PYTHON CODE FROM TEXT "A FIRST COURSE IN NUMERICAL ANALYSIS".

M. AUTH

1. Chapter 1

1.1. Program: Example 1.2.

```
# Python Code for Example 1.2 on page 6.
import numpy as np
import matplotlib.pyplot as plt
from math import sin, cos

x0 = 1.2
f0 = sin(x0)
fp = cos(x0)
i = np.r_[-20:0.5:0.5]
h = 10**i
err = np.abs(fp - (np.sin(x0+h) - f0)/h)
d_err = f0/2*h
plt.loglog(h,err)
plt.loglog(h,d_err)
```

1.2. Program: Example 1.3.

```
# Python Code for Example 1.3 on page 8.
import numpy as np
from pylab import plot, show, loglog

x0 = 1.2
f0 = np.sin(x0)
fp = np.cos(x0)
i = np.arange(-20,0,0.5)
h = 10**i
err = np.abs(fp - (np.sin(x0 + h) -f0)/h)
d_err = f0/2*h
loglog(h,err)
loglog(h,d_err,"k--")
show()
```

1.3. **Program: Example 1.4.**

```
# EXAMPLE 1.4 on page 10.
# Evaluate p(7/2) when p(x) = -3x^4 + 2.2x^3 - 11x + 5/2
# using the onion shell method in Python

import numpy as np

x = 7/2
c = np.array([5/2, -11, 0, 2.2, -3], float)
n = 4

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

print(str(round(p,3)))
```

2. Chapter 3

2.1. **Program: BISECTION.**

```
import numpy as np
from math import cos, exp

def f(x):
    return exp(x) + x - 7

a,b = 1,2  # starting guesses to bound root x*

for k in range(18):
    x = (a+b)/2  # approximate root
    if f(a)*f(x) < 0:
        b = x
    else:
        a = x
    err_bnd = b - a
    print(k,str(round(x,12)),str(round(err_bnd,8)))
```

2.2. **Program: BISECTION (with tolerance).**

```
import numpy as np

def f(x):
    return x**3 - 30*x**2 + 2552

a,b = 0,20  # starting guesses to bound root x*
atol = 1e-8  # tolerance required in answer
k = 0  # computes number of steps

while abs(b-a) > atol:
    x = (a+b)/2  # approximate root
    if f(a)*f(x) < 0:
        b = x
    else:
        a = x
    err_bnd = b - a
    print(k,x,err_bnd)
    k += 1
```

2.3. **Program: FPI.**

```
from math import log

def g(x):
    return log(7-x)

x = 1 # initial guess
print(str(round(0)),str(round(x,6)))

for k in range(13):  # 13 steps
    x = g(x)
    print(str(round(k)),str(round(x,6)))  # print 6 digits each iteration
```

2.4. **Program: Newton's Method.**

```
def f(x):
    return x**2 - 2

def fPrime(x):
    return 2*x

x = 1 # initial guess
print(str(round(0)),str(round(x,14)))

for k in range(7):  # 7 iterations
    x = x - f(x)/fPrime(x)
    print(str(round(k)),str(round(x,14)))
```

2.5. **Program: Secant Method.**

```
def f(x):
    return x**2 - 2


a = 0, b = 1 # initial guesses
print(str(round(0)),str(round(a,8)))

for k in range(11):  # 11 iterations
    b = b - f(b)*(b - a)/(f(b) - f(a))
    print(str(round(k)),str(round(a,8)))
```

3. Chapter 5

3.1. Program: Gaussian Elimination Start.

```
# ALGORITHM: Gaussian Elimination from p.99

import numpy as np

A = np.array([[1, 2, 1],
             [3, 8 , 1],
             [0, 4, 1]])
n = 3 # size of A is 3x3

j = 0 # First Eliminate first column
for i in range(j+1,n): # loop through each row starting from the second
    l = (A[j+1,j]/A[j,j]) # first compute multiplier
    A[i,:] = A[i,:] - l*A[j,:] # subtract multiplier*row to eliminate

print(A)
```

