スクリプト言語プログラミング
Pythonによる数値解析

第13回講義資料

箕原辰夫

連立方程式の解法

- 直接解法（direct method）と反復解法（iterative method）がある
- 直接解法
  - ガウスの消去法（Gauss elimination method）
  - ガウス・ジョルダン法（Gauss Jordan method）
  - 三角分解法（LU分解：LU factorization、コレスキ―分解：Cholesky factorization、LDL分解）
- 反復解法（iteration method）…元数が多いときに用いられる
  - ヤコビ（Jacobi）の反復法
  - ガウス・ザイデル（Gauss Seidel）の反復法
  - 共役勾配法（conjugate gradient method)

ガウスの消去法

- ガウスの消去法（Gaussian elimination）は、連立一次方程式を解く一般的な解法である。
- n元m立一次方程式を考える。右側が対応する拡大係数行列（argumented matrix）となる。

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \quad \left[\begin{array}{ccccc} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{array}\right]$$

- ガウスの消去法は、連立一次方程式の解法以外にも以下の用途にも用いられる。
  - 行列の階数の計算
  - 正則行列の逆行列の計算（ガウス・ジョルダン法）

ガウスの消去法の指針

- 行列に対してガウスの消去法を適用する方法は、行に関する基本変形を行列に可能な限り繰り返し行なって行列の左下部分の成分を全て 0 にして行列を上三角行列（upper triangle matrix）に変形していく。
- 行に関する基本変形には、以下のの 3 種類の操作あるいは変形がある。
  - 二つの行を入れ替えるもの
  - ある行を0でない定数倍するもの
  - ある行に他のある行の定数倍を加えるもの

ガウスの消去法の方法

- 前進消去（forward elimination）
  - 拡大係数行列を上三角形に変形するもので、以下のような漸化式で行列の要素を変形する（ここでは、行と列の添え字は1から始まるものとする）

$$a'_{ij} = a_{ij} - a_{im}\frac{a_{mj}}{a_{mm}}, \quad m = 1,2,\cdots n-1 \quad i = m+1, m+2, \cdots n \quad j = m+1, m+2, \cdots, n+1$$

  - なお、上記の漸化式において、j=mから始めると、左下が0になる。j=m+1から始めても、計算自体は求められる
- 後退代入（back substitution）
  - 上三角に変形された行列から、下の行から各変数の値を求めていくもので、以下のような式で求めることができる

$$\begin{cases} x_n = \dfrac{a'_{n,n+1}}{a'_{nn}} \\ x_i = \dfrac{a'_{i,n+1} - \displaystyle\sum_{j=1}^{n-i} a'_{i,i+j} x_{i+j}}{a'_{ij}} \quad i = n-1, \cdots, 2, 1 \end{cases}$$

ガウスの消去法の例

- 以下の連立方程式を解く
  - 2x - 3y + z = 1
  - x + 2y - 3z = 4
  - 3x + 2y -z = 5

- m = 0...2までの係数行列の変化

⎡ 2.00 -3.00 1.00 1.00 ⎤ (1)
⎢ 1.00 2.00 -3.00 4.00 ⎥ (2)
⎣ 3.00 2.00 -1.00 5.00 ⎦ (3)

⎡ 2.00 -3.00 1.00 1.00 ⎤
⎢ 0.00 3.50 -3.50 3.50 ⎥ (2') = (2) - (1) × 1/2
⎣ 0.00 6.50 -2.50 3.50 ⎦ (3') = (3) - (1) × 3/2

⎡ 2.00 -3.00 1.00 1.00 ⎤
⎢ 0.00 3.50 -3.50 3.50 ⎥
⎣ 0.00 0.00 4.00 -3.00 ⎦ (3'') = (3') - (2') × 6.5 / 3.5

- 答え：[x, y, z] = [(1+3*1/4+1*3/4)/2, (3.5+3.5*-3/4)/3.5, -3/4] = [1.25, 0.25, -0.75]

Gaussの消去法のアルゴリズム（前進消去）

- Python的に記述すると、次のような形になる、aは拡大係数行列とする：a[0～n-1]が、行列の各行、a[i][0～n-1]が各変数への係数項、a[i][n]が定数項とする

- 前進消去の部分

```
for m in range( n-1 ): # 0行からn-2行まで
    for i in range( m+1, n ): # m+1行からn-1行まで
        r = a[ i ][ m ] / a[ m ][ m ] # ピボットから係数を求める
        for j in range( m, n+1 ): # m列からn列まで
            a[ i ][ j ] -= a[ m ][ j ] * r  # ピボットのある行の係数倍を引く
```

pivotの選択（pivoting）

- 前進消去を進める際に、行列の対角要素の係数項が0になってしまい、求められない場合がある
- 対角要素の係数項の絶対値が小さい場合も、桁落ちが発生してしまい誤差が増大することが考えられる
- 前進消去に使われる対角要素のことをピボット（pivot）と呼ぶが、ピボットとして選ばれる係数項の絶対値が最大になるように、同じ列の別の行と取り換えるという操作を行なう
- この処理を、枢軸選択法（pivotal elimination method）あるいはピボッティング（pivoting）と呼ぶ
- ピボット選択を行なう、0となる対角要素がある場合でも前進消去が可能であるし、対角要素の係数項の絶対値が小さい場合でも、精度をあげることができる
- 他の行の同じ列で取り換えるものを部分的ピボッティング（partial pivoting）と呼び、行と列の両方で入れ替えを行なうものを完全ピボッティング（complete pivoting）と呼ぶ。

