print("A[-1][-1] =", A[-1][-1])
```

When we run the program, the output will be:

```
A[0][0] = 1
A[1][2] = 9
A[-1][-1] = 19
```

Access rows of a Matrix

```
import numpy as np

A = np.array([[1, 4, 5, 12],
    [-5, 8, 9, 0],
    [-6, 7, 11, 19]])

print("A[0] =", A[0]) # First Row
print("A[2] =", A[2]) # Third Row
print("A[-1] =", A[-1]) # Last Row (3rd row in this case)
```

When we run the program, the output will be:

```
A[0] = [1, 4, 5, 12]
A[2] = [-6, 7, 11, 19]
```

```
A[-1] = [-6, 7, 11, 19]
```

Access columns of a Matrix

```
import numpy as np

A = np.array([[1, 4, 5, 12],
    [-5, 8, 9, 0],
    [-6, 7, 11, 19]])

print("A[:,0] =",A[:,0]) # First Column
print("A[:,3] =", A[:,3]) # Fourth Column
print("A[:,-1] =", A[:,-1]) # Last Column (4th column in this case)
```

When we run the program, the output will be:

```
A[:,0] = [ 1 -5 -6]
A[:,3] = [12  0 19]
A[:,-1] = [12  0 19]
```

If you don't know how this above code works, read slicing of a matrix section of this article.

Slicing of a Matrix

Slicing of a one-dimensional NumPy array is similar to a list. If you don't know how slicing for a list works, visit Understanding Python's slice notation.

Let's take an example:

```
import numpy as np
letters = np.array([1, 3, 5, 7, 9, 7, 5])

# 3rd to 5th elements
print(letters[2:5])        # Output: [5, 7, 9]

# 1st to 4th elements
print(letters[:-5])        # Output: [1, 3]

# 6th to last elements
print(letters[5:])         # Output:[7, 5]

# 1st to last elements
print(letters[:])          # Output:[1, 3, 5, 7, 9, 7, 5]

# reversing a list
print(letters[::-1])          # Output:[5, 7, 9, 7, 5, 3, 1]
```

Now, let's see how we can slice a matrix.

```
import numpy as np

A = np.array([[1, 4, 5, 12, 14],
    [-5, 8, 9, 0, 17],
    [-6, 7, 11, 19, 21]])

print(A[:2, :4])  # two rows, four columns

''' Output:
[[ 1  4  5 12]
 [-5  8  9  0]]
'''


print(A[:1,])  # first row, all columns

''' Output:
[[ 1  4  5 12 14]]
'''

print(A[:,2])  # all rows, second column

''' Output:
[ 5  9 11]
'''