3.2. Program: Gaussian Elimination.

```
# ALGORITHM: Gaussian Elimination from p.99

import numpy as np

A = np.array([[1, 2, 1],
             [3, 8 , 1],
             [0, 4, 1]])
n = 3 # size of A is 3x3

for j in range(n-1): # loop each column j from 0 to (n-2)
    for i in range(j+1,n): # loop through each row starting from the second
        l = (A[j+1,j]/A[j,j]) # first compute multiplier
        A[i,:] = A[i,:] - l*A[j,:] # subtract multiplier*row to eliminate

print(A)
```

3.3. **Program: LU decompostion.**

```
# ALGORITHM: LU decomposition from p.103

import numpy as np

A = np.array([[1, 2, 1],
             [3, 8 , 1],
             [0, 4, 1]])
n = 3 # size of A is 3x3
L = np.identity(n)

for j in range(n-1): # loop each column j from 0 to (n-2)
    for i in range(j+1,n): # loop through each row starting from the second
        L[i,j] = (A[j+1,j]/A[j,j]) # first compute multiplier
        A[i,:] = A[i,:] - L[i,j]*A[j,:] # subtract multiplier*row to eliminate

print(A)
print(L)
```

3.4. **Program: Backward Substitution.**

```
# ALGORITHM: Backward Substitution from p.94-95
import numpy as np

U = np.array([[1, 2, 1],
             [0, 2 , -2],
             [0, 0, 5]])
n = 3 # size of A is 3x3

b = np.array([2,6,-10])  # U and b from classwork
x = np.zeros(n,float)  # empty vector x to hold answer

for i in range(n-1,-1,-1): # loop backwards, starting at end
    x[i] = b[i]
    for j in range (i+1,n): # for each column greater than i
        x[i] -= U[i,j]*x[j]
    x[i] = x[i]/U[i,i]
print(x)
```

4. Chapter 6

Write your own code to do QR factorization using Gram-Schmidt. You need not write code to do QR using Householder reflectors. You need only be able to use

Householder reflectors by hand in this course. You must be able to do QR with Gram-Schmidt by hand and by writing code.

5. Chapter 7

5.1. **Program: Jacobi.**

```
import numpy as np

def Jacobi(A, b, x, num_steps):
    """My Jacobi function takes four inputs
    A, a square matrix, b, the input of Ax = b,
    x, the initial guess, and num_steps to iterate
    by Jacobi."""
    D = np.diag(np.diag(A),0)

    for k in range(num_steps):
        r = b - A@x
        x =  x + np.linalg.inv(D)@r
        print(k+1, x)
    return x

# below are the matrix and vectors to input into my Jacobi function.
A = np.array([[3,1,-1],
             [1,-4,2],
              [-2,-1,5]])
b = np.array([3,-1,2])
x = np.array([0,0,0])  # initial guess for Jacobi

Jacobi(A, b, x, 11)
```

5.2. **Program: Gauss-Seidel.**

```
import numpy as np

def Gauss_Seidel(A, b, x, num_steps):
    """Gauss_Seidel function takes four inputs
    A, a square matrix, b, the input of Ax = b,
    x, the initial guess, and num_steps to iterate
    Gauss_Seidel."""
    E = np.tril(A)

    for k in range(num_steps):
        r = b - A@x
        x =  x + np.linalg.inv(E)@r
        print(k+1, x)
    return x

# below are the matrix and vectors to input into my Gauss_Seidel function.
A = np.array([[3,1,-1],
             [1,-4,2],
              [-2,-1,5]])
b = np.array([3,-1,2])
x = np.array([0,0,0])  # initial guess for Gauss_Seidel

Gauss_Seidel(A, b, x, 11)
```