列を入れ替えた場合、変数の順番も交替になるので、注意が必要である。

- Pythonでのプログラミング例

```
def pivoting( a, m ):
# 拡大係数行列のm列目のpivotingを行なう

    maxi = m # 仮にm行目が最大値を持つとする

    for p in range( m+1, n ):
      # m行以降、最大値のある行のインデックスを
       maxiに求める

       if abs( a[p][m] ) > abs( a[ maxi ][ m ] ):  maxi = p
       # より絶対値が大きい係数項が見つかったら乗り換え

    if a[ maxi ][ m ]  == 0 : raise Exception
    # m列目の係数がすべて0の場合

    a[ maxi ], a[ m ] = a[ m ], a[ maxi ]  # 行の入れ替え
```

Gaussの消去法のアルゴリズム（後退代入）

- 後退代入の部分の漸化式から各変数の値を求めていく。一番下の行から求めていく形になる

- Pythonでの記述例: x[0〜n-1]に解を求める

```
if a[ n-1 ][ n-1 ] == 0:  raise Exception # 一意の解なし
x[ n-1 ]  = a[ n-1 ][ n ] / a[ n-1 ][ n-1 ]  # 最初の解
for i in range( n-2, -1, -1 ): # 下から上に向かって
    summ = 0
    for j in range( i, n ): # 既知の変数の値を使って足していく
        summ += a[ i ][ j ] * x[ j ]
    x[ i ] = (  a[ i ][ n ]  - summ ) / a[ i ][ i ]
```

ガウスの消去法による行列の階数の計算

- 係数行列にガウスの前進消去を適用した場合、下の方の行の要素がすべて0になってしまう場合がある。このときは、一意の解は求まらない。
- 0でない要素が詰まっている行までの行数を数えると、それが行列の階数（rank）になっている。
- 行列にガウスの前進消去を適用すると上三角行列になるが、上三角行列の場合は、一般的に、その対角成分の積を取れば、その上三角行列の行列式を求めることができる（元の行列の行列式が求められるわけではない）

ガウス・ジョルダン法

- ガウス・ジョルダン法（Gauss-Jordan method）は、掃出し法（sweeping-out method, row reduction）とも呼ばれる
- ガウスの消去法と似ているが、ガウスの消去法での前進消去を、ピボット行を除くすべての行に適用したものになる、ピボット行はピボットの値で割っていく
- 次の漸化式を拡大係数行列に適用し、各要素を求めていく

$$
\begin{cases}
a'_{ij} = \dfrac{a_{ij}}{a_{mm}} & (i = m) \\
a'_{ij} = a_{ij} - a_{mj}\dfrac{a_{im}}{a_{mm}} & (i \neq m)
\end{cases}
\qquad m = 1,2,\cdots,n
$$

- 拡大係数行列に適用した結果は、次のような行列になる

$$
\begin{bmatrix}
1 & 0 & \cdots & 0 & a^{(n)}_{1,n+1} \\
0 & 1 & \cdots & 0 & a^{(n)}_{2,n+1} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 1 & a^{(n)}_{n,n+1}
\end{bmatrix}
\quad where \quad a'_{ij} = a^{(1)}_{ij}, a''_{ij} = a^{(2)}_{ij}, \cdots
$$

ガウス・ジョルダン法のアルゴリズム

- 基本的には、前進消去が中心となる（解は、拡大係数行列の定数項の縦の列に求められる）
- Pythonでの記述例：
- aは拡大係数行列とする：a[0〜n-1]が、行列の各行、a[i][0〜n-1]が各変数への係数項、a[i][n]が定数項とする

```
for m in range( n ):
    for i in range( n ):
        r = a[ i ][ m ] / a[ m ][ m ] if m != i else 1.0 / a[ m ][ m ]
        for j in range( len(a[i] )):
            a[i][j] = a[i][j] * r  if m == i  else  a[i][j] - a[m][j] * r
```

ガウス・ジョルダン法の例

- 以下の連立方程式を解く
  - 2x - 3y + z = 1
  - x + 2y - 3z = 4
  - 3x + 2y -z = 5

- m = 0...2までの係数行列の変化

⎡ 2.00 -3.00 1.00 1.00 ⎤ (1)
⎢ 1.00 2.00 -3.00 4.00 ⎥ (2)
⎣ 3.00 2.00 -1.00 5.00 ⎦ (3)

⎡ 1.00 -1.50 0.50 0.50 ⎤ (1') = (1) / 2
⎢ 0.00 3.50 -3.50 3.50 ⎥ (2') = (2) - (1) / 2
⎣ 0.00 6.50 -2.50 3.50 ⎦ (3') = (3) - (1) × 3 / 2

⎡ 1.00 0.00 -1.00 2.00 ⎤ (1'') = (1') - (2') × -1.5 / 3.5
⎢ 0.00 1.00 -1.00 1.00 ⎥ (2'') = (2') / 3.5
⎣ 0.00 0.00 4.00 -3.00 ⎦ (3'') = (3') - (2') × 6.5 / 3.5

⎡ 1.00 0.00 0.00 1.25 ⎤ (1''') = (1'') - (3'') × -1.0 / 4.0
⎢ 0.00 1.00 0.00 0.25 ⎥ (2''') = (2'') - (3'') × -1.0 / 4.0
⎣ 0.00 0.00 1.00 -0.75 ⎦ (3''') = (3'') / 4

逆行列の求め方

- ガウス・ジョルダン法を用いて、n次の正方行列の逆行列（正則行列の場合）を求める方法
- 正方行列Aの行列式|A|について
  - |A|=0のとき、Aを特異行列（singular matrix）
  - |A|≠0のとき、Aを正則行列（regular matrix）と呼ぶ
- AX = Iを満足する行列を求める、拡大行列[A｜I]に対してガウス・ジョルダン法を用いて、[I｜A']になるように変形する。このときのA' = A^{-1}となる

ガウスジョルダン法で逆行列を求めた例

- 次の3次の正方行列の逆行列を求める

⎡ 2.0 1.0 7.0 ⎤
⎢ 9.0 5.0 6.0 ⎥
⎣ 2.0 1.0 6.0 ⎦

- m=0...n-1での行列の変遷

⎡ 2.0 1.0 7.0 1.0 0.0 0.0 ⎤ (1)
⎢ 9.0 5.0 6.0 0.0 1.0 0.0 ⎥ (2)
⎣ 2.0 1.0 6.0 0.0 0.0 1.0 ⎦ (3)

⎡ 1.0 0.5   3.5 0.5 0.0 0.0 ⎤ (1') = (1) / 2
⎢ 0.0 0.5 -25.5 -4.5 1.0 0.0 ⎥ (2') = (2) - (1) × -9/2
⎣ 0.0 0.0 -1.0 -1.0 0.0 1.0 ⎦ (3') = (3) - (1) × -2/2

⎡ 1.0 0.0 29.0 5.0 -1.0 0.0 ⎤ (1'') = (1') - (2') × 0.5/0.5
⎢ 0.0 1.0 -51.0 -9.0 2.0 0.0 ⎥ (2'') = (2') / 0.5
⎣ 0.0 0.0 -1.0 -1.0 0.0 1.0 ⎦ (3'') = (3') - (2') × -0.0/0.5

⎡ 1.0 0.0 0.0 -24.0 -1.0 29.0 ⎤ (1''') = (1'') - (3'') × 29.0/-1.0
⎢ 0.0 1.0 0.0 42.0 2.0 -51.0 ⎥ (2''') = (2'') - (3'') × -51.0/-1.0
⎣-0.0 -0.0 1.0   1.0 -0.0 -1.0 ⎦ (3''') = (3'') / -1.0

numpyにおける連立一次方程式の解

- numpy.linalgパッケージのsolve関数を用いる
- 使用例：

```
import numpy as np

a = np.array( [[3, 2, 0], [1, -1, 0], [0, 5, 1]] )
b = np.array( [2, 4, -1] )
x = np.linalg.solve( a, b )
print( x )
```

- 結果：

```
[ 2. -2.  9.]
```

scipyにおける連立一次方程式の解

- scipyでは、linalgパッケージのsolve関数が連立一次方程式の解を求める。入力パラメータや、結果は、numpyの配列を用いる
- たとえば、以下のように記述する

```
import numpy as np
from scipy import linalg as linalg

a = np.array( [[3, 2, 0], [1, -1, 0], [0, 5, 1]] )
b = np.array( [2, 4, -1] )
x = linalg.solve( a, b )
print( x )
```

→ array([2., -2., 9.])

sympyにおける連立一次方程式の求解

- solve関数は、連立方程式にも適用できる
- 例：

```
import sympy
sympy.var('x, y')
eq1=sympy.Eq(2*x+1*y, 3)
eq2=sympy.Eq(1*x+3*y, 4)
sympy.solve ([eq1, eq2], [x, y])
```

- また、2次以上の連立方程式にも適用が可能になっている。
- 例：

```
import sympy
sympy.var('x, y, a, b, c, g, h')
eq3=sympy.Eq(y, a*x**2+b*x+c)
eq4=sympy.Eq(y, g*x+h)
sympy.solve ([eq3, eq4], [x, y])
```

- 参考：https://pianofisica.hatenablog.com/entry/2019/04/04/233515

sympyにおける行列を用いた連立一次方程式の求解

- Gauss Jordan法による求解の関数が用意されている
- 行列.gauss_jordan_solve(定数項の列ベクトル)
- 戻り値は、(解の列ベクトル, パラメータ)のタプルで、パラメータは、解が一意に定まらないときの、変数項を示す
- 使用例：

```
from sympy import Matrix
A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
B = Matrix([3, 6, 9])
sol, params = A.gauss_jordan_solve(B)
sol
→  Matrix([[-1],[ 2],[ 0]])
```

LU分解

- LU分解とは、n元一次連立方程式の係数行列である正方行列Aを下三角行列Lおよび上三角行列Uに分解することである
- Uの対角成分を1にする方法は、クラウト法（Cround Method）と呼ばれ、Lの対角成分を1にする方法は、ドゥーリトル法（Doolittle Method）と呼ぶ
- A=LUを保つ

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = LU = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & \cdots & u_{1n} \\ 0 & 1 & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

- LU分解をすると、次にような応用が利く
  - 連立一次方程式の解をLUで求めることができる
  - 逆行列を求めることができる
  - 行列式を簡単に求めることができる

