lu_0

10 марта 2023 г.

1 Семинар. LU-разложение

[1]:
```
%matplotlib inline
%matplotlib notebook
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

Определение. LU-разложением квадратной матрицы A называется представление матрицы A в виде произведения

$$A = LU\,,$$

где L – нижнетреугольная матрица, U – верхнетреугольная матрица.

1.1 Применение LU-разложения матрицы

LU-разложение матрицы может быть полезно для решения матричных СЛАУ:

$$AX = B \Leftrightarrow Ax_\ell = b_\ell\,, \quad \ell = 1, 2, \ldots$$

где b_1, b_2, $\ldots$ – разные матрицы столбцы, а матрица A не зависит от ℓ.

Поскольку $A = LU$, то решение исходной задачи сводится к последовательному решению следующих СЛАУ с треугольными матрицами

$$Ly_\ell = b_\ell\,, \quad Ux_\ell = y_\ell\,.$$

Отметим, что трудоёмкость решения СЛАУ с треугольной матрицей составляет $O(n^2)$, в то время как трудоёмкость решения исходной задачи может достигать $O(n^3)$.

1.2 Метод Гаусса как метод LU-разложения матрицы

Идея исключения Гаусса – это преобразование данной системы $Ax = b$ в эквивалентную треугольную систему. Преобразование достигается составлением соответствующих линейных комбинаций уравнений.

1.2.1 Пример

$$\begin{cases} 3x_1 + 5x_2 &= 9 \\ 6x_1 + 7x_2 &= 4 \end{cases}$$

умножая первую строку на 2 и вычитая ее из второй, мы получим:

$$\begin{cases} 3x_1 + 5x_2 &= 9 \\ -3x_2 &= -14 \end{cases}$$

Это и есть исключение Гаусса при $n = 2$. Наша цель в данном разделе – дать полное описание этой важной процедуры, причем описать ее выполнение на языке матричных разложений. Данный пример показывает, что алгоритм вычисляет нижнюю унитреугольную матрицу L и верхнюю треугольную матрицу U так, что $A = LU$, т.е.

$$\begin{bmatrix} 3 & 5 \\ 6 & 7 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3 & 5 \\ 0 & -3 \end{bmatrix}.$$

Решение для исходной задачи $Ax = b$ находится посредством последовательного решения двух треугольных систем:

$$Ly = b\,, \quad Ux = y \Rightarrow Ax = LUx = Ly = b\,.$$

LU-разложение – это "высокий уровень" алгебраического описания исключения Гаусса. Представление результата матричного алгоритма на "языке" матричных разложений полезно. Оно облегчает обобщение и проясняет связь между алгоритмами, которые могут казаться очень разными на скалярном уровне.

1.3 Преобразование Гаусса

Чтобы получить разложение, описывающее исключение Гаусса, нам нужно иметь некоторое матричное описание процесса обнуления матрицы.

let $n = 2 \Rightarrow x_1 \neq 0$ и $\tau_2 = \frac{x_2}{x_1}$: $\begin{bmatrix} 1 & 0 \\ -\tau_2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \end{bmatrix}$.

В общем случае предположим, что $x \in \mathbb{R}^n$ и $x_k \neq 0$

$$\tau^T = (\tau_2, \ldots, \tau_n)\,, \quad \tau_i = \frac{x_i}{x_1}\,, \quad i = \overline{2,n}\,, \quad M = E - \tau e_1^T \Rightarrow$$

$$Mx = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -\tau_2 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ -\tau_n & 0 & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Матрица M – это матрица преобразования Гаусса. Она является нижней унитреугольной. Компоненты $\tau[k+1:n]$ – это множители Гаусса. Вектор τ называется вектором Гаусса.

1.3.1 Алгоритм

Если $x \in \mathbb{R}^n$ и элемент x_1 ненулевой, функция вычисляет вектор t длины $n-1$, такой, что если M-матрица преобразования Гаусса, причем $M[2:n,1] = t$ и $y = Mx$, то $y[2:n] = 0$.

[2]:
```
def gauss(x):
    n=len(x)
    t=x[1:]/x[0]
    return t
```

1.4 Применение матриц преобразования Гаусса

Умножение на матрицу преобразования Гаусса выполняется достаточно просто. Если матрица $C \in \mathbb{R}^{n\times r}$ и $M = E - \tau e_1^T \Rightarrow MC = (E - \tau e_1^T)C = C - \tau(e_1^T C)$.

1.4.1 Алгоритм

Если матрица $C \in \mathbb{R}^{n\times r}$ и M задает $n \times n$-преобразование Гаусса, причем $M[2:n,1] = -t$, тогда следующая функция заменяет C на MC.

[3]:
```
def gauss_app(C,t):
    C[1:,:]=C[1:,:]-np.outer(t,C[0,:])
```

Вычислительная сложность алгоритма $O(n \cdot r)$.

1.5 Пример

$$\text{let } A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 10 \end{bmatrix} \Rightarrow M_1 = \begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 0 & 1 \end{bmatrix} \Rightarrow M_1 A = \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & -6 & -11 \end{bmatrix} \Rightarrow M_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -2 & 1 \end{bmatrix} \Rightarrow M_2(M_1 A) = \begin{bmatrix} 1 & 4 & 7 \\ 0 & -3 & -6 \\ 0 & 0 & 1 \end{bmatrix}$$

Обобщим пример на достаточно общий случай.

let $A \in \mathbb{R}^{n\times n}$, матрицы преобразования Гаусса M_1, …, M_{n-1}, как правило, можно подобрать так, что матрица $M_{n-1} \dots M_1 A = U$ является верхней треугольной.

Элементарные преобразования строк в методе Гаусса, приводящие матрицу A к верхнетреугольной матрице U могут быть осуществлены путём умножения матрицы A слева на некоторые матрицы M_k:

$$A^{(1)} = A, \quad A^{(k+1)} = M_k A^{(k)}, \quad U = A^{(n-1)},$$

где

$$M_k = \begin{pmatrix} 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 1 & 0 & & 0 \\ 0 & & -\tau_{k-1,k} & 1 & & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -\tau_{n,k} & 0 & \cdots & 1 \end{pmatrix}, \quad \tau_{i,k} = \frac{a_{i,k}^{(k)}}{a_{k,k}^{(k)}}.$$

Перемножив обратные матрицы M_k^{-1}, получим

$$L = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ \tau_{2,1} & 1 & 0 & \cdots & 0 \\ \tau_{3,1} & \tau_{3,2} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tau_{n,1} & \tau_{n,2} & \tau_{n,3} & \cdots & 1 \end{pmatrix}.$$

1.5.1 Алгоритм

- На очередном шаге алгоритма мы имеем дело с матрицей $A^{(k-1)} = M_{k-1} \cdot \dots \cdot M_1 A$, которая с 1-го по $(k-1)$-й столбец является верхней треугольной.
- Множители Гаусса в M_k определяются по вектор $A^{(k-1)}[k+1:n,k]$.
- Особенно важно для продолжения процесса выполнение условия $a_{kk}^{(k-1)}$.

[4]:
```
def gauss_u(A):
    m,n=A.shape
    k=0
    while (A[k,k]!=0 and k<=n-1):
        t=gauss(A[k:,k])
        gauss_app(A[k:,k:],t)
        k+=1
```

С практической точки зрения существует несколько улучшений, которые могут быть реализованы в итоговом алгоритме. Во-первых, поскольку мы уже получили нули в столбцах с 1-го до $(k-1)$-го, то преобразование Гаусса нужно применять только к столбцам с k-го до n-го. На самом деле нет необходимости применять преобразование Гаусса также и к k-му столбцу, так как мы знаем результат $\Rightarrow$ эффективным способом для вызова процедуры `gauss_app` является следующий

