SỬ DỤNG PYTHON XÂY DỰNG MODUL THỰC HIỆN MỘT SỐ TÍNH TOÁN MA TRẬN TRONG ĐẠI SỐ TUYẾN TÍNH

USING PYTHON TO PROGRAM A MODULE TO PERFORM MATRIX CALCULATIONS IN LINEAR ALGEBRA

Chu Bình Minh[1], Trần Bảo Trung[2], Nguyễn Mai Quyên[3]

[1]*Khoa Khoa học ứng dụng, Trường Đại học Kinh tế - Kỹ thuật Công nghiệp*

[2]*Khoa Công nghệ thông tin, Trường Đại học Kinh tế - Kỹ thuật Công nghiệp*

[3]*Khoa Toán kinh tế, Trường Đại học Kinh tế quốc dân*

Đến Tòa soạn ngày 28/08/2022, chấp nhận đăng ngày 12/09/2022

Tóm tắt: Bài báo trình bày cách sử dụng Python lõi xây dựng một modul thực hiện tính toán một số phép tính toán ma trận. Modul được xây dựng một cách trực quan, rõ ràng bởi tập các hàm được lập trình dựa trên các thuật toán về các phép toán về ma trận trong đại số tuyến tính.

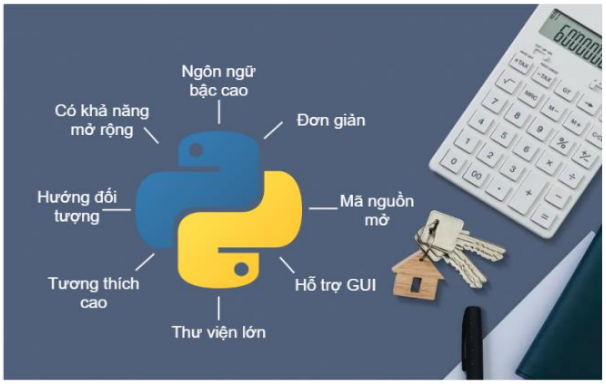
Từ khóa: Python, ma trận, đại số tuyến tính.

Abstract: The paper presents a way to use the Python-core to program a module that performs some matrix calculations. The module is built intuitively and clearly by a set of functions programmed based on algorithms for matrix operations in linear algebra.

Keywords: Python, matrix, linear algebra.

1. GIỚI THIỆU

Ma trận không những là một trong những khái niệm trung tâm của toán học mà còn là một khái niệm được ứng dụng vào rất nhiều lĩnh vực. Do vậy, tính toán ma trận là một trong những kỹ năng cần thiết và quan trọng đối với mỗi kỹ sư và mỗi nhà khoa học. Các phép toán về ma trận, vectơ được đưa vào giảng dạy cho sinh viên Trường Đại học Kinh tế - Kỹ thuật Công nghiệp trong học phần Đại số tuyến tính (ĐSTT) nhằm trang bị cho sinh viên những kỹ năng tính toán ma trận để áp dụng vào các lĩnh vực chuyên ngành [1]. Thời lượng học của học phần ĐSTT là 02 tín chỉ nên thông qua học phần này, sinh viên chỉ có thể nắm được một số khái niệm cơ bản về tính toán ma trận và thực hiện các phép tính toán ma trận một cách thủ công với những bài toán có kích thước nhỏ. Việc hướng dẫn sinh viên sử dụng các phần mềm chuyên dụng để tính toán hoặc sử dụng các ngôn ngữ lập trình để cài đặt các phép tính toán ma trận chưa được đề cập trong quá trình giảng - dạy trong Trường Đại học Kinh tế - Kỹ thuật Công nghiệp.