LU分解のための計算式

- 行列Lと行列Uの各要素は以下の式によって計算できる（クラウト法：Crout Method）

$$
\begin{cases}
l_{ij} = 0 & (i < j) \\
l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} & (i \geq j)
\end{cases}
$$

$$
\begin{cases}
u_{ij} = 0 & (i > j) \\
u_{ij} = 1 & (i = j) \\
u_{ij} = \dfrac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}}{l_{ii}} & (i < j)
\end{cases}
$$

- なお対角成分の1は、並べ替え行列（Permutation）Pによって、Lの方に入れられることもある、その際は、$PA = LU$という形になる

LU分解による連立1次方程式の解法

- LU分解された行列Aにおける連立1次方程式は、以下の式のように記述できる

$$A\boldsymbol{x} = LU\boldsymbol{x} = \boldsymbol{b}$$

- $U\boldsymbol{x}=\boldsymbol{y}$とおくと、次のように記述できる

$$\begin{cases} L\boldsymbol{y} = \boldsymbol{b} \\ U\boldsymbol{x} = \boldsymbol{y} \end{cases}$$

- これらの式から以下の手順で解を求める

1. Lと定数項の列ベクトル$\boldsymbol{b}$から、$\boldsymbol{y}$の列ベクトルを求める（前進消去）
2. Uと列ベクトル$\boldsymbol{y}$から、$\boldsymbol{x}$の列ベクトルを求める（後退代入）

- それぞれの列ベクトルの各要素は以下のように求める

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} l_{ij} y_j}{l_{ii}} \quad (i = 1, \cdots, n)$$

$$x_i = y_i - \sum_{j=i+1}^{n} u_{ij} x_j \quad (i = n, n-1, \cdots 1)$$

LU分解による行列式と逆行列の求め方

- 行列式は、以下のように求められる
（クラウト法の場合）

$$\det(A) = \det(LU) = \det(L) \cdot \det(U) = l_{11} l_{22} \cdots l_{nn}$$

- 逆行列は、以下の式からLとUの逆行列の積として求められる

$$A^{-1} = (LU)^{-1} = U^{-1} L^{-1}$$

- クラウト法に場合、それぞれの逆行列は以下のようにして求められる

$$L^{-1} : ll_{ij} = \begin{cases} if\ i = j : \dfrac{1}{l_{ij}} \\ if\ i < j : 0 \\ if\ i > j : -\dfrac{\sum_{k=1}^{j-1} l_{jk} ll_{ki}}{l_{ii}} \end{cases}$$

$$U^{-1} : uu_{ij} = \begin{cases} if\ i = j : 1 \\ if\ i < j : 0 \\ if\ i > j : -\sum_{k=j}^{i-1} u_{jk} uu_{ki} \end{cases}$$

参照：https://qiita.com/mnanri/items/f0e9b20395545dd674c9

scipyによるLU分解と求解

- linalgパッケージにLU分解をするlu関数がある $PA = LU$で求められるP, L, Uを返す
- linalgパッケージには、更に、lu_factor関数によってLU分解された行列（右上三角部分がU、左下三角部分がL）を用いて、解を求めるlu_solve関数がある。
- 例：

```
import scipy.linalg as linalg
import numpy as np

A = np.array([[6, 4, 1], [1, 8, -2], [3, 2, 0]])
b = np.array([7, 6, 8])

p, l, u = linalg.lu( A )  # LU decomposition
LU, piv = linalg.lu_factor( A )  # LU factorization, pivot indices
x = linalg.lu_solve( LU, b )  # LU solve with LU factorization
print(LU, x)
```

https://org-technology.com/posts/solving-linear-equations-LU.html

sympyによるLU分解と求解

- **from** sympy.matrices **import** *
- 行列.LUdecomposition()…LU分解された行列のタプルを返す

  A.LUdecomposition()では、PA = LUとなるようなL, U, Pを返す。

  PはPermutationであり、

  eye(A.row).permuteFwd(P)で計算される、Pが必要ないときは空リストで返される

- 行列.LUSolve(定数項列ベクトル)…

  LU分解による変数の求解

- 使用例：

```
a = Matrix( cofmat )
b = Matrix( 3, 1, conmat )
l,u,p = a.LUdecomposition( )
for m in [l,u,p]: print( m )
x = a.LUsolve( b )
print( x )
```

コレスキー分解

- 対角対称形となる正方行列（対称行列）について、コレスキー分解（Cholesky decomposition）は、三角分解の1つで、$A=LL^T$ となるような三角行列Lとその転置行列L^Tの積で正方行列Aを表わす分解の仕方である

$$A=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = LL^{\mathrm{T}} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{nn} \end{bmatrix}$$

- コレスキー分解の下三角行列の各要素は以下のように求めることができる（Cholesky–Banachiewicz および Cholesky–Crout アルゴリズム）

$$\begin{cases} l_{jj} = \sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} & (i = j) \\ l_{ij} = \dfrac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk}}{l_{jj}} & (j < i \leq n) \end{cases}$$

コレスキー分解による求解

- コレスキー分解された行列Aにおける連立1次方程式は、以下の式のように記述できる

$$A\boldsymbol{x} = LL^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{b}$$

- $U\boldsymbol{x}=\boldsymbol{y}$とおくと、次のように記述できる

$$\begin{cases} L\boldsymbol{y} = \boldsymbol{b} \\ L^{\mathrm{T}}\boldsymbol{x} = \boldsymbol{y} \end{cases}$$