5.3. **Program: SOR.**

```
import numpy as np

def SOR(A, b, x, num_steps):
    """My SOR function takes four inputs
    A, a square matrix, b, the input of Ax = b,
    x, the initial guess, and num_steps to iterate
    by SOR."""
    E = np.tril(A)
    D = np.diag(np.diag(A),0)
    omega = 1.2

    for k in range(num_steps):
        r = b - A@x
        x =  x + omega*np.linalg.inv((1 - omega)*(D) + omega*E)@r
        print(k, x)
    return x

# below are the matrix and vectors to input into my SOR function.
A = np.array([[3,1,-1],
             [1,-4,2],
              [-2,-1,5]])
b = np.array([3,-1,2])
x = np.array([0,0,0])  # initial guess

SOR(A, b, x, 11)
```

6. Chapter 8

6.1. **Program: Power Method.**

```
#ALGORITHM: Power Method p. 222.
import numpy as np

# matrix A.  Looking for dominant eigenvector v so that Av = (lambda)v
A = np.array([[-1,2,2],
             [-1,-4,-2],
              [-3,9,7]])

v = np.array([1,0,0])  # initial guess for dominant eigenvector

for k in range(16):
    v = A@v
    v = v / np.linalg.norm(v)
    lam = np.dot(v,A@v)
    print(k+1, v, lam)
```

6.2. **Program: Inverse Power Method.**

```
# ALGORITHM: Inverse Power Iteration with alpha = 1.  This approximates
# the smallest eigenvalue.
import numpy as np
import scipy
import scipy.linalg

# matrix A.  Looking for dominant eigenvector v so that Av = (lambda)v
A = np.array([[-1,2,2],
             [-1,-4,-2],
              [-3,9,7]])

P,L,U = scipy.linalg.lu(A)  # use Python's PA = LU code

v = np.array([1,0,0])  # initial guess for smallest eigenvalue

for k in range(7):
    c = np.linalg.solve(L,P@v)  # use numpy's solve command (F.S.)
    v = np.linalg.solve(U,c)    # use numpy's solve command (B.S.)
    v = v / np.linalg.norm(v)
    lam = np.dot(v,A@v)
    print(k+1, v, lam)
```

6.3. **Program: Inverse Power Method with alpha.**

```
# ALGORITHM: Inverse Iteration p. 228
import numpy as np
import scipy
import scipy.linalg

# matrix A.  Looking for dominant eigenvector v so that Av = (lambda)v
A = np.array([[-1,2,2],
             [-1,-4,-2],
              [-3,9,7]])

alpha = -1  # approximate guess of eigenvalue
P,L,U = scipy.linalg.lu(A - alpha*np.eye(3))  # use Python's PA = LU code

v = np.array([1,0,0])  # initial guess for smallest eigenvalue

for k in range(9):
    c = np.linalg.solve(L,P@v)  # use numpy's solve command (F.S.)
    v = np.linalg.solve(U,c)    # use numpy's solve command (B.S.)
    v = v / np.linalg.norm(v)
    lam = np.dot(v,A@v)
    print(k+1, v, lam)
```

7. Chapter 9

7.1. **Newton's Method.**

```
#ALGORITHM: Newton's Method for Systems p.254
import numpy as np

def Newton_method(F, DF, x, num_steps):
    """Applies Newton's Method FPI num_steps times
    to F with initial guess x0."""
    for k in range(num_steps):
        x = x - np.linalg.solve(DF(x),F(x))  # use numpy's solve command
        print(k,x)

# Input to Newton's method:
# A function F(x); its Jacobian DF and
# an initial guess x = x0.
def F(x):
    """F is the function whose roots we will approximate."""
    return np.array([x[0]**2 - 2*x[0] - x[1] + 1, x[0]**2 + x[1]**2 - 1])

def J(x):
    """J is the Jacobian matrix of F."""
    return np.array([[2*x[0] - 2, -1],
                     [2*x[0], 2*x[1]]])

x = np.array([1,3]) # initial guess

Newton_method(F,J,x,5)
```

