

Table of Contents

- 1 行列計算の概要
- 2 ガウス消去法による連立一次方程式の解
- 3 pythonによるLU分解
- 4 LU分解のコード
- 5 ピボット操作
- 6 反復法による連立方程式の解
- 7 課題
 - 7.1 行列AをLU分解
 - 7.2 pivot付きのLU分解
 - 7.3 Gauss-Seidel法

線形代数-逆行列

cc by Shigeto R. Nishitani 2017-21

行列計算の概要

数値計算の中心課題の一つである、行列に関する演算について見ていきます。多次元、大規模な行列に対する効率のよい計算法が多数開発されており、多くの既存のライブラリが用意されています。本章ではそれらの中心をなす、逆行列(matrix inverse)と固有値(Eigen values)に関して具体的な計算方法を示します。現実的な問題には既存のライブラリを使うのが上策ですが、それでも基礎となる原理の理解や、ちょっとした計算、ライブラリの結果の検証に使えるルーチンを示します。

逆行列は連立一次方程式を解くことと等価です。ルーチン的なやり方にガウスの消去法があります。これは上三角行列になれば代入を適宜おこなうことで解が容易に求めることができます。さらに、初期値から始めて次々に解に近づけていく反復法があります。この代表例であるJacobi(ヤコビ)法と、収束性を高めたGauss-Seidel(ガウス-ザイデル)法を紹介します。

上記の手法をより高速にした修正コレスキー分解と共役傾斜(共役勾配)法がありますが、複雑になるのでパスります。必要ならばNumRecipeを読んでください。

ガウス消去法による連立一次方程式の解

逆行列は連立一次方程式を解くことと等価です。すなわち、 A を行列、 x を未知数ベクトル、 b を数値ベクトルとすると、

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

です。未知数の少ない連立一次方程式では、適当に組み合わせて未知数を消していくべきですが、未知数が多くなってしまうと破綻します。未知数の多い多元連立一次方程式で、ルーチン的に解を求めていく方法がガウス消去法で、前進消去と後退代入という2つの操作からなります。

後退代入(Backward substitution)による解の求め方を先ず見ましょう。たとえば、

$$\begin{aligned} x + y - 2z &= -4 \\ -3y + 3z &= 9 \\ -2z &= -3 \end{aligned}$$

では、下から順番に $z \rightarrow y \rightarrow x$ と適宜代入することによって、簡単に解を求めることが出来ます。係数で作る行列でこのような形をした上三角行列にする操作を前進消去あるいはガウスの消去法(Gaussian elimination)といいます。下三角行列L(lower triangular matrix)と上三角行列U(upper triangular matrix)の積に分解する操作

$$A = L \cdot U$$

①拡大係数行列をつくる

をLU分解(LU decomposition)といいます。例えば先に示した上三角行列を係数とする連立方程式は、

$$\begin{aligned} x + y - 2z &= -4 \\ x - 2y + z &= 5 \\ x - 2y - z &= 2 \end{aligned}$$

を変形することで得られます。連立方程式からの変形を示してください。

$$\begin{array}{l} \text{①} \xrightarrow{\text{②} - \text{①}} \begin{matrix} x + y - 2z &= -4 \\ x - 2y + z &= 5 \\ x - 2y - z &= 2 \end{matrix} \\ \text{②} \xrightarrow{\text{③} - \text{②}} \begin{matrix} x + y - 2z &= -4 \\ 0 - 3y &= 3 \\ 0 - 3y &= 2 \end{matrix} \\ \text{③} \xrightarrow{\text{③} - \text{②}} \begin{matrix} x + y - 2z &= -4 \\ 0 - 3y &= 3 \\ 0 - 3y &= -1 \end{matrix} \end{array}$$

pythonによるLU分解

係数行列(coefficient matrix)Aと定数項(b)を定義します。

```
In [1]: import scipy.linalg as linalg # SciPy Linear Algebra Library
import numpy as np

# A = np.array([[1,1,-2,-4],[1,-2,1,5],[2,-2,-1,2]])
A = np.array([[1,1,-2],[1,-2,1],[1,-2,-1]])
b = np.array([[-4],[5],[2]])
print(A)
print(b)

[[ 1  1 -2]
 [ 1 -2  1]
 [ 1 -2 -1]]
[[-4]]
```

[5]
[2]
単に逆行列を求める際は

Matrix
Inverse

In [2]:
inv_A = linalg.inv(A)
print(inv_A)
np.dot(inv_A,b)

[[0.66666667 0.83333333 -0.5]
[0.33333333 0.16666667 -0.5]
[0. 0.5 -0.5]]

Out[2]: array([[0.5],
[-1.5],
[1.5]])

拡大係数行列はhstack(horizontal stack)かcolumn_stackを使います.