Hình Error! No text of specified style in document.**. Ưu điểm của ngôn ngữ lập trình Python**

Hiện nay, sự phát triển nhanh chóng của các

nền tảng công nghệ thông tin, kéo theo sự ra đời của nhiều ngôn ngữ lập trình. Mỗi ngôn ngữ đều có ưu điểm, nhược điểm khác nhau tùy thuộc vào mục đích sử dụng hay nhu cầu của người dùng. Python là một ngôn ngữ lập trình bậc cao được Guido van Rossum tạo ra vào năm 1991 và tiếp tục được phát triển cho đến ngày nay [3]. Python là một ngôn ngữ thông dịch với các ưu điểm như đơn giản, dễ sử dụng, linh hoạt... (hình 1) nên Python được sử dụng rộng rãi trong rất nhiều lĩnh vực và ngày càng trở nên phổ biến. Do Python có cấu trúc cao cấp, mạnh mẽ nhưng lại có tiếp cận hết sức đơn giản với lập trình hướng đối tượng nên Python rất phù hợp cho những người lần đầu tiếp xúc với ngôn ngữ lập trình.

Python cho phép người dùng thực hiện các phép tính toán ma trận bằng cách gọi thư viện Numpy. Numpy là thư viện nổi tiếng của Python hỗ trợ các tính toán ma trận rất hiệu quả và được sử dụng phổ biến trong cộng đồng lập trình Python. Các phép tính toán về ma trận trong thư viện Numpy sẽ chỉ cho người dùng kết quả cuối cùng mà không cho biết quá trình tính toán. Do vậy, nếu sử dụng thư viện Numpy để minh hoạ các tính toán ma trận cho sinh viên sẽ dẫn đến hạn chế là sinh viên sẽ không hiểu được các thuật toán trong tính toán ma trận cũng như không rèn luyện được tư duy lập trình cho sinh viên. Hơn nữa, việc áp dụng máy móc các phép toán trong Numpy cho một số bài toán đặc thù như ma trận thưa, ma trận dải sẽ làm quá trình tính toán không hiệu quả. Do đó, chúng tôi sử dụng Python lõi và các thuật toán được trình bài trong học phần ĐSTT, xây dựng modul Uneti_Matrix chứa các hàm tính toán ma trận nhằm giúp cho sinh viên vừa có thể nắm vững các thuật toán trong tính toán ma trận để hiểu sâu các kiến thức trong học phần ĐSTT, vừa rèn luyện tư duy lập trình và kỹ năng viết code cho sinh viên. Cấu trúc của bài báo này được trình bày như sau: Phần 2 giới thiệu một số dạng ma trận cơ bản và một số phép tính toán ma trận được sử dụng phổ biến; Phần 3 là nội dung chính của bài báo, phần này sẽ trình bài chi tiết 26 hàm tính toán ma trận bằng Python lõi minh họa cho các tính toán ma trận. Các vấn đề liên quan và kết luận sẽ được chúng tôi trình bày trong Phần 4.

2. MỘT SỐ PHÉP TÍNH TOÁN MA TRẬN

Mục này sẽ tóm tắt một số khái niệm, tính toán liên quan đến ma trận đã được trình bày trong [1,2] và giới thiệu các hàm thực hiện các tính toán này. Chi tiết của các hàm sẽ được trình bày trong Mục 3. Trước hết, chúng tôi sẽ giới thiệu lại một số dạng ma trận được sử dụng phổ biến và một số cách tạo ma trận.

2.1. Tạo và biểu diễn một số ma trận cơ bản

Ma trận cỡ n×m là một mảng các giá trị hình chữ nhật gồm n hàng, m cột. Các giá trị trong ma trận được gọi là các phần tử của ma trận. Để lưu trữ ma trận trong Python, người ta sử dụng kiểu dữ liệu List of List. Chẳng hạn, ma trận cỡ 2×3 sau:

$$A = \begin{bmatrix} 2 & 2 & 3 \\ 3 & 0 & 1 \end{bmatrix}$$

sẽ được lưu trữ dưới dạng list

$$A = \big[[2\ \ 2\ \ 3], [3\ \ 0\ \ 1]\big]$$

Để tạo được một ma trận, ta có thể nhập lần lượt từng phần tử từ bàn phím hoặc các phần tử có thể được tạo một cách ngẫu nhiên. Việc nhập ma trận từ bàn phím được thực hiện bởi hàm `input_matrix(row,col)` và nhập ma trận với các phần tử ngẫu nhiên được thực biện bằng hàm

`creat_matrix(row,col,max=100)`.