```
gauss_app(A[k:,k+1:],t)
```

Другое существенное замечание состоит в том, что множители, задающие матрицу M_k, могут храниться в позициях, в которых получены нули, т. е. в элементах $A[k+1:n,k]$. С учетом этих изменений можно усовершенствовать алгоритм.

1.5.2 Алгоритм

Предположим, что матрица $A \in \mathbb{R}^{n\times n}$ обладает таким свойством, что подматрицы $A[1:k,1:k]$ невырождены для $k = \overline{1,n-1}$. Данный алгоритм вычисляет разложение $M_{n-1} \cdot \dots \cdot M_1 A = U$, где матрица U является верхней треугольной, а каждая матрица M_k – это матрица преобразования Гаусса. Матрица U хранится в верхнем треугольнике матрицы A. Множители, задающие матрицы M_k, запоминаются в элементах $A[k+1:n,k]$, т. е. $A[k+1:n,k] = -M_k[k+1:n,k]$, т. е. после выполнения алгоритма матрица

$$A = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ \tau_{2,1} & u_{22} & u_{23} & \cdots & u_{2n} \\ \tau_{3,1} & \tau_{3,2} & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \tau_{n,1} & \tau_{n,2} & \tau_{n,3} & \cdots & u_{nn} \end{bmatrix}.$$

[5]:
```
def gauss_lu(A):
    m,n=A.shape
    for j in range(n-1):
        t=gauss(A[j:,j])
        A[j+1:,j]=t
        gauss_app(A[j:,j+1:],t)
```

Пример Вычислить LU разложение матрицы $A = \begin{bmatrix} 8 & 12 & 3 & 4 & 7 \\ 7 & 8 & 9 & 10 & 15 \\ 1 & 3 & 4 & 5 & 16 \\ 3 & 7 & 8 & 5 & 3 \\ -3 & 2 & 1 & 2 & 8 \end{bmatrix}$.

[6]:
```
A=np.array([
    [8, 12, 3, 4, 7],
    [7, 8, 9, 10, 15],
    [1, 3, 4, 5, 16],
    [3, 7, 8, 5, 3],
    [-3, 2, 1, 2, 8]
], dtype='float64')
gauss_lu(A)