In [3]:
e_A = np.hstack((A, b))
print(e_A)
print(np.column_stack((A,b)))

[[1 1 -2 -4]
[1 -2 1 5]
[1 -2 -1 2]]
[[1 1 -2 -4]
[1 -2 1 5]
[1 -2 -1 2]]

水平

LU分解.

In [4]:
P, L, U = linalg.lu(e_A)
print("P:")
print(P)

print("L:")
print(L)

print("U:")
print(U)

P:
[[1. 0. 0.]
[0. 1. 0.]
[0. 0. 1.]]

L:
[[1. 0. 0.]
[1. 1. 0.]
[1. 1. 1.]]

U:
[[1. 1. -2. -4.]
[0. -3. 3. 9.]
[0. 0. -2. -3.]]

元の拡大行列はこれらの掛け算で求められます.

In [5]:
print("P.e_A=L.U:")
print(np.dot(P,A))
print(np.dot(L,U))

P.e_A=L.U:
[[1. 1. -2.]
[1. -2. 1.]

[1. -2. -1.]]
[[1. 1. -2. -4.]
[1. -2. 1. 5.]
[1. -2. -1. 2.]]

LU分解のコード

LU分解すれば線形方程式の解が容易に求まることは理解できると思う. 具体的にAをLU分解する行列(消去行列と称す)T1,T2の係数は次のようにして求められる.

In [6]:
import scipy.linalg as linalg # SciPy Linear Algebra Library
import numpy as np

→ A = np.array([[1.0,1,-2],[1,-2,1],[1,-2,-1]])
A0 = np.array([[1.0,1,-2],[1,-2,1],[1,-2,-1]])
→ b = np.array([-4.0,5,2])

n = 3
L = np.identity(n)
T = []
for i in range(n): # i行目
 T.append(np.identity(n))
 for j in range(i+1, n):
 am = A[j,i]/A[i,i] # i行の要素を使って, i+1行目の先頭を消す係数を求める
 T[i][j,i] = am # i番目の消去行列に要素を入れる
 L[j,i] = am # LTMの要素
 for k in range(n):
 A[j,k] = am*A[i,k] # もとの行列をUTMにしていく
 b[j] = b[i]*am # 数値ベクトルも操作
print(A)
print(b)

[[1. 1. -2.]
[0. -3. 3.]
[0. 0. -2.]]
[-4.]
[9.]
[-3.]

上のコードによって得られた消去行列.

In [7]:
print(T)

[array([[1., 0., 0.],
[-1., 1., 0.],
[-1., 0., 1.]]), array([[1., 0., 0.],
[0., 1., 0.],
[0., -1., 1.]]), array([[1., 0., 0.],
[0., 1., 0.],
[0., 0., 1.]])]

これを実際に元の行列A0に作用させると、UTMが求められる.

In [8]:
np.dot(T[1],np.dot(T[0],A0))

Out[8]: array([[1., 1., -2.],
[0., -3., 3.],
[0., 0., -2.]])

求められたLTM, UTMを掛けると

```
In [9]: np.dot(L,A)
```

```
Out[9]: array([[ 1.,  1., -2.],
   [ 1., -2.,  1.],
   [ 1., -2., -1.]])
```

元の行列を得られる。L,Aに求めたい行列が入っていることを確認。

```
In [10]: print(L)
print(A)
```

```
[[1. 0. 0.]
 [1. 1. 0.]
 [1. 1. 1.]]
 [[ 1.  1. -2.]
 [ 0. -3.  3.]
 [ 0.  0. -2.]]]
```

数値ベクトルも期待通り変換されている。

```
In [11]: print(b)
```

```
[-4.]
 [ 9.]
 [-3.]]
```

luの出力で得られる拡張係数行列のUTMは、次の通り。

```
In [12]: np.hstack((A, b))
```

```
Out[12]: array([[ 1.,  1., -2., -4.],
   [ 0., -3.,  3.,  9.],
   [ 0.,  0., -2., -3.]])
```

ピボット操作

ガウス消去法で困るのは、割ろうとした対角要素が0の場合である。

$$\begin{array}{l} 2x + y - 2z = -4 \\ -2x - y + z = 5 \\ 2x - 2y - z = 2 \end{array}$$

第1式で第2、第3式を掃き出すと

$$\begin{array}{l} x + y - 2z = -4 \\ \quad \quad \quad -z = 1 \\ \quad \quad \quad -3y + z = 6 \end{array}$$