Ma trận không là trường hợp đặc biệt khi tất cả các phần tử của ma trận đều bằng 0. Để tạo ma trận không cỡ row×col, ta sử dụng hàm `zero_matrix(row,col)`.

Ngoài việc tạo ma trận bằng hai cách trên, ta có thể tạo ma trận con của A bằng cách lấy các phần tử từ dòng row1 đến dòng row2 và các phần tử từ cột col1 đến cột col2 của ma trận cho trước. Phép toán này được thực hiện bởi hàm `submatrix(A,row1,row2,col1,col2)`.

Khi có một ma trận, ta có thể thực hiện đổi vị trí của 2 hàng (hoặc 2 cột tuỳ ý). Hơn nữa ta có thể đổi các hàng của một ma trận thành các cột. Phép đổi chỗ hai hàng/cột được thực hiện bởi các hàm `change_row_matrix(A,row1,row2)` và `change_row_matrix(A,row1,row2)`. Phép đổi hàng thành cột của ma trận được gọi là phép chuyển vị và được tính toán trong hàm `transpose_matrix(A)`.

Một trường hợp phổ biến trong tính toán ma trận là khi ma trận có số hàng và số cột bằng nhau, ta gọi là ma trận vuông. Khi đó, các phần từ nằm chéo từ góc trên bên trái xuống góc dưới bên phải được gọi là đường chéo chính của ma trận. Một ma trận vuông có các phần tử nằm dưới đường chéo chính bằng 0 được gọi là ma trận tam giác trên, ngược lại một ma trận vuông có các phần tử nằm phía trên đường chéo chính bằng 0 ta gọi là ma trận tam giác dưới. Nếu một ma trận vuông có các phần tử nằm trên đường chéo chính bằng 1 và các phần tử nằm ngoài đường chéo chính bằng 0 ta gọi là ma trận đơn vị. Ma trận đơn vị được tính toán bởi hàm `identify_matrix(row)`.

Trong quá trình tính toán ta cần hiển thị ma trận ra màn hình một cách trực quan để theo dõi kết quả, khi đó ta sử dụng hàm `print_matrix(A)`.

2.2. Một số phép tính toán ma trận cơ bản

Với các ma trận thoả mãn một số điều kiện nhất định, ta có thể thực hiện các phép toán ký hiệu hình thức tương tự như các phép toán trong số thực. Đơn giản nhất là phép cộng/trừ hai ma trận. Để thực hiện cộng/trừ hai ma trận cùng cỡ, ta chỉ việc lấy các phần từ cùng vị trí của hai ma trận cộng/trừ cho nhau. Phép toán cộng/trừ hai ma trận được thực hiện bởi các hàm `add_matrix(A,B)`, `sub_matrix(A,B)`.

Tương tự như cộng/trừ ma trận, với hai ma trận cùng cỡ, ta các thể lấy các phần tử cùng vị trị nhân với nhau để được một ma trận mới, phép toán này được thực hiện bởi hàm `dot_matrix(A,B)`.

Nếu ma trận **A** có số cột bằng số hàng của ma trận **B** thì ta có thể tính tích **A** với **B**, phép toán này thực hiện bởi hàm `mult_matrix(A,B)`. Nếu ta muốn nhân một số với một ma trận, ta sẽ lấy số đó nhân với tất cả các phần tử của ma trận. Phép toán này được tính toán bởi `mult_real_matrix(A,k)`.

Với ma trận vuông A, hàm `mult_matrix(A,A)` sẽ được một ma trận mới ký hiệu là A^2, gọi là luỹ thừa bậc 2 của **A**. Ta có thể thực hiện luỹ thừa bậc k của ma trận **A** bởi hàm `power_matrix(A,n)`.

Ngoài các phép toán cơ bản như trên, ma trận còn có các phép toán phức tạp như: tính định thức của ma trận, tính hạng của ma trận, phân rã ma trận,... Mục sau đây sẽ giới thiệu các phép tính toán này.

2.3. Một số phép tính toán ma trận nâng cao

Tìm nghiệm hệ phương trình ĐSTT là một trong những tính toán quan trong của ĐSTT và được áp dụng trong nhiều lĩnh vực. Có một

số phương pháp giải hệ phương trình ĐSTT như phương pháp Cramer, phương pháp ma trận nghịch đảo, phương pháp Gauss. Tuy nhiên khi số lượng biến của hệ phương trình lớn thì chỉ có phương pháp Gauss mới tính toán hiệu quả. Ý tưởng của phương pháp Gauss là biến đổi phương trình về dạng tam giác trên (hoặc ma trận tam giác dưới), sau đó sử dụng các phương pháp thế ngược (hoặc thế thuận) để tìm nghiệm. Để việc tính toán không gặp phải gặp lỗi chia cho 0, phương pháp Gauss được kết hợp với thuật toán tìm phần tử trội. Quá trình tính toán của phương pháp Gauss để tìm nghiệm hệ phương trình ĐSTT **Ax=b** được minh hoạ trong hàm `sol_gauss(A,b)`.

Việc áp dụng phương pháp Gauss khi cần giải hệ phương trình **Ax** = **b** mà hệ số tự do b thay đổi sẽ không hiệu quả vì ta phải lặp lại quá trình biến đổi ma trận **A**. Do vậy, để tính toán hiệu quả trước tiên ta phân tích ma trận **A=L.U**, với **L** là ma trận tam giác dưới và **U** là ma trận tam giác trên. Sau đó thực hiện giải hai hệ tam giác là **Ly=b** và **Ux=y**. Quá trình phân tích **A=L.U** được tính toán bởi hàm `LUdecomp(A)` và quá trình thực hiện giải hệ **Ax**=**b** phương trình được thực hiện bởi `sol_LUequs(A,b)`.

Ngoài việc sử dụng phân tích ma trận để giải hệ phương trình ĐSTT thì ta còn có thể sử dụng phân tích này để tính định thức và tìm ma trận nghịch đảo. Dựa vào tính chất của phương pháp Gauss thì định thức của ma trận **A** chính là tích các phần tử nằm trên đường chéo của ma trận sau khi đã chéo hoá. Do vậy, định thức của ma trận **A** được tính bởi `determinant(A)`.

Do $\mathbf{A.A^{-1}=I}$, với $\mathbf{A^{-1}}$ là ma trận nghịch đảo của **A, I** là ma trận đơn vị, nên để tìm ma trận nghịch đảo $\mathbf{A^{-1}}$, ta thực hiện giải phương trình ma trận **Ax=I**. Việc này tương đương với ta phải giải n phương trình $\mathbf{Ax_i=I_i}$. Do vậy, ta có thể sử dụng phương pháp phân rã **LU** để tìm $\mathbf{x_i}$. Dựa vào thuật toán trên, ma trận nghịch đảo được tính toán bởi hàm `invMatrix(A)`.

Hạng của một ma trận là số chiều của không gian vectơ sinh bởi các cột (hoặc các hàng) của nó. Để tính hạng của ma trận, ta có thể sử dụng phép biến đổi sơ cấp, đưa ma trận về dạng hình thang. Khi đó, hạng của ma trận chính là số hàng khác 0 của ma trận. Hạng của ma trận được tính toán bởi hàm `rank_matrix(A)`.

Có hai loại chuẩn ma trận được sử dụng phổ biến là max và chuẩn Frobenius. Chuẩn max của ma trận **A** được tính bằng giá trị lớn nhất của tổng trị tuyệt đối của mỗi hàng, chuẩn Frobenius tính bằng căn bậc hai của tổng bình phương các phần tử của một ma trận. Hàm `norm_Matrix(A,mode=1)` sẽ tính chuẩn của ma trận **A** với mặc định là chuẩn max. Nếu chọn tham biến mode=2 thì sẽ cho ta chuẩn Frobenius.