print(A[:, 2:5])  # all rows, third to the fifth column

'''Output:
[[ 5 12 14]
 [ 9  0 17]
 [11 19 21]]
'''
```

As you can see, using NumPy (instead of nested lists) makes it a lot easier to work with matrices, and we haven't even scratched the basics. We suggest you to explore NumPy package in detail, especially if you are trying to use Python for data science/analytics.

NumPy Resources you might find helpful:

- NumPy Tutorial
- NumPy Reference

Gauss elimination method to solve linear equations and evaluating matrix determinant.

Pivoting and solution of linear equation

Solve

$$3x + 2y + z = 11$$
$$2x + 3y + z = 13$$
$$x + y + 4z = 12$$

The determinant of the inhomogeneous linear equations is 18, so a solution exists. For convenience and for the optimum numerical accuracy, the equations are rearranged so that the largest coefficients run along the main diagonal (upper left to lower right). This has already been done in the preceding set.

The Gauss technique is to use the first equation to eliminate the first unknown, x, from the remaining equations. Then the (new) second equation is used to eliminate y from the last equation. In general, we work down through the set of equations, and then, with one unknown determined, we work back up to solve for each of the other unknowns in succession.

Dividing each row by its initial coefficient, we see that Eqs. (3.19) become

$$x + \frac{2}{3}y + \frac{1}{3}z = \frac{11}{3}$$
$$x + \frac{3}{2}y + \frac{1}{2}z = \frac{13}{2}$$
$$x + y + 4z = 12$$

Now, using the first equation, we eliminate x from the second and third equations:

$$x + \frac{2}{3}y + \frac{1}{3}z = \frac{11}{3}$$
$$\frac{5}{6}y + \frac{1}{6}z = \frac{17}{6}$$
$$\frac{1}{3}y + \frac{11}{3}z = \frac{25}{3}$$

and

$$x + \frac{2}{3}y + \frac{1}{3}z = \frac{11}{3}$$
$$y + \frac{1}{5}z = \frac{17}{5}$$
$$y + 11z = 25.$$

Repeating the technique, we use the new second equation to eliminate y from the third equation:

$$x + \frac{2}{3}y + \frac{1}{3}z = \frac{11}{3}$$
$$y + \frac{1}{5}15z = \frac{17}{5}$$
$$54z = 108$$
$$or,$$
$$z = 2$$

Finally, working back up, we get

$$y + 15 \times 2 = \frac{17}{5},$$
$$or$$
$$y = 3.$$

Then with z and y determined,

$$x + \frac{2}{3} \times 3 + \frac{1}{3} \times 2 = \frac{11}{3},$$
$$and$$
$$x = 1.$$

Python code for the above problem

Python Source Code: Gauss Elimination Method

```
# Importing NumPy Library
import numpy as np
import sys
n = 3
# Making numpy array of n x n+1 size and initializing
# to zero for storing augmented matrix
a = np.array([[3.0,2.0,1.0,11.0],[2.0,3.0,1.0,13.0],[1.0,1.0,4.0,12.0]])

# Making numpy array of n size and initializing
# to zero for storing solution vector
x = np.zeros(n)

print(a)
print(x)
# Applying Gauss Elimination
for i in range(n):
    if a[i][i] == 0.0:
        sys.exit('Divide by zero detected!')

    for j in range(i+1, n):
        ratio = a[j][i]/a[i][i]

        for k in range(n+1):
            a[j][k] = a[j][k] - ratio * a[i][k]
```

```
# Back Substitution
x[n-1] = a[n-1][n]/a[n-1][n-1]

for i in range(n-2,-1,-1):
    x[i] = a[i][n]

    for j in range(i+1,n):
        x[i] = x[i] - a[i][j]*x[j]

    x[i] = x[i]/a[i][i]

# Displaying solution
print('\nRequired solution is: ')
for i in range(n):
    print('X%d = %0.2f' %(i,x[i]), end = '\t')
```

Result

```
[[ 3.  2.  1. 11.]
 [ 2.  3.  1. 13.]
 [ 1.  1.  4. 12.]]
[0. 0. 0.]

Required solution is:
X0 = 1.00       X1 = 3.00       X2 = 2.00
```

Code for an arbitrary matrix

```
# Importing NumPy Library
import numpy as np
import sys

# Reading number of unknowns
n = int(input('Enter number of unknowns: '))

# Making numpy array of n x n+1 size and initializing
# to zero for storing augmented matrix
a = np.zeros((n,n+1))

# Making numpy array of n size and initializing
# to zero for storing solution vector
x = np.zeros(n)

# Reading augmented matrix coefficients
print('Enter Augmented Matrix Coefficients:')
for i in range(n):
    for j in range(n+1):
        a[i][j] = float(input( 'a['+str(i)+']['+ str(j)+']='))
print(a)
# Applying Gauss Elimination
for i in range(n):
    if a[i][i] == 0.0:
        sys.exit('Divide by zero detected!')
```

```
    for j in range(i+1, n):
        ratio = a[j][i]/a[i][i]

        for k in range(n+1):
            a[j][k] = a[j][k] - ratio * a[i][k]

# Back Substitution
x[n-1] = a[n-1][n]/a[n-1][n-1]

for i in range(n-2,-1,-1):
    x[i] = a[i][n]

    for j in range(i+1,n):
        x[i] = x[i] - a[i][j]*x[j]

    x[i] = x[i]/a[i][i]