_,n=A.shape
L=np.eye(n)
U=np.zeros((n,n))

for j in range(n):
    L[j+1:,j]=A[j+1:,j]
    U[j,j:]=A[j,j:]

display(L@U)
```

```
array([[ 8., 12.,  3.,  4.,  7.],
       [ 7.,  8.,  9., 10., 15.],
       [ 1.,  3.,  4.,  5., 16.],
       [ 3.,  7.,  8.,  5.,  3.],
       [-3.,  2.,  1.,  2.,  8.]])
```

[7]:
```
A=np.array([
    [0, 1],
    [1, 1],
], dtype='float64')
gauss_lu(A)
```

```
C:\Users\vapan\AppData\Local\Temp\ipykernel_4904\2529989523.py:3:
RuntimeWarning: divide by zero encountered in divide
  t=x[1:]/x[0]
```

[8]:
```
def gauss_lu_row(A):
    _,n=A.shape
    piv=np.arange(n)
    for j in range(n-1):
        i_max=np.argmax(np.abs(A[j:,j]))
        A[[j,j+i_max],:]=A[[j+i_max,j],:]
        piv[j],piv[j+i_max]=piv[j+i_max],piv[j]
        t=gauss(A[j:,j])
```

```
        A[j+1:,j]=t
        gauss_app(A[j:,j+1:],t)
    return piv
```

[9]:
```
A=np.array([
    [0, 1],
    [1, 1],
], dtype='float64')
_,n=A.shape
piv=np.zeros(n-1, dtype='int32')
gauss_lu_row(A)
```

[9]:
```
array([1, 0])
```

[10]:
```
A=np.array([
    [8, 12, 3, 4, 7],
    [7, 8, 9, 10, 15],
    [1, 3, 4, 5, 16],
    [3, 7, 8, 5, 3],
    [-3, 2, 1, 2, 8]
], dtype='float64')
display(A)
piv=gauss_lu_row(A)
_,n=A.shape
L=np.eye(n)
U=np.zeros((n,n))

for j in range(n):
    L[j+1:,j]=A[j+1:,j]
    U[j,j:]=A[j,j:]

B=L@U
display(B,piv)

for i in range(n-1):
    B[[i,piv[i]],:]=B[[piv[i],i],:]
display(B)
```

```
array([[ 8., 12.,  3.,  4.,  7.],
       [ 7.,  8.,  9., 10., 15.],
       [ 1.,  3.,  4.,  5., 16.],
       [ 3.,  7.,  8.,  5.,  3.],
       [-3.,  2.,  1.,  2.,  8.]])
```

```
array([[ 8., 12.,  3.,  4.,  7.],
       [-3.,  2.,  1.,  2.,  8.],
       [ 7.,  8.,  9., 10., 15.],
       [ 3.,  7.,  8.,  5.,  3.],
```

```
       [ 1.,  3.,  4.,  5., 16.]])
```

```
array([0, 4, 1, 3, 2])
```

```
array([[ 8., 12.,  3.,  4.,  7.],
       [ 7.,  8.,  9., 10., 15.],
       [ 1.,  3.,  4.,  5., 16.],
       [ 3.,  7.,  8.,  5.,  3.],
       [-3.,  2.,  1.,  2.,  8.]])
```

1.6 Семинар. Метод прогонки. Методы решения СЛАУ с положительно определенной симметричной матрицей

[11]:
```
%matplotlib inline
%matplotlib notebook
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

1.6.1 Метод прогонки

- Прямой ход

$$\begin{cases} \alpha_i &= -\dfrac{b_i}{a_i\alpha_{i-1}+c_i}, \quad i=2,\ldots,(n-1), \\ \beta_i &= \dfrac{f_i-a_i\beta_{i-1}}{a_i\alpha_{i-1}+c_i}, \quad i=2,\ldots,(n-1). \end{cases}$$

  Начальные условия: $\alpha_1 = -b_1/c_1$, $\beta_1 = f_1/c_1$.
- Обратный ход

$$\begin{aligned} x_n =& \frac{f_n-a_n\beta_{n-1}}{c_n+a_n\alpha_{n-1}}, \\ x_i =& \alpha_i x_{i+1}+\beta_i, \quad i=(n-1),(n-2),\ldots,1. \end{aligned}$$

[12]:
```
def tridiagonal_method(a,b,c,f):
    n,=c.shape
    alpha=np.zeros(n-1); beta=np.zeros(n-1); x=np.zeros(n)
    alpha[0]=-b[0]/c[0]; beta[0]=f[0]/c[0]
    for i in range(1,n-1):
        d=a[i-1]*alpha[i-1]+c[i]
        alpha[i]=-b[i]/d
        beta[i]=(f[i]-a[i-1]*beta[i-1])/d
    x[-1]=(f[-1]-a[-1]*beta[-1])/(c[-1]+a[-1]*alpha[-1])
    for i in range(n-2,-1,-1):
        x[i]=alpha[i]*x[i+1]+beta[i]
    return x
```

[13]:
```
a=np.array([1, 2, -2, 4], dtype='float64')
b=np.array([2, -1, 3, 3], dtype='float64')
c=np.array([8, 7, -10, 12, 4], dtype='float64')
```

```
f=np.array([0, -2, -1, 5, 4], dtype='float64')

tridiagonal_method(a,b,c,f)
```

[13]:
```
array([ 0.06967213, -0.27868852,  0.11885246,  0.24863388,  0.75136612])
```

1.6.2 Методы решения СЛАУ с положительно определенной симметричной матрицей

Метод Холецкого Метод факторизации симметрической матрицы A

$$A = LL^T,$$

где L – нижнетреугольная.

Элементы матрицы L могут быть вычислены по формулам

$$l_{11} = \sqrt{a_{11}}$$

далее для $i = 2, \ldots, n$

$$l_{ij} = \frac{1}{l_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} \right), \quad j = 1, \ldots, i-1,$$

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}.$$

[14]:
```
def chol_dec(A):
    n,_= A.shape
    L = np.zeros((n,n))
    L[0,0] = np.sqrt(A[0,0])
    for i in range(1,n):
        for j in range(i):
            L[i,j] = (A[i,j] - np.dot(L[i,:j],L[j,:j]))/L[j,j]
        L[i,i]=np.sqrt(A[i,i] - np.dot(L[i,:i],L[i,:i]))
    return L
```

[15]:
```
A=np.array([
    [8, 12, 3, 4],
    [12, 32, 2, 1],
    [3, 2, 4, 1],
    [4, 1, 1, 5],
], dtype='float64')
display(A)
L=chol_dec(A)
display(L)
display(L@L.T)
```

```
array([[ 8., 12.,  3.,  4.],
       [12., 32.,  2.,  1.],
       [ 3.,  2.,  4.,  1.],
       [ 4.,  1.,  1.,  5.]])
```

```
array([[ 2.82842712,  0.        ,  0.        ,  0.        ],
       [ 4.24264069,  3.74165739,  0.        ,  0.        ],
       [ 1.06066017, -0.6681531 ,  1.55838744,  0.        ],
       [ 1.41421356, -1.33630621, -0.89378103,  0.64454726]])
```

```
array([[ 8., 12.,  3.,  4.],
       [12., 32.,  2.,  1.],
       [ 3.,  2.,  4.,  1.],
       [ 4.,  1.,  1.,  5.]])
```

Метод Холецкого может быть полезен для решения СЛАУ с симметричной, положительно определенной матрицей:

$$Ax = b\,, \quad A^{\mathrm{T}} = A\,, \quad A > 0 \Leftrightarrow (LL^{\mathrm{T}})x = b \Rightarrow$$

решение исходной задачи можно свести к последовательному решению следующих СЛАУ с треугольными матрицами

$$Ly = b\,, \quad L^{\mathrm{T}}x = y,.$$

1.6.3 Функционал энергии

Отметим, что решение СЛАУ

$$Ax = b\,,$$

где A – симметрическая положительно определённая матрица, можно заменить поиском наименьшего значения квадратичного функционала

$$\Phi(x) = \frac{1}{2}x^T Ax - x^T b,$$

называемого *функционалом энергии.*

Для градиента Φ имеем

$$\nabla\Phi(x) = \frac{1}{2}(A^T + A)x - b = Ax - b.$$

Поэтому стационарные точки функционала Φ совпадают с решениями системы $Ax = b$. Поскольку матрица A положительно определённая, то в стационарных точках достигается наименьшее значение.

1.6.4 Методы спуска

Будем искать наименьшее значение функционала Φ используя следующую итерационную схему

$$x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}.$$

Здесь $d^{(k)}$ определяет направление изменения x, а скаляр α_k – величину шага.

1.6.5 Определение величины шага

Определим α_k из условия обеспечения наименьшего значения для значения $\Phi(x^{(k+1)})$. Имеем

$$\Phi(x^{(k+1)}) = \frac{1}{2}(x^{(k)} + \alpha_k d^{(k)})^T A(x^{(k)} + \alpha_k d^{(k)}) - (x^{(k)} + \alpha_k d^{(k)})^T b,$$

Вычислив производную $\Phi(x^{(k+1)})$ по α_k и приравняв её к нулю

$$\begin{aligned}\frac{\partial}{\partial \alpha_k}\Phi(x^{(k+1)}) =& \frac{1}{2}{d^{(k)}}^T A(x^{(k)} + \alpha_k d^{(k)}) + \frac{1}{2}(x^{(k)} + \alpha_k d^{(k)})^T A d^{(k)} - {d^{(k)}}^T b = \\ =& \frac{1}{2}{d^{(k)}}^T A x^{(k)} + \frac{1}{2}{x^{(k)}}^T A d^{(k)} + \alpha_k {d^{(k)}}^T A d^{(k)} - {d^{(k)}}^T b = \\ =& {d^{(k)}}^T A x^{(k)} - {d^{(k)}}^T b + \alpha_k {d^{(k)}}^T A d^{(k)} = \\ =& {d^{(k)}}^T (A x^{(k)} - b) + \alpha_k {d^{(k)}}^T A d^{(k)} = 0\,,\end{aligned}$$

определим наилучшее значение α_k

$$\alpha_k = \frac{{d^{(k)}}^T (b - A x^{(k)})}{{d^{(k)}}^T A d^{(k)}} = \frac{{d^{(k)}}^T r^{(k)}}{{d^{(k)}}^T A d^{(k)}},$$

которое зависит только от выбранного направления спуска $d^{(k)}$ и вектора невязки $r^{(k)} = b - Ax^{(k)}$.

1.6.6 Сопряжённые направления

Определение. Точка $x^{(k)}$ называется оптимальной относительно направления p, если для любого α

$$\Phi(x^{(k)}) \le \Phi(x^{(k)} + \alpha p)\,.$$

Для того, чтобы точка $x^{(k)}$ была оптимальной относительно направления $p \Leftrightarrow$

$$\frac{\partial}{\partial \alpha}\Phi(x^{(k)} + \alpha p)\Big|_{\alpha=0} = 0\,.$$

Отсюда

$$\frac{\partial}{\partial \alpha}\Phi(x^{(k)} + \alpha p) = p^T(Ax^{(k)} - b) + \alpha p^T A p = 0\,,$$

имеем $p^T(Ax^{(k)} - b) = 0$ или $p^T r^{(k)} = 0$.

Естественнен вопрос о существовании направлений оптимальных на всех итерациях.

$$let \quad x^{(k+1)} = x^{(k)} + q.$$

Предположим точка $x^{(k)}$ оптимальна по отношению к направлению p, т. е. $p^T r^{(k)} = 0$.

Потребуем, чтобы точка $x^{(k+1)}$ также была оптимальна по отношению к направлению p, т.е. $p^T r^{(k+1)} = 0$. Имеем

$$\begin{aligned}0 =& p^T r^{(k+1)} = p^T(b - Ax^{(k+1)}) = p^T(b - A(x^{(k)} + q)) = \\ =& p^T(b - Ax^{(k+1)} - Aq) = p^T r^{(k)} - p^T A q = -p^T A q\,,\end{aligned}$$

т.е. p и q должны быть *A-ортогональны* или *A-сопряжены*.

Положим $p^{(0)} = r^{(0)}$ и будем искать направления в виде

$$p^{(k+1)} = r^{(k+1)} - \beta_k p^{(k)}, \quad k = 0, 1, \ldots$$

так, чтобы

$$p^{(j)^T} A p^{(k+1)} = 0, \quad j = 0, 1, \ldots, k.$$

Указанное требование при $j = k$ удовлетворяется для

$$\beta_k = \frac{p^{(k)^T} A r^{(k+1)}}{p^{(k)^T} A p^{(k)}}, \quad k = 0, 1, \ldots$$

Нетрудно проверить, что указанное требование будет выполнено и при $j = 0, 1, \ldots, k-1$.

1.6.7 Метод сопряжённых градиентов

Инициализация

$$r^{(0)} = b - Ax^{(0)}, \quad p^{(0)} = r^{(0)},$$

итерационная схема для $k = 0, 1, \ldots$

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}, \quad \alpha_k = \frac{p^{(k)^T} r^{(k)}}{p^{(k)^T} A p^{(k)}},$$
$$r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}, \quad \beta_k = \frac{p^{(k)^T} A r^{(k+1)}}{p^{(k)^T} A p^{(k)}}, \quad p^{(k+1)} = r^{(k+1)} - \beta_k p^{(k)}.$$

[16]:
```
def conj_grad(A,x,b,tol,N):
    r = b - np.matmul(A,x)
    p = r.copy()
    for i in range(N):
        Ap = A.dot(p)
        alpha = np.dot(p,r)/np.dot(p,Ap)
        x = x + alpha*p
        r = b - A.dot(x)
        if np.sqrt(np.sum((r**2))) < tol:
            print('Itr:', i)
            break
        else:
            beta = -np.dot(r,Ap)/np.dot(p,Ap)
            p = r + beta*p
    return x
```

[17]:
```
import numpy as np
uadd = np.frompyfunc(lambda x, y: x + y, 2, 1)
uadd.accumulate([1,2,3], dtype=object).astype(int)
# array([1, 3, 6])
```

[17]:
```
array([1, 3, 6])
```