3. XÂY DỰNG MODUL UNETI_MATRIX THỰC HIỆN TÍNH TOÁN MA TRẬN

3.1. Modul trong Python

Python được xây dựng với lõi nhỏ gọn với lý do cho phép người dùng có thể bổ sung thêm các thư viện hàm số hoặc tạo các thư viện cho riêng mình để tiện cho quá trình sử dụng. Các thư viện hàm số này được gọi là modul. Để thiết lập một modul, ta thực hiện một file dạng tên_modul.py mà có chứa các hàm cần sử dụng. Sau đó, để sử dụng các modul đã có sẵn ta thực hiện lệnh

`import` <tên_modul>

và sử dụng các hàm đã có trong modul dưới dạng câu lệnh: tên_modul.tên_hàm

Để đỡ mất thời gian viết lại tên_modul, ta có thể thực hiện lệnh

`from` <tên_modul> `import *`

và thực hiện hàm trong modul bằng cách gọi: `tên_hàm`

3.2. Xây dựng modul Uneti_Matrix

Sử dụng các thuật toán được trình bày trong Mục 2 và các câu lệnh cơ bản trong Python lõi, chúng tôi xây dựng modul Uneti_Matrix để tính toán ma trận như sau:

Bước 1. Tạo modul Uneti_Matrix

Lưu nội dung sau vào file: Uneti_Matrix.py

```
from random import random
def input_matrix(row,col):
    A=[]
    for i in range(row):
        s=input("Nhập hàng {} của ma trận ({} số
cách nhau dấu cách): ".format(i+1,col))
        r=[float(x) for x in s.split()]
        A.append(r)
    return A
def creat_matrix(row,col,max=100):
    A=[]
    for i in range(row):
        r=[]
        for j in range(col):
            r.append(max*random())
        A.append(r)
    return A
def zero_matrix(row,col):
    A=[]
    for i in range(row):
        A.append([float(0)]*col)
    return A
def identify_matrix(row):
    A=[]
    for i in range(row):
      temp=[float(0)]*i+[float(1)]+[float(0)]
*(row-1)
        A.append(temp)
    return A
def print_matrix(A):
    for i in range(len(A)):
        for j in range(len(A[0])):
            print("{:#8.3}".format(A[i][j]),
end="")
        print()
def add_matrix(A,B):
            if     (len(A)!=len(B))     or
(len(A[0])!=len(B[0])):
     return print("Hai ma trận không cùng cỡ!")
    else:
        T=[]
        for i in range(len(A)):
            S=[]
            for j in range(len(A[0])):
                S.append(A[i][j]+B[i][j])
            T.append(S)
        return T
def sub_matrix(A,B):
          if       (len(A)!=len(B))       or
(len(A[0])!=len(B[0])):
     return print("Hai ma trận không cùng cỡ!")
    else:
        T=[]
        for i in range(len(A)):
            S=[]
            for j in range(len(A[0])):
                S.append(A[i][j]-B[i][j])
            T.append(S)
        return T
def mult_matrix(A,B):
   if len(A[0])!=len(B):
        return print("Hai ma trận không nhân
được!")
    else:
        AB=[]
        for i in range(len(A)):
            m=[]
            for j in range(len(B[0])):
                S=0
                for k in range(len(A[0])):
                    S=S+A[i][k]*B[k][j]
                m.append(S)
            AB.append(m)
        return AB
def mult_real_matrix(A,k):
    Ak=[]
    for i in range(len(A)):
        m=[]
```

```
        for j in range(len(A[0])):
            m.append(A[i][j]*k)
        Ak.append(m)
    return Ak
def transpose_matrix(A):
    T=[]
    for j in range(len(A[0])):
        m=[]
        for i in range(len(A)):
            m.append(A[i][j])
        T.append(m)
    return T
def change_list(lst,k,p):
    l=lst
    temp=l[k]
    l[k]=l[p]
    l[p]=temp
    return l
def change_row_matrix(A,row1,row2):
    M=A.copy()
    temp=M[row1].copy()
    M[row1]=M[row2].copy()
    M[row2]=temp
    return M
def change_col_matrix(A,col1,col2):
    M=A.copy()
    for i in range(len(A)):
        M[i][col1-1],M[i][col2-1]=M[i][col2-1
],M[i][col1-1]
    return M
def submatrix(A,row1,row2,col1,col2):
    M=[]
    for row in A[row1-1:row2]:
        M.append(row[col1-1:col2])
    return M
def determinant(A):
   if len(A) !=len(A[0]):
    return print("Không phải là ma trận vuông!")
    else:
        M=A.copy()
        n = len(M)
        s=[] # Hệ số tỉ lệ
        for row in M:
            abs_row=[abs(i) for i in row]
            s.append(max(abs_row))
        for k in range(0,n-1):
            p=k
            mmax=abs(M[k][k]/s[k])
            for i in range(k+1,len(M)): # Tìm
phần tử chội
                if mmax<abs(M[i][k]/s[i]):
                    mmax=abs(M[i][k]/s[i])
                    p=i
            M=change_row_matrix(M,k,p)
            s=change_list(s,k,p)
            for i in range(k+1,n):
                lam = M[i][k]/M[k][k]
                for j in range(0,n):
                   M[i][j]=M[i][j]-lam*M[k][j]
        d=1
        for k in range(n):
            d=d*M[k][k]
        return d
def power_matrix(A,n):
    if len(A) !=len(A[0]):
         return print("Không phải là ma trận
vuông!")
    else:
        P=A.copy()
        n=int(n)
        for i in range(2,n):
            M=P.copy()
            P=mult_matrix(M,A)
        return P
def dot_matrix(A,B):
              if      (len(A)!=len(B))      or
(len(A[0])!=len(B[0])):
      return print("Hai ma trận không cùng cỡ!")
    else:
        Ad=[]
        for i in range(len(A)):
            row=[]
            for j in range(len(A[0])):
                row.append(A[i][j]*B[i][j])
            Ad.append(row)
        return Ad
def rank_matrix(A):
    M=A.copy()
    if len(M)>len(M[0]):
        M=transpose_matrix(M)
    s=[] # Hệ số tỉ lệ
    for row in M:
        abs_row=[abs(i) for i in row]
        s.append(max(abs_row))
```

```
    for k in range(0,len(M)-1):
        p=k
        mmax=abs(M[k][k]/s[k])
        for i in range(k+1,len(M)): # Tìm phần
tử chội
            if mmax<abs(M[i][k]/s[i]):
                mmax=abs(M[i][k]/s[i])
                p=i
        M=change_row_matrix(M,k,p)
        s=change_list(s,k,p)
        for i in range(k+1,len(M)):
            lam = M[i][k]/M[k][k]
            for j in range(0,len(M[0])):
                M[i][j]=M[i][j]-lam*M[k][j]
    rank=len(M)
    while rank>0:
        if M[rank-1][len(M[0])-1]==0:
            rank=rank-1
        else:
            break
    return rank,M
def norm_Matrix(A,mode=1):
   if mode==1:
        l=[]
        for row in A:
            r=[abs(i) for i in row]
            l.append(sum(r))
        return max(l)
    if mode==2:
        l=[]
        for row in A:
            r=[i**2 for i in row]
            l.append(sum(r))
        return sum(l)
def norm_Vector(v,mode=2):
   if mode==1:
        l=[abs(i) for i in v]
        return sum(l)
    if mode>1:
        l=[i**mode for i in v]
        return (sum(l))**(1/mode)
def sol_upperTri(A,b):
    flag=0
    if (len(A)!=len(A[0])) or (len(A)!=len(b))
or (len(A[0])!=len(b)):
        flag=1
        return print("Các hệ số không cùng cỡ")
    else:
        M=A.copy()
        n=len(M)
        x=[0]*n
        if M[n-1][n-1]==0:
            flag=1
            return print("Phần tử {} của đường
chéo bằng không".format(n-1))
        else:
            x[n-1]=b[n-1]/M[n-1][n-1]
            for i in range(n-2,-1,-1):
                s=b[i]
                for j in range(i+1,n):
                    s=s-M[i][j]*x[j]
                if M[i][i]==0:
                    flag=1
                    return print("Phần tử {} của
đường chéo bằng không".format(i))
                else:
                    x[i]=s/M[i][i]
            return x
def sol_lowerTri(A,b):
    flag=0
    if (len(A)!=len(A[0])) or (len(A)!=len(b))
or (len(A[0])!=len(b)):
        flag=1
        return print("Các hệ số không cùng cỡ")
    else:
        M=A.copy()
        n=len(M)
        x=[0]*n
        if M[0][0]==0:
            flag=1
            return print("Phần tử {} của đường
chéo bằng không".format(0))
        else:
            x[0]=b[0]/M[0][0]
            for i in range(1,n):
                s=b[i]
                for j in range(0,i):
                    s=s-M[i][j]*x[j]
                if M[i][i]==0:
                    flag=1
                    return print("Phần tử {} của
đường chéo bằng không".format(i))
                else:
                    x[i]=s/M[i][i]
```

```
            return x
def sol_gauss(A,b):
    flag=0
    if (len(A)!=len(A[0])) or (len(A)!=len(b))
or (len(A[0])!=len(b)):
        flag=1
        return print("Các hệ số không cùng cỡ")
    else:
        M=A.copy()
        n=len(M)
        s=[] # Hệ số tỉ lệ
        for row in M:
            abs_row=[abs(i) for i in row]
            s.append(max(abs_row))
        for k in range(0,n-1):
            p=k
            mmax=abs(M[k][k]/s[k])
            for i in range(k+1,n): # Tìm phần tử
chội
                if mmax<abs(M[i][k]/s[i]):
                    mmax=abs(M[i][k]/s[i])
                    p=i
            M=change_row_matrix(M,k,p)
            s=change_list(s,k,p)
            b=change_list(b,k,p)
            for i in range(k+1,n):
                lam = M[i][k]/M[k][k]
                for j in range(0,n):
                   M[i][j]=M[i][j]-lam*M[k][j]
                b[i]=b[i]-lam*b[k]
        x=sol_upperTri(M,b)
        return x
def LUdecomp(A):
   if len(A) !=len(A[0]):
         return print("Không phải là ma trận
vuông!")
    else:
        M=A.copy()
        n = len(M)
        s=[] # Hệ số tỉ lệ
        for row in M:
            abs_row=[abs(i) for i in row]
            s.append(max(abs_row))
        pern=[i for i in range(n)]
        for k in range(0,n-1):
            p=k
            mmax=abs(M[k][k]/s[k])
            for i in range(k+1,n): # Tìm phần tử
chội
                if mmax<abs(M[i][k]/s[i]):
                    mmax=abs(M[i][k]/s[i])
                    p=i
            M=change_row_matrix(M,k,p)
            s=change_list(s,k,p)
            pern=change_list(pern,k,p)
            for i in range(k+1,n):
                lam = M[i][k]/M[k][k]
                for j in range(k+1,n):
                   M[i][j]=M[i][j]-lam*M[k][j]
                   M[i][k]=lam
     U=[]
     L=[]
     for i in range(len(A)):
       U.append([0.0]*i+M[i][i:])
       L.append(M[i][:i]+[1.0]+[0.0]*(n-i-1))
        return U,L,pern
def sol_LUequs(A,b):
    flag=0
    if (len(A)!=len(A[0])) or (len(A)!=len(b))
or (len(A[0])!=len(b)):
        flag=1
        return print("Các hệ số không cùng cỡ")
    else:
        M=A.copy()
        n=len(M)
        U,L,pern=LUdecomp(M)
        b=[b[i] for i in pern]
        y=sol_lowerTri(L,b)
        x=sol_upperTri(U,y)
        return x
def invMatrix(A):
    flag=0
    if (len(A)!=len(A[0])):
        flag=1
        return print("Đây không phải là ma trận
vuông!")
    else:
        M=A.copy()
        n=len(M)
        U,L,pern=LUdecomp(M)
        eye=identify_matrix(n)
        invA=[]
        for b in eye:
            b=[b[i] for i in pern]
```

```
            y=sol_lowerTri(L,b)
            x=sol_upperTri(U,y)
            invA.append(x)
        return transpose_matrix(invA)
```