- これらの式から以下の手順で解を求める

1. Lと定数項の列ベクトル$\boldsymbol{b}$から、$\boldsymbol{y}$の列ベクトルを求める（前進消去）
2. L^Tと列ベクトル$\boldsymbol{y}$から、$\boldsymbol{x}$の列ベクトルを求める（後退代入）

- それぞれの列ベクトルの各要素は以下のように求める

$$y_i = \frac{b_i - \sum_{j=1}^{i-1} l_{ij} y_j}{l_{ii}} \quad (i = 1, \cdots, n)$$

$$x_i = y_i - \sum_{j=i+1}^{n} l_{ji} x_j \quad (i = n, n-1, \cdots 1)$$

scipy, sympyのコレスキー分解と求解

- 正定値（positive definite）の対称行列が対象となる。
- 正定値は、零ベクトルでない任意のベクトル**x**とその転置ベクトル$\mathbf{x}^T$とその行列Aを乗算したときに、いつでもその値が正になるもの（$\mathbf{x}^T A \mathbf{x} > 0$）
- 正定値の対称行列では、対角成分の要素の値の方が、周辺の値よりも大きい傾向がある
- 正定値の対称行列かどうかチェックする方法は、以下を参照

  https://www.gaussianwaves.com/2013/04/tests-for-positive-definiteness-of-a-matrix/

- scipy

```
import scipy.linalg as linalg
L = linalg.cholesky( A, lower=True )
L, low = linalg.cho_factor( A )
x = scipy.linalg.cho_solve( (L, low), b )
```

- sympy

```
from sympy.matrices import Matrix
L = A.cholesky( )
x = A.cholesky_solve( b )
```

LDL分解

- 対称行列Aに対して、コレスキー分解のLL^Tではなくて、間にDという対角行列をいれて、
  $A = LDL^T$という形に分解して解を求める
- 修正コレスキー分解（modified Cholesky decomposition）あるいは、LDL（LDLT）分解と呼ばれている

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} = LDL^{\mathrm{T}} = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \vdots & 0 \\ \vdots & \cdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_n \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & \cdots & l_{n1} \\ 0 & l_{22} & \cdots & l_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & l_{nn} \end{bmatrix}$$

- Aが大型で成分に0が多い疎行列の時、特定の要素を強制的に0とおき、近似的に$A \approx LDL^T$（あるいは$A = LDL^T + N$：Nは残差）が成り立つように分解する。
- 疎行列のときに使われるこの分解は、不完全コレスキー分解（incomplete Cholesky decompostion）と呼ばれる

LDL分解の仕方

- LとDの各要素を行列Aから以下のようにして求める

$$d_1 = a_{11}, \quad l_{11} = 1$$

- $k = 2, 3, \ldots, n$において

$$\begin{cases} d_k = a_{kk} - \sum_{j=1}^{i-1} l_{kj}^2 d_j & (i = k) \\ l_{kk} = 1 & (i = k) \\ l_{ki} = \dfrac{a_{ki} - \sum_{j=1}^{i-1} l_{kj} l_{ij} d_j}{d_i} & (i = 1, 2, \cdots, k-1) \end{cases}$$

LDL分解による求解

- LDL^Tの対角行列Dの対角成分の各要素の平方根を持つ行列を$D^{\frac{1}{2}} = \mathrm{eye}(\sqrt{d_{ii}})$とする。
- Dは対角行列なので、$D^{\frac{1}{2}} = (D^{\frac{1}{2}})^{\mathrm{T}}$であり、以下の式が成り立つ

$$LDL^T = LD^{\frac{1}{2}}D^{\frac{1}{2}}L^T = \left(LD^{\frac{1}{2}}\right)\left(D^{\frac{1}{2}}L\right)^T$$

- $A\boldsymbol{x} = \boldsymbol{b}$を$(LD^{\frac{1}{2}})^{\mathrm{T}}$を用いて次のように変形する

$$A\boldsymbol{x} = A((LD^{\frac{1}{2}})^T)^{-1}(LD^{\frac{1}{2}})^T\boldsymbol{x} = \boldsymbol{b}$$

- この両辺に$(LD^{\frac{1}{2}})^{-1}$を掛けると次の式が得られる

$$(LD^{\frac{1}{2}})^{-1}A((LD^{\frac{1}{2}})^T)^{-1}(LD^{\frac{1}{2}})^T\boldsymbol{x} = (LD^{\frac{1}{2}})^{-1}\boldsymbol{b}$$

- この式の$(LD^{\frac{1}{2}})^{-1}A((LD^{\frac{1}{2}})^T)^{-1}$は対角行列になる。この行列をBとする

$$B = (LD^{\frac{1}{2}})^{-1}A((LD^{\frac{1}{2}})^T)^{-1}$$

- このBを用いて、$A\boldsymbol{x} = \boldsymbol{b}$は、LU分解やコレスキー分解と同様に以下のように記述できる

$$\begin{cases} (LD^{\frac{1}{2}})^T\boldsymbol{x} = \boldsymbol{y} \\ B\boldsymbol{y} = (LD^{\frac{1}{2}})^{-1}\boldsymbol{b} \end{cases}$$

Scipy, SympyのLDL分解とLDL求解

- Scipy

```
import numpy as np

from scipy import linalg

A = np.array( [ [25, 15, -5], [15, 18, 0], [-5, 0, 11] ] )

L, D, P = linalg.ldl( A )

print( L.dot(D).dot(L.T) )  # L @ D @ L.T


from scipy.linalg.lapack import dsysv

lult, piv, x, _ = dsysv(A, b, lower=1)
```

- Sympy

```
from sympy.matrices import Matrix

A = Matrix(((25, 15, -5), (15, 18, 0), (-5, 0, 11)))

L, D = A.LDLdecomposition( )

print( L * D * L.T )

x = A.LDLsolve( B )
```

https://stackoverflow.com/questions/47258075/ldlt-factorization-using-scipys-python-bindings-to-lapack

numpy, scipy, sympyによる逆行列の求値方法

- numpy
  - numpy.linalg.inv(行列)
  - numpy.matrixのオブジェクト.I
- scipy
  - scipy.linalg.inv(行列)
- sympy
  - 行列.inv(method=None)
  - method=Noneのときは、ガウスの消去法による、なお行列が疎行列（SparseMatrix）のときは、標準でQR分解（inverse_QR()関数と同じ)が用いられる
  - 以下のメソッド（文字列で名前を指定）が使える
    - "GE"…Gaussian Elimination, inverse_GE()と同じ
    - "LU"…LU Decomposition, inverse_LU()と同じ
    - "ADJ"…ADJugate matrix and a determinant, inverse_ADJ()と同じ
    - "CH"…Cholesky decomposition, inverse_CH()と同じ
    - "LDL"…LDL decomposition, inverse_LDL()と同じ

反復法による連立1次方程式の求解

- 元数（変数）が多い連立1次方程式の場合、係数行列Aの要素には0が含まれる場合が多い
- 行列の要素に0が含まれる割合が多い行列は、疎行列（sparse matrix）と呼ばれる
- 係数行列Aが疎行列であり、元数が多い場合は、求解をするために、反復法（iteration method）が用いられる
- 以下に4つの反復法を用いた求解のアルゴリズムを紹介する
  - ヤコビの反復法
  - ガウス・ザイデルの反復法
  - 共役勾配法
  - ICCG法

ヤコビの反復法

- n元n立一次方程式を考える。

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

- この対角項の係数について $a_{ii} \neq 0 \quad (i = 1,2,\cdots n)$ ならば、x_iについて次の式が得られる

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}}{a_{ii}} \quad (k = 0,1,2,3,\cdots)$$

- ここで、$x_i^{(0)}$はx_iの初期値である。この初期値$x_i^{(0)}$には、適切な値が代入されているとする

- この式に基づいて、x_iを求めていく方法は、ヤコビの反復法（Jacobi method）と呼ばれる
- 反復計算の終了を判定する条件式は、以下のいずれかの式による。εは許容誤差

$$\sum_{i=1}^{n} \left| x_i^{(k+1)} - x_i^{(k)} \right| \leq \varepsilon \quad \sum_{i=1}^{n} \left| \frac{x_i^{(k+1)} - x_i^{(k)}}{x_i^{(k+1)}} \right| \leq \varepsilon$$

- 方程式がヤコビの反復法によって収束する条件は以下のようになる。この条件を対角優位（diagonal dominant）と呼び、この条件が満たされていれば、反復法で解が求まる

$$\sum_{j=1}^{i-1} \left| \frac{a_{ij}}{a_{ii}} \right| + \sum_{j=i+1}^{n} \left| \frac{a_{ij}}{a_{ii}} \right| < 1 \quad (i = 1,2,\cdots,n)$$

ガウス・ザイデルの反復法

- ヤコビの反復法に対して、右辺の $x_j^{(k)}$ を最も新しい $x_j^{(k+1)}$ $(j<i)$ で置き換えて計算する方法を、ガウス・ザイデルの反復法（Gauss-Seidel method）と呼ぶ
- $x_i^{(k+1)}$ の計算式は、以下のようになる

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}} \quad (k = 0,1,2,3,\cdots)$$

- 収束判定は、ヤコビの反復法と同じもの（以下の条件式のいずれか）が使える。

$$\sum_{i=1}^{n} \left| x_i^{(k+1)} - x_i^{(k)} \right| \le \varepsilon \quad \sum_{i=1}^{n} \left| \frac{x_i^{(k+1)} - x_i^{(k)}}{x_i^{(k+1)}} \right| \le \varepsilon$$

- なお、解が収束するかどうかについても、ヤコビの反復法と同じ対角優位の条件式が使える
- ガウス・ザイデルの反復法の収束は、ヤコビの反復法よりも速いことがわかっている

Scipy, Sympyの疎行列求解関数

- Scipy

```
from scipy.sparse import csc_matrix

from scipy.sparse.linalg import spsolve

A = csc_matrix([[3, 2, 0], [1, -1, 0], [0, 5, 1]], dtype=float)

B = csc_matrix([[2, 0], [-1, 0], [2, 0]], dtype=float)

x = spsolve(A, B)
```

- Sympy

```
from sympy.matrices import SparseMatrix, Matrix

A = Matrix([1, 2, 3])

B = Matrix([2, 3, 4])

S = SparseMatrix(A.row_join(B))

x = S.solve_least_squares(Matrix([8, 14, 18]))
```

共役勾配法

- Aがn次正方行列で、$A\boldsymbol{x}=\boldsymbol{b}$のn元連立方程式を解くための関数$F(\boldsymbol{x})$をベクトルの内積を用いて、以下のように定義する

$$F(\boldsymbol{x})=\frac{1}{2}(\boldsymbol{x}\cdot A\boldsymbol{x})-(\boldsymbol{x}\cdot\boldsymbol{b})$$