となり、第2式で第3式が掃き出せなくなる。しかし、この場合にも、方程式の順序を、行列の行と右辺の値をペアにして入れ替えれば解決する。

$$\begin{array}{l} x + y - 2z = -4 \\ -3y + z = 6 \\ \quad \quad \quad -z = 1 \end{array}$$

これをピボット操作あるいはピボット(pivot, バスケの軸足を動かさず回すやつ)と呼ぶ。この操作は、変数の並びを変えたわけではなく、単に方程式の順番を変更する操作に相当

する。

さらに対角要素の数値が厳密に0でなくとも、極端に0に近づいた場合にも、その数で割った数値が大きくなり他の数との差を取ると以前に示した情報落ちの可能性が出てくる。この現象を防ぐためには、絶対値が最大のピボットを選んで行の入れ替えを毎回おこなうといい結果が得られることが知られている。

MapleのLUdecompositionコマンドをこのような行列に適用すると、置換行列(permuation matrix)Pが単位行列ではなく、ピボット操作に対応した行列となる。P.A=L.Uとなることに注意。

実際に先ほどの例をpythonで解かせると、置換行列Pはidentity行列ではなく、2、3行目でpivotが起こっていることを表している。

```
In [13]: A = np.array([[2,1,-2],[-2,-1,1],[2,-2,-1]])
b = np.array([-4.0,5,2])
e_A = np.hstack((A, b))
print(e_A)
P, L, U = linalg.lu(e_A)
```

```
print("P:")
print(P)

print("L:")
print(L)

print("U:")
print(U)
```

$$\begin{matrix} P & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 2 & 1 & -2 & -4 \\ -2 & -1 & 1 & 5 \\ 2 & -2 & -1 & 2 \end{pmatrix} = \begin{pmatrix} 2 & 1 & -2 & -4 \\ 2 & -2 & -1 & 2 \\ -2 & -1 & 1 & 5 \end{pmatrix} \\ L & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \text{ピボット} \\ U & \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} & \text{右辺} \end{matrix}$$

反復法による連立方程式の解

以下のような連立方程式を

$$\begin{bmatrix} 5x + y + z + u \\ x + 3y + z + u \\ x - 2y - 9z + u \\ x + 3y - 2z + 5u \end{bmatrix} = \begin{bmatrix} -6 \\ 2 \\ -7 \\ 3 \end{bmatrix}$$

$$3y = 2 - x - z - u$$

$$y = \frac{2 - x - z - u}{3}$$

$$-9z = -7 - x + 2y - u$$

$$z = \frac{-7 - x + 2y - u}{9}$$

$$5u = 3 - x - 3y + 2z$$

$$u = \frac{3 - x - 3y + 2z}{5}$$

形式的に解くと

$$x = \frac{-6 - (y + z + u)}{5}$$

となる。他の未知数も、

$$\begin{aligned} y &= \\ z &= \\ u &= \end{aligned}$$

となる。適当に初期値 (x_0, y_0, z_0, u_0) をとり、下側の方程式に代入すると、得られた出力 (x_1, y_1, z_1, u_1) はより精解に近い値となる。これを繰り返すことによって精解が得られる。これをヤコビ(Jacobi)法と呼び、係数行列の対角要素が非対角要素にくらべて大きいときに適用できる。多くの現実の問題ではこの状況が成り立っている。

Gauss-Seidel法はJacobi法の高速版である。 n 番目の解の組が得られた後に一度に次の解の組に入れ替えるのではなく、得られた解を順次改良した解として使っていく。これにより、収束が早まる。以下にはヤコビ法のコードを示した。Gauss-Seidel法となる。

In [14]:

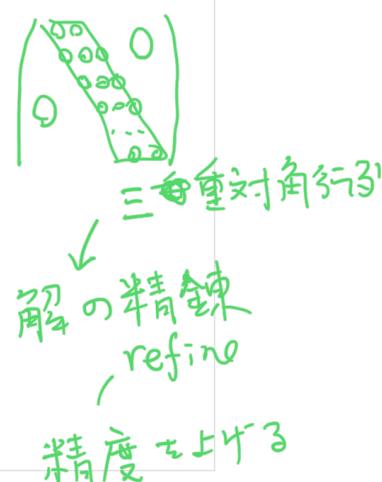
```
# Jacobi iterative method for solving Ax=b by bob
import numpy as np
np.set_printoptions(precision=6, suppress=True)
```

