Task 1

I have created the following code that implement Gaussian elimination and backsubsitution for square linear systems of equations. The code is documented here in the Software Manual.

```
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 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]))
```

with the following output:

```
xianggao@Xiangs-MacBook-Pro Tasksheet_08 % python Task_1.py
Enter number of unknowns: 3
Enter Augmented Matrix Coefficients:
a[0][0]=1
a[0][1]=1
a[0][2]=1
a[0][3]=9
a[1][0]=2
a[1][1]=-3
a[1][2]=4
a[1][3]=13
a[2][0]=3
a[2][1]=4
a[2][2]=5
a[2][3]=40

Required solution is:
X0 = 1.00
X1 = 3.00
X2 = 5.00
```

Figure 1. Running the Task 1.py From the Terminal.

Task 2

I have created the following code that implement forward substitution. The code is documented here in the Software Manual.

```python
import numpy as np

def forward_substitution(A, b):
    # Get number of rows
    n = A.shape[0]

    # Allocating space for the solution vector
    y = np.zeros_like(b, dtype=np.double);

    # Here we perform the forward-substitution.
    # Initializing  with the first row.
    y[0] = b[0] / A[0, 0]

    # Looping over rows in reverse (from the bottom  up), starting with the second to last row, because the
    # last row solve was completed in the last step.
    for i in range(1, n):
        y[i] = (b[i] - np.dot(A[i, :i], y[:i])) / A[i, i]
    return y
```

I have also created the following code that implement LU-factorization. The code is documented here in the Software Manual.

And the testing matrix A is

$$A = \begin{bmatrix} 7 & 3 & -1 & 2 \\ 3 & 8 & 1 & -4 \\ -1 & 1 & 4 & -1 \\ 2 & -4 & -1 & 6 \end{bmatrix}$$

```python
import numpy as np
from matrixP import pivot_matrix

A = [[7, 3, -1, 2], [3, 8, 1, -4], [-1, 1, 4, -1], [2, -4, -1, 6]]

n = len(A)

# Create zero matrices for L and U
L = np.zeros([n, n])
U = np.zeros([n, n])

# Perform the LU Decomposition

for j in range(n):
    # All diagonal entries of L are set to unity
    L[j][j] = 1.0

    for i in range(j + 1):
        s1 = sum(U[k][j] * L[i][k] for k in range(i))
        U[i][j] = np.matmul(pivot_matrix(A), A)[i][j] - s1

    for i in range(j, n):
        s2 = sum(U[k][j] * L[i][k] for k in range(i))
        L[i][j] = (np.matmul(pivot_matrix(A), A)[i][j] - s2) / U[j][j]

print("The lower Triangular Matrix is: ")
print(L)
print("The upper Triangular Matrix is: ")
print(U)
```

```
The lower Triangular Matrix:
[[ 1.          0.          0.          0.        ]
 [ 0.42857143  1.          0.          0.        ]
 [-0.14285714  0.21276596  1.          0.        ]
 [ 0.28571429 -0.72340426  0.08982036  1.        ]]
The upper Triangular Matrix:
[[ 7.          3.         -1.          2.        ]
 [ 0.          6.71428571  1.42857143 -4.85714286]
 [ 0.          0.          3.55319149  0.31914894]
 [ 0.          0.          0.          1.88622754]]
```

Figure 2. Result of Running the Task 2.py.

Task 3

I have created the following code that uses LU-factorization on Hilbert matrices of size $n = 4, 5, \ldots, 10$. The code is documented here in the Software Manual.

```python
import numpy as np
from matrixP import pivot_matrix

np.set_printoptions(precision=2)

def hilmat(n):
    return [[1 / (i + j + 1) for j in range(n)] for i in range(n)]

def inverse(A):
    n = len(A)
    # Create zero matrices for L and U
    L = np.zeros([n, n])
    U = np.zeros([n, n])

    for j in range(n):
        # All diagonal entries of L are set to unity
        L[j][j] = 1.0

        for i in range(j + 1):
            s1 = sum(U[k][j] * L[i][k] for k in range(i))
            U[i][j] = np.matmul(pivot_matrix(A), A)[i][j] - s1

        for i in range(j, n):
            s2 = sum(U[k][j] * L[i][k] for k in range(i))
            L[i][j] = (np.matmul(pivot_matrix(A), A)[i][j] - s2) / U[j][j]

        inverseMat = np.matmul(np.linalg.inv(U), np.linalg.inv(L))
        return inverseMat

for i in range(4, 11):
    x = np.ones(i)
    sol = np.matmul(x, np.linalg.inv(hilmat(i)))
    print('The solution with n =', i)
    print(sol)
```

with the following output:

```
The solution with n = 4
[  -4.   60. -180.  140.]
The solution with n = 5
[    5.  -120.   630. -1120.   630.]
The solution with n = 6
[-6.00e+00  2.10e+02 -1.68e+03  5.04e+03 -6.30e+03  2.77e+03]
The solution with n = 7
[ 7.00e+00 -3.36e+02  3.78e+03 -1.68e+04  3.47e+04 -3.33e+04  1.20e+04]
The solution with n = 8
[-8.00e+00  5.04e+02 -7.56e+03  4.62e+04 -1.39e+05  2.16e+05 -1.68e+05
  5.15e+04]
The solution with n = 9
[ 9.00e+00 -7.20e+02  1.39e+04 -1.11e+05  4.50e+05 -1.01e+06  1.26e+06
 -8.24e+05  2.19e+05]
The solution with n = 10
[-1.00e+01  9.90e+02 -2.38e+04  2.40e+05 -1.26e+06  3.78e+06 -6.73e+06
  7.00e+06 -3.94e+06  9.24e+05]
```

Figure 3. Result of Running the Task 3.py.

Task 4

I have created the following code that implements scaled partial pivoting for the Gaussian elimination method for solving linear systems of equations. The code is documented here in the Software Manual.
And the testing matrix A is

$$A = \begin{bmatrix} 7 & 3 & -1 & 2 \\ 3 & 8 & 1 & -4 \\ -1 & 1 & 4 & -1 \\ 2 & -4 & -1 & 6 \end{bmatrix}$$

where the vector b is

$$b = \begin{bmatrix} 1, 1, 1, 1 \end{bmatrix}$$

```python
import numpy as np

A = [[7, 3, -1, 2], [3, 8, 1, -4], [-1, 1, 4, -1], [2, -4, -1, 6]]

b = np.ones(4)

def GussPivot(A, b):
    n = len(A)
    M = A
    i = 0
    for x in M:
        x.append(b[i])
        i += 1
    for k in range(n):
        for i in range(k,n):
            if abs(M[i][k]) > abs(M[k][k]):
                M[k], M[i] = M[i],M[k]
            else:
                pass
        for j in range(k + 1, n):
            q = float(M[j][k]) / M[k][k]
            for m in range(k, n+1):
                M[j][m] -= q * M[k][m]
    x = [0 for i in range(n)]
    x[n-1] = float(M[n-1][n]) / M[n-1][n-1]
    for i in range(n-1, -1, -1):
        z = 0
        for j in range(i+1, n):
            z = z + float(M[i][j])*x[j]
        x[i] = float(M[i][n] - z)/M[i][i]
    return x

print(GussPivot(A, b))
```

with the following output:

```
xianggao@Xiangs-MacBook-Pro Tasksheet_08 % python Task_4.py
[-0.16507936507936494, 0.43174603174603154, 0.23809523809523805, 0.549206349206349]
```

Figure 4. Result of Running the Task 4.py.

Task 5

I have created the following code that uses LU-factorization on Hilbert matrices of size $n = 4, 5, \ldots, 8$. The code is documented here in the Software Manual.

```python
import numpy as np
from Task_4 import GussPivot

np.set_printoptions(precision=2)
```

```
5
6  def hilmat(n):
7      return [[1 / (i + j + 1) for j in range(n)] for i in range(n)]
8
9  for i in range(4, 8):
10     x = np.ones(i)
11     sol = GussPivot(hilmat(i), x)
12     print('The solution with n =', i)
13     print(sol)
```

with the following output:

```
The solution with n = 4
[-3.9999999999993214, 59.99999999999225, -179.99999999998124, 139.99999999998778]
The solution with n = 5
[5.000000000000284, -120.00000000003425, 630.000000000222, -1120.0000000004109, 630.0000000002276]
The solution with n = 6
[-6.00000000112027, 210.0000000329486, -1680.0000002266825, 5040.00000059605, -6300.000000662943, 2772.0000002626984]
The solution with n = 7
[7.000000043019099, -336.0000017111369, 3780.0000164323465, -16800.000063703596, 34650.00011650232, -33264.00010045749, 12012.000032924145]
```

Figure 5. Result of Running the Task 5.py.

Task 6

The Hilbert Matrix is a popular Matrix throughout Mathematics. In this article[1] I found, specifically, its use in studying digital computing during its early days. The Hilbert Matrix has a condition number that grows rapidly as the square matrix grows in number of columns/rows. Below is the growth rate for the asymptotic growth rate for the 2-norm:

$$\kappa_2 (H_n) \sim e^{3.5n}$$

And in this article[2] I found, there exists multiple ways of generating the Hilbert Matrix that's both beneficial to Computer Scientists and Mathematicians.

[1] https://www.tandfonline.com/doi/abs/10.1080/00029890.1983.11971218
[2] https://nhigham.com/2020/06/30/what-is-the-hilbert-matrix/