Lưu ý: Ta cần khai báo modul random vì để tạo một ma trận gồm các phần tử chọn ngẫu nhiên bởi hàm `creat_matrix(row,col,max=100)` ta cần sử dụng hàm random trong modul này để tạo số ngẫu nhiên.

Bước 2. Gọi modul Uneti_Matrix

```
from Uneti_Matrix import *
```

Bước 3. Sử dụng modul Uneti_Matrix tính toán

Sau khi gọi modul Uneti_Matrix ở Bước 2, ta có thể sử dụng các hàm trong modul để thực hiện tính toán ma trận. Ví dụ sau đây thực hiện việc tạo một ma trận ngẫu nhiên **A** có cỡ 3x3, sau đó thực hiện tính ma trận **invA** là nghịch đảo của **A** và thực hiện tìm nghiệm **Sol** của hệ phương trình **Ax**=**b**, với **b** được nhập từ bàn phím.

```
>>> from Uneti_Matrix import *
>>> A=creat_matrix(3,3)
>>> print_matrix(A)
   58.8    58.4    85.3
   47.8    3.90    88.6
   84.7    32.4    4.69
>>> invA=invMatrix(A)
>>> print_matrix(invA)
-0.00790  0.00689  0.01730
 0.02010 -0.01920 -0.01400
 0.00338  0.00841 -0.00235
>>> b=[1,1,1]
>>> Sol=sol_gauss(A,b)
>>> Sol
[0.0162796511658441,   -0.013060442022808835,
0.009445686290818843]
```

4. KẾT LUẬN

Dựa vào các thuật toán tính toán ma trận trong ĐSTT, tác giả đã xây dựng được modul Uneti_Matrix thực hiện một số tính toán ma trận bằng ngôn ngữ lập trình Python lõi. Các hàm số trong modul được lập trình một cách rõ ràng, bám sát vào các thuật toán tính toán ma trận trong ĐSTT nên dễ hiểu đối với sinh viên mới tiếp cận lập trình. Khi sử dụng modul Uneti_Matrix để minh hoạ một số tính toán ma trận trong ĐSTT cho sinh viên, không những rèn luyện tư lập trình và kỹ năng viết code bằng Python cho sinh viên mà còn giúp cho sinh viên hiểu sâu và nắm vững hơn những về ma trận trong ĐSTT.

TÀI LIỆU THAM KHẢO

[1] Phạm Văn Bằng, *"Tài liệu học tập Đại số tuyến tính",* Trường Đại học Kinh tế - Kỹ thuật Công nghiệp (2016).

[2] Lê Tuấn Hoa, *"Đại số tuyến tính qua các bài tập và ví dụ",* NXB Đại học Quốc gia Hà Nội (2005).

[3] Masoud Nosrati, *"Python: An appropriate language for real world programming",* World Applied Programming, Vol. 1, No. 2, pp. 110-117 (2011).

Thông tin liên hệ: **Chu Bình Minh**

Điện thoại: 0912207854 - Email: cbminh@uneti.edu.vn

Khoa Khoa học ứng dụng, Trường Đại học Kinh tế - Kỹ thuật Công nghiệp.