- 要素で表わすと以下のようになる

$$F(\boldsymbol{x})=\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}a_{ij}x_j x_i-\sum_{i=1}^{n}x_i b_i$$

- 解の変数について、任意の初期ベクトル$\boldsymbol{x}_0$から出発し、$F(\boldsymbol{x})$が減少するようにベクトル$\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$を求め、$F(\boldsymbol{x})$を最小にする$\boldsymbol{x}$を連立一次方程式の解とする方法を共役勾配法（conjugate gradient method）と呼ぶ

共役勾配法の計算方法

- 適当な初期値ベクトル$\boldsymbol{x}_0$を設定し、$\boldsymbol{r}_0$と$\boldsymbol{p}_0$を以下のように計算する

$$\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$$
$$\boldsymbol{p}_0 = \boldsymbol{r}_0$$

- k=0,1,2,…に対して、収束するまで以下の式を繰返し計算する

$$a_k = \frac{(\boldsymbol{r}_k \cdot \boldsymbol{r}_k)}{(\boldsymbol{p}_k \cdot A\boldsymbol{p}_k)}$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + a_k \boldsymbol{p}_k$$
$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - a_k A\boldsymbol{p}_k$$

- $\|\boldsymbol{r}_{k+1}\| \leq \varepsilon$ならば終了（εは許容誤差）
  そうでないときは以下を計算

$$\beta_k = \frac{(\boldsymbol{r}_{k+1} \cdot \boldsymbol{r}_{k+1})}{(\boldsymbol{r}_k \cdot \boldsymbol{r}_k)}$$
$$\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta_k \boldsymbol{p}_k$$

Scipyの共役勾配法を使った求解関数

- **from** scipy.sparse **import** linalg
- x = linalg.cg(A, b, x0=None, tol=1e-05)　あるいは
- x = linalg.cgs(A, b, x0=None, tol=1e-05)
  - cgs関数では、Aは正方行列である必要性がある
  - x0は、xベクトルの初期値
  - tolは許容誤差
- 使用例：
  - **from** scipy.sparse **import** linalg
  - x, _ = linalg.cg(A, b, x, tol=1e-14)

ICCG法

- 不完全コレスキー分解（incomplete Cholesky decomposition）で、前処理を行ない、この結果に共役勾配法を適用する方法をICCG法（Incomplete Cholesky Conjugate Gradient method）と呼ぶ
- 係数行列Aに対して、$A=LDL^{T}$を不完全コレスキー分解または修正コレスキー分解で求めておき、その逆行列$(LDL^{T})^{-1}$を使って共役勾配法で解を求めていく

ICCG法の計算方法

- 係数行列Aに対して、$A=LDL^{\mathrm{T}}$を不完全コレスキー分解または修正コレスキー分解で求めているものとする
- 適当な初期値ベクトル$\boldsymbol{x}_0$を設定し、$\boldsymbol{r}_0$と$\boldsymbol{p}_0$を以下のように計算する

$$\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$$
$$\boldsymbol{p}_0 = (LDL^T)^{-1}\boldsymbol{r}_0$$

- k=0,1,2,…に対して、収束するまで以下の式を繰返し計算する

$$a_k = \frac{(\boldsymbol{r}_k \cdot (LDL^T)^{-1}\boldsymbol{r}_k)}{(\boldsymbol{p}_k \cdot A\boldsymbol{p}_k)}$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + a_k\boldsymbol{p}_k$$
$$\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - a_k A\boldsymbol{p}_k$$

- $\|\boldsymbol{r}_{k+1}\| \le \varepsilon$ならば終了（εは許容誤差）
  そうでないときは以下を計算

$$\beta_k = \frac{(\boldsymbol{r}_{k+1} \cdot (LDL^T)^{-1}\boldsymbol{r}_{k+1})}{(\boldsymbol{r}_k \cdot (LDL^T)^{-1}\boldsymbol{r}_k)}$$
$$\boldsymbol{p}_{k+1} = (LDL^T)^{-1}\boldsymbol{r}_{k+1} + \beta_k\boldsymbol{p}_k$$

固有値と固有ベクトル

- 有限次元線形空間 V 上の線形変換 A に対して、次の方程式を満たす零でないベクトル$\boldsymbol{x}$とスカラーλが存在するとき、$\boldsymbol{x}$をAの固有ベクトル、λをAの固有値と呼ぶ。

$$A\boldsymbol{x} = \lambda\boldsymbol{x}$$

- 固有値は、行列式に関する次の方程式を解くことによって求められる。

$$\det\left(\lambda I - A\right) = 0$$

- 但しI は単位行列である。この方程式のことを固有方程式（または特性方程式）という。
- 特性方程式で複数求まる固有値$\lambda_j (j=1,2,...,n)$に対して、各x_iを求めると、λ_jに対応した固有ベクトル$\boldsymbol{x}_j$が求められることになる

固有値の求め方

- べき乗法（power method）
  - 最大固有値を求めることができる
- ヤコビ法（Jacobi eigenvalue algorithm）・ハウスホルダー（Householder）法
  - 対称行列の固有値を求めることができる
- QR法
  - 対称でない行列の固有値も求めることができる

scipy, sympyで固有値・固有ベクトル

- scipy.linalgパッケージ
  - w, vl, vr = linalg.eig(A) … Aの固有値配列、左固有ベクトル、右固有ベクトルを返す
  - w = linalg.eigvals(A) …Aの固有値配列を返す
- sympy.matricesパッケージのMatrixクラス
  - edict = A.eigenvals( )…Aの固有値を辞書で返す（キーは固有値、値は多重度)
  - elist = A.eigenvals(multiple=True) …Aの固有値をリストで返す
  - evlist = A.eigenvectors( ) … Aの(固有値, 多重度, 固有ベクトル)のタプルを要素に持つリストを返す

べき乗法

- べき乗法（power method）は、絶対値が最大である固有値を1つ求める方法になっている
- $A\boldsymbol{x}^{(0)} \neq 0$であるような$\boldsymbol{x}^{(0)}$を選んで、以下のように繰返しAと乗算を行なう

$$\boldsymbol{x}^{(1)} = A\boldsymbol{x}^{(0)}, \boldsymbol{x}^{(1)} = A\boldsymbol{x}^{(0)}, \cdots, \boldsymbol{x}^{(k+1)} = A\boldsymbol{x}^{(k)}, \cdots$$

- $\boldsymbol{x}^{(k)}$の成分を$x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}$とすると、それらの成分比はAの最大固有値に属する固有ベクトルに収束する

$$x_1^{(k)} : x_2^{(k)} : \cdots : x_n^{(k)}$$

- この固有ベクトルから、対応する絶対値最大固有値を求めるには、レイレイ商（Rayleigh quotient）と呼ばれるベクトルの内積の商が用いられる

$$\lambda = \frac{(\boldsymbol{x}^{(k)} \cdot A\boldsymbol{x}^{(k)})}{(\boldsymbol{x}^{(k)} \cdot \boldsymbol{x}^{(k)})} = \frac{(\boldsymbol{x}^{(k)} \cdot \boldsymbol{x}^{(k+1)})}{(\boldsymbol{x}^{(k)} \cdot \boldsymbol{x}^{(k)})} = \frac{\sum_{i=1}^{n} x_i^{(k)} x_i^{(k+1)}}{\sum_{i=1}^{n} \{x_i^{(k)}\}^2}$$

- あるいは、$\boldsymbol{x}^{(k)}$と$\boldsymbol{x}^{(k+1)}$の対応する各要素の成分比を計算し、$r_1, r_2, \ldots, r_n$のうちの１つまたは、平均値を固有値とする方法もある

$$r_1 = \frac{x_1^{(k+1)}}{x_1^{(k)}}, r_2 = \frac{x_2^{(k+1)}}{x_2^{(k)}}, \cdots, r_n = \frac{x_n^{(k+1)}}{x_n^{(k)}}$$

ヤコビ法

- ヤコビ法（Jacobi eigenvalue algorithm）は、固有値を求める方法で、ヤコビの反復法（Jacobi method）とは、別のものである。
- ヤコビ法は、実対称行列Aのn個の固有値と固有ベクトルを直交行列（orthogonal matrix）Pを用いて、求める方法である。

ハウスホルダー法

- ハウスホルダー法（Householder method）は、実対称行列Aを三角対称行列（tridiagonal matrix）に変換して、固有値と固有ベクトルを求める方法である。

QR分解

- A = QR
- Qは、直交行列（直交行列とは、転置行列と逆行列が等しくなる正方行列のこと）
- Rは、上三角行列

- scipy.linalgにおいて、Q, R = qr(A)
- sympy.matricesにおいて、Q, R, p = QRdecomposition(A)

QR分解の仕方

- QR分解を計算する方法として、以下の3つが良く知られている
  - グラム・シュミットの正規直交化法を用いるもの
  - ハウスホルダー変換を用いるもの
  - ギブンス回転を用いるもの
- グラム・シュミットの正規直交化法を利用したQR分解

QR法

- QR法は、正方行列AをQR分解して、QとRの乗算順序を交替して、新たなAを求めることを繰り返して、Aを上三角行列または、ブロック三角行列に収束させて、固有値を求める方法である
- $A_1=A$から始める、k=1,2,…について、毎回QR分解を行ない、次のA_{k+1}をRQの行列の積で求めていく

$$A_k = Q_k R_k \quad A_{k+1} = R_k Q_k$$

- A_kは、$k\to\infty$のときに、右上三角行列に収束し、その対角要素には絶対値が大きい順にAの固有値が並ぶ

$$A_\infty = \begin{bmatrix} \lambda_1 & & & \\ 0 & \lambda_2 & & \\ \vdots & \cdots & \ddots & \\ 0 & 0 & \cdots & \lambda_n \end{bmatrix}$$

- 固有ベクトルを求めるためには、それぞれの固有値から特性方程式に戻って、Aとλから求める

参考・引用文献

- よくわかる数値計算　アルゴリズムと誤差解析の実際、戸川隼人他監修、佐藤次男、中村理一郎著、日刊工業新聞社、2001年
- 数値計算の基礎と応用［新訂版］数値解析学への入門、杉浦洋、サイエンス社、2009年
- Numpy linear algorithm reference, https://numpy.org/doc/stable/reference/routines.linalg.html
- Scipy linear algorithm reference, https://docs.scipy.org/doc/scipy/reference/linalg.html
- Scipy sparse linear algorithm, https://docs.scipy.org/doc/scipy/reference/sparse.linalg.html
- Sympy Matrices reference, https://docs.sympy.org/latest/modules/matrices/matrices.html
- Sympy Sparse Matrices reference, https://docs.sympy.org/latest/modules/matrices/sparse.html