8. Chapter 14

8.1. **Derivation approximation by order.**

```
# Python Code for Example 14.1 on page 412
import numpy as np
import matplotlib.pyplot as plt
from math import exp

x0 = 0
f0 = exp(x0)

i = np.r_[-8:0.5:0.5]
h = 10**i
err1 = np.abs((np.exp(x0+h) - f0)/h - 1)  #order 1 error
err2 = np.abs((np.exp(x0+h) - np.exp(x0-h))/(2*h) - 1)  #order 2 error
err4 = np.abs((np.exp(x0-2*h) - 8*np.exp(x0-h) + 8*np.exp(x0+h) - np.exp(x0+2*h))/(12*h) - 1)

plt.loglog(h,err1)
plt.loglog(h,err2)
plt.loglog(h,err4)
plt.show()
```

8.2. Richardson Extrapolation.

```
from math import exp
# approximate the derivative to f at x0
def f(x):
    return exp(x)

x0 = 0
for k in range(1,6):
    h = 10**(-k)
    fPrime1 = (f(x0+h) - f(x0-h))/(2*h) # 3pt approximate derivative
    fPrime2 = (f(x0+h/2) - f(x0-h/2))/(2*(h/2)) # 2nd 3pt approximate derivative
    fPrime_better = (4*fPrime2 - fPrime1)/3 # mix of two 3pt approximates
    err = abs(fPrime_better - 1)  # absolute error
    print(k,h,err)
```

9. Chapter 15

9.1. trapezoid rule.

```
# Code from textbook "Computational Physics" by Mark Newman

def f(x):
    return x**4 - 2*x + 1

N = 10
a = 0.0
b = 2.0
h = (b-a)/N

s = 0.5*f(a) + 0.5*f(b)
for k in range(1,N):
    s += f(a+k*h)

print(h*s)
```

10. Chapter 16

10.1. Euler’s Method.

```
# Code from textbook "Computational Physics" by Mark Newman

from math import sin
from numpy import arange
from pylab import plot,xlabel,ylabel,show

def f(x,t):
    return -x**3 + sin(t)

a = 0.0           # Start of the interval
```

```
b = 10.0          # End of the interval
N = 1000          # Number of steps
h = (b-a)/N       # Size of a single step
x = 0.0           # Initial condition

tpoints = arange(a,b,h) # t-values of approx. soln x(t)
xpoints = []  # x-values of approx. soln x(t)
for t in tpoints:
    xpoints.append(x)
    x += h*f(x,t)

plot(tpoints,xpoints)
xlabel("t")
ylabel("x(t)")
show()
```

10.2. **Midpoint.**

```
# Code from textbook "Computational Physics" by Mark Newman

from math import sin
from numpy import arange
from pylab import plot,xlabel,ylabel,show

def f(x,t):
    return -x**3 + sin(t)

a = 0.0
b = 10.0
N = 10
h = (b-a)/N

tpoints = arange(a,b,h)
xpoints = []

x = 0.0
for t in tpoints:
    xpoints.append(x)
    k1 = h*f(x,t)
    k2 = h*f(x+0.5*k1,t+0.5*h)
    x += k2

plot(tpoints,xpoints)
xlabel("t")
ylabel("x(t)")
show()
```

10.3. **RK4.**

```
# Code from textbook "Computational Physics" by Mark Newman
from math import sin
from numpy import arange
from pylab import plot,xlabel,ylabel,show

def f(x,t):
    return -x**3 + sin(t)

a = 0.0
b = 10.0
N = 10
h = (b-a)/N

tpoints = arange(a,b,h)
xpoints = []
x = 0.0

for t in tpoints:
    xpoints.append(x)
    k1 = h*f(x,t)
    k2 = h*f(x+0.5*k1,t+0.5*h)
    k3 = h*f(x+0.5*k2,t+0.5*h)
    k4 = h*f(x+k3,t+h)
    x += (k1+2*k2+2*k3+k4)/6

plot(tpoints,xpoints)
xlabel("t")
ylabel("x(t)")
show()
```

References