# Displaying solution
print('\nRequired solution is: ')
for i in range(n):
    print('X%d = %0.2f' %(i,x[i]), end = '\t')
```

Evaluating matrix determinant

Alternatives approaches inspired by Gauss elimination

Gauss-Jordon method

In linear algebra, **Gauss Jordan Method** is a procedure for solving systems of linear equation. It is also known as **Row Reduction Technique**. In this method, the problem of systems of linear equation having n unknown variables, matrix having rows n and columns n+1 is formed. This matrix is also known as **Augmented Matrix**. After forming **n x n+1** matrix, the matrix is transformed to **diagonal matrix by row operations**. Finally, the result is obtained by making all diagonal element to 1 i.e. identity matrix.

Algorithm for Gauss Jordan Method

```
1. Start
2. Read Number of Unknowns: n
3. Read Augmented Matrix (A) of n by n+1 Size
4. Transform Augmented Matrix (A) to Diagonal Matrix by Row Operations.
5. Obtain Solution by Making All Diagonal Elements to 1.
6. Display Result.
7. Stop
```

This python program solves systems of linear equation with n unknowns using **Gauss Jordan Method.**

In Gauss Jordan method, given system is first transformed to **Diagonal Matrix** by row operations, then solution is obtained by directly.

```
# Importing NumPy Library
import numpy as np
import sys

# Reading number of unknowns
n = int(input('Enter number of unknowns: '))

# Making numpy array of n x n+1 size and initializing
# to zero for storing augmented matrix
a = np.zeros((n,n+1))

# Making numpy array of n size and initializing
# to zero for storing solution vector
x = np.zeros(n)

# Reading augmented matrix coefficients
print('Enter Augmented Matrix Coefficients:')
for i in range(n):
    for j in range(n+1):
        a[i][j] = float(input( 'a['+str(i)+']['+ str(j)+']='))

# Applying Gauss Jordan Elimination
for i in range(n):
    if a[i][i] == 0.0:
        sys.exit('Divide by zero detected!')

    for j in range(n):
        if i != j:
            ratio = a[j][i]/a[i][i]

            for k in range(n+1):
                a[j][k] = a[j][k] - ratio * a[i][k]

# Obtaining Solution

for i in range(n):
    x[i] = a[i][n]/a[i][i]

# Displaying solution
print('\nRequired solution is: ')
for i in range(n):
    print('X%d = %0.2f' %(i,x[i]), end = '\t')
```

Result

```
Enter number of unknowns: 3
Enter Augmented Matrix Coefficients:
a[0][0]=3
```

```
a[0][1]=2
a[0][2]=1
a[0][3]=11
a[1][0]=2
a[1][1]=3
a[1][2]=1
a[1][3]=10
a[2][0]=1
a[2][1]=1
a[2][2]=4
a[2][3]=55

Required solution is:
X0 = -0.22      X1 = -1.22      X2 = 14.11
```

Gauss-Seidel method

Gauss Seidel method is an iterative approach for solving system of linear equations. In this method, first given system of linear equations are arranged in diagonally dominant form. For guaranteed convergence, the system must be in Diagonally Dominant Form. In this article, we are going to develop an algorithm for Gauss Seidel method.

Gauss Seidel Iterative Method Algorithm

```
1. Start
2. Arrange given system of linear equations in diagonally dominant form
3. Read tolerable error (e)
4. Convert the first equation in terms of first variable,second equation in terms of second variable an
d so on.
5. Set initial guesses for x0,  y0, z0 and so on
6. Substitute value of y0, z0 ... from step 5 in first equation obtained from step 4 to calculate new v
alue of x1. Use x1, z0, u0 .... in second equation obtained from step 4 to caluclate new value of y1. S
imilarly, use x1, y1, u0... to find new z1 and so on.
7. If| x0 - x1| > e and | y0 - y1| > e and | z0 - z1|  > e and so on then goto step 9
8. Set x0=x1, y0=y1, z0=z1 and so on and goto step 6
9. Print value of x1, y1, z1 and so on
10. Stop
```

This program implements Gauss Seidel Iteration Method for solving systems of linear equation in python programming language.

In Gauss Seidel method, we first arrange a given system of linear equations in diagonally dominant form. For example, if a system of linear equations are:

$$3x + 20y - z = -18$$
$$2x - 3y + 20z = 25$$
$$20x + y - 2z = 17$$

Then they will be arranged like this:

$$20x + y - 2z = 17$$
$$3x + 20y - z = -18$$
$$2x - 3y + 20z = 25$$

After arranging equations in diagonally dominant form, we form equations for x, y, & z like this:

$$x = (17 - y + 2z)/20$$
$$y = (-18 - 3x + z)/20$$
$$z = (25 - 2x + 3y)/20$$

These equations are defined later in Gauss Seidel python program using `lambda` expression.

Python Source Code: Gauss Seidel Method

```
# Gauss Seidel Iteration

# Defining equations to be solved
# in diagonally dominant form
f1 = lambda x,y,z: (17-y+2*z)/20
f2 = lambda x,y,z: (-18-3*x+z)/20
f3 = lambda x,y,z: (25-2*x+3*y)/20

# Initial setup
x0 = 0
y0 = 0
z0 = 0
count = 1

# Reading tolerable error
e = float(input('Enter tolerable error: '))

# Implementation of Gauss Seidel Iteration
print('\nCount\tx\ty\tz\n')

condition = True

while condition:
    x1 = f1(x0,y0,z0)
    y1 = f2(x1,y0,z0)
    z1 = f3(x1,y1,z0)
    print('%d\t%0.4f\t%0.4f\t%0.4f\n' %(count, x1,y1,z1))
    e1 = abs(x0-x1);
    e2 = abs(y0-y1);
    e3 = abs(z0-z1);

    count += 1
    x0 = x1
    y0 = y1
    z0 = z1

    condition = e1>e and e2>e and e3>e

print('\nSolution: x=%0.3f, y=%0.3f and z = %0.3f\n'% (x1,y1,z1))
```

Result

```
Enter tolerable error: 0.00001

Count   x       y       z

1       0.8500  -1.0275 1.0109

2       1.0025  -0.9998 0.9998

3       1.0000  -1.0000 1.0000

4       1.0000  -1.0000 1.0000


Solution: x=1.000, y=-1.000 and z = 1.000
```

Linear algebra

Python’s mathematical libraries, NumPy and SciPy, have extensive tools for numerically solving problems in linear algebra. Here, we focus on two problems that arise commonly in scientific and engineering settings:

(1) solving a system of linear equations

(2) eigenvalue problems.

In addition, we also show how to perform a number of other basic computations, such as finding the determinant of a matrix, matrix inversion, and LU decomposition. The SciPy package for linear algebra is called scipy.linalg

Basic computations in linear algebra

NumPy has a number of routines for performing basic operations with matrices. The determinant of a matrix is computed using the scipy.linalg.det function:

```
import numpy as np

a = np.array([[-2,3],[4,5]])
print(a)
det = np.linalg.det(a)
print(det)
```

Result

```
[[-2  3] [ 4  5]]
-22.000000000000004
```

The inverse of a matrix is computed using the numpy.linalg.inv function, while the product of two matrices is calculated using the NumPy dot function:

```
import numpy as np

a = np.array([[-2,3],[4,5]])
b = np.array([[-0.22727273,  0.13636364],
              [ 0.18181818,  0.09090909]])
inverse = np.linalg.inv(a)
print(inverse)
dotproduct = np.dot(a, b)
print(dotproduct)
```

Result

```
[[-0.22727273  0.13636364]
 [ 0.18181818  0.09090909]]
[[ 1.00000000e+00 -1.00000000e-08]
 [-2.00000000e-08  1.00000001e+00]]
```

Solving systems of linear equations

Solving systems of equations is nearly as simple as constructing a coefficient matrix and a column vector. Suppose you have the following system of linear equations to solve:

The first task is to recast this set of equations as a matrix equation of the form. Finally we use the SciPy function numpy.linalg.solve to find

```
import numpy as np

A = np.array([[2, 4, 6], [1, -3, -9], [8, 5, -7]])
b = np.array([4, -11, 2])
solution = np.linalg.solve(A,b)
print(solution)
```

```
[-8.91304348 10.2173913  -3.17391304]
```

Another approach

Following this approach, we can use the scipy.linalg.inv.

```
import numpy as np

A = np.array([[2, 4, 6], [1, -3, -9], [8, 5, -7]])
b = np.array([4, -11, 2])
```

```
# Another method
Ainv = np.linalg.inv(A)
solution = np.dot(Ainv,b)
print(solution)
```

Result

```
[-8.91304348 10.2173913  -3.17391304]
```

which is the same answer we obtained using numpy.linalg.solve. This method is numerically more stable and a faster than using numpy.linalg.solve, so it is the preferred method for solving systems of equations. You might wonder what happens if the system of equations are not all linearly independent. For example, if the matrix is given where the third row is a multiple of the first row. Let's try it out and see what happens. First we change the bottom row of the matrix and then try to solve the system as we did before.

```
import numpy as np

A = np.array([[ 2,  4,  6],
              [ 1, -3, -9],
              [ 1,  2,  3]])

b = np.array([4, -11, 2])
solution = np.linalg.solve(A,b)
# using alternate method
Ainv = np.linalg.inv(A)
```

Result

```
Traceback (most recent call last):
  File "d:/CLass files/B.Sc Python/3rd Semester/Codes/singualmatsolve.py", line 8, in <module>
    solution = np.linalg.solve(A,b)
  File "<__array_function__ internals>", line 5, in solve
  File "C:\Python3.8.10\lib\site-packages\numpy\linalg\linalg.py", line 393, in solve
    r = gufunc(a, b, signature=signature, extobj=extobj)
  File "C:\Python3.8.10\lib\site-packages\numpy\linalg\linalg.py", line 88, in _raise_linalgerror_singular
    raise LinAlgError("Singular matrix")
numpy.linalg.LinAlgError: Singular matrix
```

Whether we use numpy.linalg.solve or numpy.linalg.inv, NumPy raises an error because the matrix is singular.

Eigenvalue problems:

One of the most common problems in science and engineering is the eigenvalue problem, which in matrix form is written as where is a square matrix, is a column vector, and is a scalar (number).

Given the matrix, the problem is to find the set of eigenvectors and their corresponding eigenvalues, that solve this equation.

We can solve eigenvalue equations like this using numpy.linalg.eig. The outputs of this function is an array whose entries are the eigenvalues and a matrix whose rows are the eigenvectors. Let's return to the matrix we were using previously and find its eigenvalues and eigenvectors.

```
import numpy as np

A = np.array([[ 2,  4,  6],
              [ 1, -3, -9],
              [ 8,  5, -7]])

lam, evec = np.linalg.eig(A)
print(lam)
print(evec)
```

Result

```
[ 2.40995356 -8.03416016 -2.3757934 ]
[[-0.77167559 -0.52633654  0.57513303]
 [ 0.50360249  0.76565448 -0.80920669]
 [-0.38846018  0.36978786  0.12002724]]
```

The first eigenvalue and its corresponding eigenvector are given by

```
print(lam[0])
print(evec[:,0])
```

Result

```
2.4099535647625494
[-0.77167559  0.50360249 -0.38846018]
```

Hermitian and banded matrices

NumPy has a specialized routine for solving eigenvalue problems for Hermitian (or real symmetric) matrices. The routine for hermitian matrices is numpy.linalg.eigh. It is more efficient (faster and uses less memory) than numpy.linalg.eig. The basic syntax of the two routines is the same, though some o*ptional* arguments are different. Both routines can solve generalized and standard eigenvalue problems.