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

```
x0=np.zeros(n)
x1=np.zeros(n)
```

```
for iter in range(0, 20):
    for i in range(0, n):
        x1[i]=b[i]
        for j in range(0, n):
            x1[i]=x1[i]-A[i][j]*x0[j]
            x1[i]=x1[i]+A[i][i]*x0[i]
            x1[i]=x1[i]/A[i][i]
        for j in range(0, n):
            x0[j]=x1[j]
    print(x0)
```

[-1.2 0.666667 0.777778 0.6]
[-1.608889 0.607407 0.562963 0.751111]
[-1.584296 0.764938 0.54749 0.782519]
[-1.618989 0.751429 0.518705 0.676892]
[-1.589405 0.807797 0.506116 0.680422]
[-1.598867 0.800956 0.497269 0.635649]
[-1.586775 0.821983 0.492763 0.638108]
[-1.590571 0.818635 0.489707 0.621271]
[-1.585923 0.826531 0.488159 0.622816]
[-1.587501 0.824982 0.487092 0.61653]
[-1.585721 0.82796 0.486563 0.617348]
[-1.586374 0.82727 0.48619 0.614993]
[-1.585691 0.828397 0.486009 0.615389]
[-1.585959 0.828098 0.485878 0.614503]
[-1.585696 0.828526 0.485817 0.614684]
[-1.585805 0.828898 0.485771 0.61435]
[-1.585704 0.828561 0.48575 0.61443]



```
[ -1.585748 0.828508 0.485734 0.614304 ]
[ -1.585709 0.82857 0.485727 0.614338 ]
[ -1.585727 0.828548 0.485721 0.61429 ]
```

もう少しpython(numpy)の機能を使うとおしゃれな書き方ができる。以下は、ネットから拾ってきたcode。上(by bob)とやってることの本質的なところは同じですが、numpyのindexを自動的に判断する機能を使って簡潔に書いています。こういうのを読める、書けるようになれば、あなたもpython使い。私はまだpythonに馴染めてなくて、いちいち頭の中で治しています。

In [15]:

```
# https://www.quantstart.com/articles/Jacobi-Method-in-Python-and-Numpy
from numpy import array, zeros, diag, diagflat, dot
```

```
np.set_printoptions(precision=6, suppress=True)
```

```
def jacobi(A,b,N=None,x=None):
```

```
if x is None:
    x = zeros(len(A[0]))
```

```
D = diag(A)
R = A - diagflat(D)
# Iterate for N times
for i in range(N):
    x = (b - dot(R,x)) / D
    print(x)
return x
```

```
A = array([[5,1,1,1],[1,3,1,1],[1,-2,-9,1],[1,3,-2,5]])
b = array([-6,2,-7,3])
```

```
sol = jacobi(A,b,N=10)
```

```
[ -1.2 0.666667 0.777778 0.6 ]
[ -1.608889 0.607407 0.562963 0.751111 ]
[ -1.584296 0.764938 0.54749 0.782519 ]
[ -1.618989 0.751429 0.518705 0.676892 ]
[ -1.589405 0.807797 0.506116 0.680422 ]
[ -1.598867 0.800956 0.497269 0.635649 ]
[ -1.586775 0.821983 0.492763 0.638108 ]
[ -1.590571 0.818635 0.489707 0.621271 ]
[ -1.585923 0.826531 0.488159 0.622816 ]
[ -1.587501 0.824982 0.487092 0.61653 ]
```

課題

① 行列AをLU分解

次の行列AをLU分解し、上・下三角行列を求めよ。さらに連立方程式の解を求めよ。

$$\begin{bmatrix} x + 4y + 3z \\ x - 2y + z \\ 2x - 2y - z \end{bmatrix} = \begin{bmatrix} 11 \\ 11 \\ 11 \end{bmatrix}$$

② pivot付きのLU分解

次の連立方程式の拡大行列をLU分解せよ。pivot操作が必要となる。P.A = L.Uを確かめよ。

$$\rightarrow \begin{bmatrix} 3w + 2x + 2y + z \\ 3w + 2x + 3y + z \\ w - 2x - 3y + z \\ 5w + 3x - 2y + 5z \end{bmatrix} = \begin{bmatrix} -6 \\ 2 \\ -9 \\ 2 \end{bmatrix}$$

③ Gauss-Seidel法

Jacobi法のプログラムを参照してGauss-Seidel法のプログラムを作れ。

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

に対して計算を行い、Jacobi法と収束性を比べよ。

最初の数回はこんな感じ。

→ [-1.2 1.066667 0.407407 0.362963]
[-1.567407 0.932346 0.436763 0.528779]
[-1.579578 0.871345 0.46739 0.580064]
[-1.58376 0.845435 0.478382 0.600844]
[-1.584932 0.835236 0.482827 0.608976]

In []: