# 05_Numerical_Linear_Algebra

February 11, 2020

[1]:
```
from IPython.display import HTML

import warnings
warnings.filterwarnings("ignore")


HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
The raw code for this Jupyter notebook is by default hidden for easier reading.
To toggle on/off the raw code, click <a href="javascript:code_toggle()">here</
 ↪a>.
<style>
.output_png {
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
</style>
''')
```

[1]:
```
<IPython.core.display.HTML object>
```


title

Module 06: Numerical Linear Algebra

This chapter discusses how to solve linear systems of equations numerically, and also introduce some basic parallel computing algorithms using OpenMP. We start with simple matrix multiplication, and then introduce Gauss elimination, which is similar to how human being solving linear equations on the paper. We will also discuss several variants of this method (Doolittle, Cholesky, Gauss-Jordan). Then we will introduce numeric iteration methods to address the problem. These methods are less intuitive, but more computational effective and much more frequently used nowadays.

## 0.1 Matrix Multiplication and its Parallel Computing Algorithm

Matrix multiplication serves as the fundamental of linear algebra. The numerical method to calculate a matrix multiplication is fairly straightforward:

$$A = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ . & . & . & . \\ a_{m1} & a_{m2} & ... & a_{mn} \end{bmatrix}, \quad B = \begin{bmatrix} b_{11} & b_{12} & ... & b_{1p} \\ b_{21} & b_{22} & ... & b_{2p} \\ . & . & . & . \\ b_{n1} & b_{n2} & ... & b_{np} \end{bmatrix}, \quad C = AB = \begin{bmatrix} c_{11} & c_{12} & ... & c_{1p} \\ c_{21} & c_{22} & ... & c_{2p} \\ . & . & . & . \\ c_{m1} & c_{m2} & ... & c_{mp} \end{bmatrix},$$

where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Matrix multiplication has a computational complexity of $O(n^3)$. Because each $c_{ij}$ can be calculated independently, matrix multiplication can be highly parallelized in the OpenMP environment.

[24]:
```
from time import time
N = 200;M = 200
A = np.random.rand(M,N)
B = np.random.rand(N,M)
C = np.random.rand(M,M)
tic = time()
for i in range(M):
    for j in range(M):
#         direct implement
```

```
#         for k in range(N):
#             C[i,j] += A[i,k]*B[k,j]
#         vectorization
        C[i,j] = np.sum(A[i,:]*B[:,j])
toc = time()
print(toc-tic)
```

```
0.2509636878967285
```

[26]:
```
N = 1000; M=500;
A = np.random.rand(M,N)
B = np.random.rand(N,M)
tic = time()
C = A@B  # A.dot(B)
toc = time()
print(toc-tic)
```

```
0.017945528030395508
```

void matrix$_m ul_p arallel(intm, intn, intp, floatA[m][n], floatB[n][p], floatC[m][p], intthread_num)omp_set_num_threads($

## 0.2 Strassen Algorithm

$$C = AB$$

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}, C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

The naive algorithm would be:

$$C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$$

, which still need 8 multiplications to calculate the $C_{i,j}$.

The Strassen algorithm defines instead new matrices:

$$M_1 := (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2})M_2 := (A_{2,1} + A_{2,2})B_{1,1}M_3 := A_{1,1}(B_{1,2} - B_{2,2})M_4 := A_{2,2}(B_{2,1} - B_{1,1})M_5 := (A_{1,1} -$$

in this case, only 7 multiplications are used instead of 8. We may now express the $C$ in terms of $M$:

$$C_{1,1} = M_1 + M_4 - M_5 + M_7C_{1,2} = M_3 + M_5C_{2,1} = M_2 + M_4C_{2,2} = M_1 - M_2 + M_3 + M_6$$

The complexity of Strassen algorithm is $\approx O(N^{2.8})$. The fastest known algorithm is called Coppersmith-Winograd algorithm, which has a complexity of $O(n^{2.375477})$.

## 0.3 Gauss Elimination

Gauss elimination and back substitution are common methods to solve linear equation set using matrix method. A **linear system of $n$ equations** in $n$ unknowns $x_1, ..., x_n$ is a set of equations of the form

$$a_{11}x_1 + ...a_{1n}x_n = b_1 a_{21}x_1 + ...a_{2n}x_n = b_2 .....a_{n1}x_1 + ...a_{nn}x_n = b_n$$

where the **coefficient** $a_{jk}$ and the $b_j$ are given numbers. This equation set can be written in matrix form as

$$Ax = b$$

where the **coefficient matrix** $A = [a_{jk}]$ is the $n \times n$ matrix, $x$ and $b$ are $n \times 1$ vectors.

$$A = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ . & . & . & . \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

We then form an **argmented matrix** of the system:

$$\tilde{A} = [A\ b] = \begin{bmatrix} a_{11} & ... & a_{1n} & b_1 \\ a_{21} & ... & a_{2n} & b_2 \\ . & . & . & . \\ a_{n1} & ... & a_{nn} & b_n \end{bmatrix}.$$

There are **three elementary row operations** we can perform on $\tilde{A}$ that will not change the solution of the linear equation system: 1. Interchange of two rows 2. Addition of a constant multiple of one row to another row 3. Multiplication of a row by a nonzero constant $c$

These three operations correspond to the following 1. Interchange of two equations 2. Addition of a constant multiple of one equation to another euqation 3. Multiplication of an equation by a nonzero constant $c$

Gauss Elimination try to reduce the coefficient matrix $A$ from a full matrix to a **triangular** matrix through a combination of these three elementary row operations, which can then be easily solved by **back substitution**. For instance, a triangular system is

$$\begin{aligned} \hat{a}_{11}x_1 + \hat{a}_{12}x_2 + \hat{a}_{13}x_3 + ... + \hat{a}_{1n}x_n &= \hat{b}_1 \\ \hat{a}_{22}x_2 + \hat{a}_{23}x_3 + ... + \hat{a}_{2n}x_n &= \hat{b}_2 \\ \hat{a}_{33}x_3 + ... + \hat{a}_{3n}x_n &= \hat{b}_3 \\ ..... & \\ \hat{a}_{n-1n-1}x_{n-1} + \hat{a}_{n-1n}x_n &= \hat{b}_{n-1} \\ \hat{a}_{nn}x_n &= \hat{b}_n \end{aligned}$$

Then from the last equation, we can solve the entire system step by step by **back substitution**:

$$\begin{aligned} x_n &= \frac{\hat{b}_n}{\hat{a}_{nn}} \\ x_{n-1} &= \frac{1}{\hat{a}_{n-1n-1}}(\hat{b}_{n-1} - \hat{a}_{n-1n}x_n) \\ &..... \\ x_1 &= \frac{1}{\hat{a}_{11}}(\hat{b}_1 - \hat{a}_{12}x_2 - \hat{a}_{13}x_3 - ... - \hat{a}_{1n}x_n) \end{aligned}$$

To transform coefficient matrix $A$ into a triangular form, we first eliminate coefficient of $x_1$ from row 2 to $n$ of $\tilde{A}$ by subtracting suitable multiples of row 1 from the other rows. In this step, the first row is called the **pivot equation** and $a_{11}$ is called the **pivot**. We changes all the rows of $\tilde{A}$ except for the pivot row. We then eliminate coefficient of $x_2$ from row 3 to $n$ by subtracting suitable multiples of row 2 from all the rows below, and so on.

$$\tilde{A}_0 = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} & b_1 \\ a_{21} & a_{22} & ... & a_{2n} & b_2 \\ . & . & . & . & . \\ a_{n1} & a_{n2} & ... & a_{nn} & b_n \end{bmatrix}.$$

$$\tilde{A}_1 = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} & b_1 \\ a_{21} - a_{11}\frac{a_{21}}{a_{11}} & a_{22} - a_{12}\frac{a_{21}}{a_{11}} & ... & a_{2n} - a_{1n}\frac{a_{21}}{a_{11}} & b_2 - b_1\frac{a_{21}}{a_{11}} \\ . & . & . & . & . \\ a_{n1} - a_{11}\frac{a_{n1}}{a_{11}} & a_{n2} - a_{12}\frac{a_{n1}}{a_{11}} & ... & a_{nn} - a_{1n}\frac{a_{n1}}{a_{11}} & b_n - b_1\frac{a_{n1}}{a_{11}} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} & b_1 \\ 0 & a_{22}^{(1)} & ... & a_{2n}^{(1)} & b_2^{(1)} \\ . & . & . & . & . \\ 0 & a_{n2}^{(1)} & ... & a_{nn}^{(1)} & b_n^{(1)} \end{bmatrix}.$$

$$\vdots$$

$$\vdots$$

$$\tilde{A}_n = \begin{bmatrix} a_{11} & a_{12} & a_{13} & ... & a_{1n} & b_1 \\ 0 & a_{22}^{(1)} & a_{23}^{(1)} & ... & a_{2n}^{(1)} & b_2^{(1)} \\ 0 & 0 & a_{33}^{(2)} & ... & a_{3n}^{(2)} & b_2^{(2)} \\ . & . & . & . & . & . \\ 0 & 0 & 0 & ... & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{bmatrix}.$$

One practical concern about Gauss Elimination is that the pivot $a_{kk}$ **must be** different from zero and **should be** large in absolute value to avoid numeric instability in the elimination. For this reason in each step we choose our pivot row the one with the absolutely largest $a_{jk}$ (with $j > k$) and interchange it with current row $k$. This method is called **partial pivoting** and is quite popular (e.g., Maple).

[3]:
```
def Gauss_elimination(A,b,print_process=False):
    ''' Function that utilizes Gauss Elimination and back substitution to solve
    a linear system Ax = b
    usage: x = Gauss_elimination(A,b,print_process=False):
    input:
        A: nxn coefficient matrix, full rank
        b: nx1 vector
        print_process: boolean, whether to print the step-by-step process
                       for Gauss Elimination
    output:
        solution x
    written by Ge Jin, gjin@mines.edu, 06/2019
    '''
    n = A.shape[0]
    # generate argmented matrix
    Ab = np.hstack((A,b.reshape(-1,1)))
```

```
    # Gauss Elimination
    for k in range(n-1):
        # find the row with max ajk
        maxi = np.argmax(np.abs(Ab[k:,k]))
        maxi += k
        # swap the rows
        Ab[[k,maxi]] = Ab[[maxi,k]]
        # eliminate ajk from row k+1 to n
        for j in range(k+1,n):
            Ab[j,:] = Ab[j,:] - Ab[k,:]*Ab[j,k]/Ab[k,k]
        if print_process:
            print('Ab_{}='.format(k))
            print(Ab)

    # back substitution
    x = np.zeros(n)
    x[n-1] = Ab[n-1,n]/Ab[n-1,n-1]
    for k in range(len(x)-2,-1,-1):
        x[k] = 1/Ab[k,k]*(Ab[k,n]-np.sum(Ab[k,:-1]*x))
    return x
```

[4]:
```
from time import time
tic = time()
N = 1000
x = Gauss_elimination(np.random.rand(N,N)+1,np.random.
 ↪rand(N),print_process=False)
toc =time()
toc-tic
```

[4]:
```
10.986541509628296
```

[5]:
```
# an example
A = np.array([[6.,2,8],[3,5,2],[0,8,2]])
b = np.array([26.,8.,-7.])
x = Gauss_elimination(A,b,print_process=True)
print('x=',x)
```

```
Ab_0=
[[ 6.  2.  8. 26.]
 [ 0.  4. -2. -5.]
 [ 0.  8.  2. -7.]]
Ab_1=
[[ 6.   2.   8.  26. ]
 [ 0.   8.   2.  -7. ]
 [ 0.   0.  -3.  -1.5]]
x= [ 4.  -1.   0.5]
```

[6]:
```
# an example
A = np.array([[3.,6,3],[1,1,1],[2,1,1]])
b = np.array([12.,3.,4.])
x = Gauss_elimination(A,b,print_process=True)
print('x=',x)
```

```
Ab_0=
[[ 3.  6.  3. 12.]
 [ 0. -1.  0. -1.]
 [ 0. -3. -1. -4.]]
Ab_1=
[[ 3.          6.          3.          12.        ]
 [ 0.         -3.         -1.         -4.        ]
 [ 0.          0.          0.33333333  0.33333333]]
x= [1. 1. 1.]
```

### 0.3.1 Computation cost of Gauss Elimination

Generally, the important factors in judging the quality of a numeric method are - Amount of storage - Amount of time (number of operations) - Effect of roundoff error (numerical stability)

For Gauss elimination, the operation count for a full matrix is as follows. In Step $k$ we eliminate $x_k$ from $n-k$ equations. This needs $n-k$ divisions in computing the $a_{jk}/a_{kk}$, and $(n-k)(n-k+1)$ multiplications and as many subtractions. Since we do $n-1$ steps, $k$ goes from 1 to $n-1$ and thus the total number of operations in this forward elimination is

$$f(n) = \sum_{k=1}^{n-1}(n-k) + 2\sum_{k=1}^{n-1}(n-k)(n-k+1) = \frac{1}{2}(n-1)n + \frac{2}{3}(n^2-1)n \approx \frac{2}{3}n^3$$

where $2n^3/3$ is obtained by dropping lower powers of $n$ (when $n$ is large, the only term that matters is the one with highest power of $n$). Because $f(n)$ grows about proportional to $n^3$, we say that $f(n)$ is order $n^3$ and write

$$f(n) = O(n^3)$$

In the back substituion of $x_i$ we make $n-i$ multiplications and as many subtractions, as well as 1 division. Hence the number of operations in the back substitution is

$$b(n) = 2\sum_{i=1}^{n}(n-i) + n = n(n+1) + n = N^2 + 2n = O(n^2)$$

For instance, if an operation takes $10^{-9}$ second, then the times needed are:

| Algorithm | $n = 1000$ | $n = 10000$ |
|---|---|---|
| Elimination | 0.7 sec | 660 sec |
| Back substitution | 0.001 sec | 0.1 sec |

## 0.4 LU-Factorization

Another way to solve linear systems $ Ax=b$ numerically if through **LU-Factorization**. An LU-factorization of a given square matrix $A$ is of the form

$$A = LU$$

where $L$ is **lower triangular** matrix and $U$ is **upper triangular** matrix.

It can be proved that for any nonsingular matrix, the rows can be reordered so that the resulting matrix A has an LU-factorization in which $L$ turns out to be the matrix of the multipliers $m_{jk}$ of the Gauss elimination, with main diagonals equal 1, and $U$ is the matrix of the triangular system at the end of the Gauss elimination.

$$A = LU = \begin{bmatrix} 1 & 0 & 0 & ... & 0 \\ m_{21} & 1 & 0 & ... & 0 \\ m_{31} & m_{32} & 1 & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ m_{n1} & m_{n2} & m_{n3} & ... & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & ... & u_{1n} \\ 0 & u_{22} & u_{23} & ... & u_{2n} \\ 0 & 0 & u_{33} & ... & u_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & ... & u_{nn} \end{bmatrix}$$

And $L$ and $U$ can be calculated using **Doolittle method**:

$$\begin{aligned} u_{1i} &= a_{1i} && i = 1, ..., n \\ m_{j1} &= \frac{a_{j1}}{u_{11}} && j = 2, ..., n \\ u_{jk} &= a_{jk} - \sum_{s=1}^{j-1} m_{js} u_{sk} && k = j, ..., n; \;\; j \geq 2 \\ m_{lj} &= \frac{1}{u_{jj}} \left( a_{lj} - \sum_{s=1}^{j-1} m_{ls} u_{sj} \right) && l = j+1, ..., n; \;\; k \geq 2 \end{aligned}$$

The crucial idea is that $L$ and $U$ can be computed directly, without solving simultaneous equations (thus, without suing the Gauss elimiantion). Once we have $L$ and $U$, we can use it for solving $Ax = b$ in two steps, involving only about $n^2$ operations.

Noting that $Ax = LUx = b$ may be written as

$$Ly = b \quad where \quad Ux = y$$

We can use forward substitution to solve $y$, and then use backward substitution to solve $x$.

A similar method, **Crout's method**, is obtained if $U$ (instead of $L$) is required to have main diagonals equal 1. In either case the factorization is unique.

### 0.4.1 Exercise

1. Solve the LU factorization of a general $3 \times 3$ matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

### 0.4.2 Cholesky's Method

If matrix $A$ is symmetric and positive definite ($A = A^T, x^T Ax > 0$ for all $x \neq 0$), we can even choose $U = L^T$ to further reduce the computational cost. In this case, we cannot impose conditions on the main diagonal entries.

$$A = LL^T = \begin{bmatrix} l_{11} & 0 & 0 & ... & 0 \\ l_{21} & l_{22} & 0 & ... & 0 \\ l_{31} & l_{32} & l_{33} & ... & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & ... & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} & ... & l_{n1} \\ 0 & l_{22} & l_{32} & ... & l_{n2} \\ 0 & 0 & l_{33} & ... & l_{n3} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & ... & l_{nn} \end{bmatrix}$$

The formulas to calculate $l_{jk}$ are:

$$\begin{aligned} l_{11} &= \sqrt{a_{11}} \\ l_{j1} &= \frac{a_{j1}}{l_{11}} && j = 2, ..., n \\ l_{jj} &= \sqrt{a_{jj} - \sum_{s=1}^{j-1} l_{js}^2} && j = 2, ..., n \\ l_{pk} &= \frac{1}{l_{jj}} \left( a_{pj} - \sum_{s=1}^{j-1} l_{js} l_{ps} \right) && p = j+1, ..., n; \;\; j \geq 2 \end{aligned}$$

## 0.5 Gauss-Jordan Elimination: Matrix Inversion

Sometimes it is useful to get the inversion of the matrix $A$, and **Gauss-Jordan elimination** is one of the methods calculate it. The inversion of a matrix is defined as

$$AA^{-1} = A^{-1}A = I$$

To get $A^{-1}$, similar to Gauss elimination, we can form the **argmented matrix**:

$$\tilde{A} = [A \; I] = \begin{bmatrix} a_{11} & ... & a_{1n} & 1 & 0 & 0 & ... & 0 \\ a_{21} & ... & a_{2n} & 0 & 1 & 0 & ... & 0 \\ . & . & . & . & . & . & . & . \\ a_{n1} & ... & a_{nn} & 0 & 0 & 0 & ... & 1 \end{bmatrix}.$$

We then apply the Gauss elimination to $\tilde{A}$, which gives a matrix of the form $[U \; H]$ with $U$ being an upper triangular matrix. The Gauss-Jordan method reduces U by further elementary row operations to diagonal form $[I \; K]$, where $K$ is the inversion of $A$.

[7]:
```
def gauss_jordan_matrix_inv(A,print_process=False):
    ''' Function that utilizes Gauss-Jordan Elimination
    to calculate the inversion of the input matrix
    usage: x = gauss_jordan_matrix_inv(A,print_process=False):
    input:
        A: nxn coefficient matrix, full rank
        print_process: boolean, whether to print the step-by-step process
    output:
```

```
        inversion of A
    written by Ge Jin, gjin@mines.edu, 06/2019
    '''
    n = A.shape[0]
    step = 0
    # generate argmented matrix
    Ab = np.hstack((A,np.identity(n)))
    # Gauss Elimination
    for k in range(n-1):
        # find the row with max ajk
        maxi = np.argmax(np.abs(Ab[k:,k]))
        maxi += k
        # swap the rows
        Ab[[k,maxi]] = Ab[[maxi,k]]
        # eliminate ajk from row k+1 to n
        for j in range(k+1,n):
            Ab[j,:] = Ab[j,:] - Ab[k,:]*Ab[j,k]/Ab[k,k]
        if print_process:
            print('Ab_{}='.format(step))
            print(Ab)
            step += 1
    # change diagonal elements to 1
    for k in range(n):
        Ab[k] /= Ab[k,k]
        if print_process:
            print('Ab_{}='.format(step))
            print(Ab)
            step += 1
    # Gauss-Jordan elimination
    for k in range(n-1,0,-1):
        for i in range(0,k):
            Ab[i] -= Ab[k]*Ab[i,k]
            if print_process:
                print('Ab_{}='.format(step))
                print(Ab)
                step += 1
    invA = Ab[:,n:]
    return invA
```

[8]:
```
A = np.array([[-1,1,2],[3,-1,1],[-1,3,4]])
invA = gauss_jordan_matrix_inv(A,print_process=True)
print('Inverted Matrix:')
print(invA)
```

```
Ab_0=
[[ 3.         -1.          1.          0.          1.          0.        ]
 [ 0.          0.66666667  2.33333333  1.          0.33333333  0.        ]
 [ 0.          2.66666667  4.33333333  0.          0.33333333  1.        ]]
```

```
Ab_1=
[[ 3.         -1.          1.          0.          1.          0.        ]
 [ 0.          2.66666667  4.33333333  0.          0.33333333  1.        ]
 [ 0.          0.          1.25        1.          0.25       -0.25      ]]
Ab_2=
[[ 1.         -0.33333333  0.33333333  0.          0.33333333  0.        ]
 [ 0.          2.66666667  4.33333333  0.          0.33333333  1.        ]
 [ 0.          0.          1.25        1.          0.25       -0.25      ]]
Ab_3=
[[ 1.         -0.33333333  0.33333333  0.          0.33333333  0.        ]
 [ 0.          1.          1.625       0.          0.125       0.375     ]
 [ 0.          0.          1.25        1.          0.25       -0.25      ]]
Ab_4=
[[ 1.         -0.33333333  0.33333333  0.          0.33333333  0.        ]
 [ 0.          1.          1.625       0.          0.125       0.375     ]
 [ 0.          0.          1.          0.8         0.2        -0.2       ]]
Ab_5=
[[ 1.         -0.33333333  0.         -0.26666667  0.26666667  0.06666667]
 [ 0.          1.          1.625       0.          0.125       0.375     ]
 [ 0.          0.          1.          0.8         0.2        -0.2       ]]
Ab_6=
[[ 1.         -0.33333333  0.         -0.26666667  0.26666667  0.06666667]
 [ 0.          1.          0.         -1.3        -0.2         0.7       ]
 [ 0.          0.          1.          0.8         0.2        -0.2       ]]
Ab_7=
[[ 1.   0.   0.  -0.7  0.2  0.3]
 [ 0.   1.   0.  -1.3 -0.2  0.7]
 [ 0.   0.   1.   0.8  0.2 -0.2]]
Inverted Matrix:
[[-0.7  0.2  0.3]
 [-1.3 -0.2  0.7]
 [ 0.8  0.2 -0.2]]
```

## 0.6 Linear Systems: Solution by Iteration

The Gauss elimination and its variants we just discussed belong to the **direct methods** for solving linear systems of equations. These methods give solutions after an amount of computation that can specified in advance. In contrast, in an **indirect** or **iterative method**, we start from an approximation to the true solution and, if successful, obtain better and better approximations from a computational cycle repeated as often as may be necessary for achieving a required accuracy. Iterative methods in general have less computational cost comparing to direct methods, especially for very large matrix or very sparse ones.

### 0.6.1 Gauss-Seidel Iteration Method

This is an iterative method of great practical importance for solving linear system $Ax = b$. To obtain the algorithm, let us derive the general formulas for this iteration.

We can rearrange the equations so that $a_{jj} > 0$ for $j = 1, ..., n$. We then decompose $A$ into a lower triangular component $L_*$, and a strictly upper triangular component $U$

$$A = L_* + U \ \ where \ \ L_* = \begin{bmatrix} a_{11} & 0 & ... & 0 \\ a_{21} & a_{22} & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & ... & a_{nn} \end{bmatrix}, \ \ U = \begin{bmatrix} 0 & a_{12} & ... & a_{1n} \\ 0 & 0 & ... & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 0 \end{bmatrix},$$

By substituting it into $Ax = b$, we have

$$Ax = (L_* + U)x = b.$$

$$L_* x = b - Ux$$

The Gauss-Seidel method now solves the left hand side of this expression for $x$, using previous value for $x$ on the right hand side. Analytically, this may be written as:

$$L_* x^{(k+1)} = b - Ux^{(k)}.$$

By taking advantage of the triangular form of $L_*$, the elements of $x^{(k+1)}$ can be computed sequentially using forward substitution:

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

**Jacobi Method for Parallel Computing** It can be observed that Gaussian-Seidel iteration method is very difficult to parallelize, because of the input dependency at each step. Jacobi method is a slightly altered version of Gaussian-Seidel method, which is more parallel friendly.

For Jacobi method, each iteration can be presented as:

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

For Jacobi method, the new iteration $x^{(k+1)}$ only depends on the value of current iteration $x^{(k)}$, thus each element of $x^{(k+1)}$ can be calculated at different nodes to accelerate the computation.

**Convergence**

One of the common problem for all iteration-based methods is that the iteration may not always converge to the true answer. For Gauss-Seidel method, to get the convergence condition, we have

$$x^{(k+1)} = -L_*^{-1} U x^{(k)} + L_*^{-1} b = C x^{(k)} + L_*^{-1} b$$

The Gauss-Seidel iteration converges for every $x^{(0)}$ if and only if all the eigenvalues of $C$ have absolute value less than 1. However, calculating eigenvalues of $C$ is a more computational expensive operation than the iteration itself. We usually estimate the convergence using **Sufficient Convergence Condition**:

$$\|C\| < 1.$$

Here $\$\|C\|\$ is the **matrix norm**. There are several frequently used ones.

**Frobenius norm**: root square of all the element summations.

$$\|C\| = \sqrt{\sum_{j=1}^{n}\sum_{k=1}^{n} c_{jk}^2}$$

**Column "sum" norm**: the greatest of the sums of the elements in a column of $C$.

$$\|C\| = \max_k \sum_{j=1}^{n} |c_{jk}|$$

**Row "sum" norm**: the greatest of the sums of the elements in a row of $C$.

$$\|C\| = \max_j \sum_{k=1}^{n} |c_{jk}|$$

[9]:
```
def Gauss_Seidel_iteration(A,b,x0,max_iter=100,err_tol =␣
↪1e-3,print_process=False):
    ''' Function that utilizes Gauss Seidel Iteration to solve
    a linear system Ax = b
    usage: x = Gauss_Seidel_iteration(A,b,x0,max_iter=100,err_tol =␣
↪1e-3,print_process=False)
    input:
        A: nxn coefficient matrix, full rank
        b: nx1 vector
        x0: initial gauss of solution
        max_iter: maximum number of iterations
        err_tol: error tolorate of the solution
        print_process: boolean, whether to print the step-by-step process
    output:
        solution x
    written by Ge Jin, gjin@mines.edu, 06/2019
    '''
    n = A.shape[0]
    x = x0.copy()
    iter_success = False
    for niter in range(max_iter):
        if print_process:
            print(x)
        old_x = x.copy()
        for i in range(n):
            x[i] = 1/A[i,i]*(b[i]-np.sum(A[i,:]*x)+A[i,i]*x[i])
        max_diff = np.max(np.abs(x-old_x))
        if max_diff < err_tol:
            iter_success = True
            break
    if not iter_success:
        print('Max iteration number reached! Solution may not be accurate!')
```

```
    return x
```

[10]:
```
# demo of Gauss-Seidel Iteration
A = np.array([[1,-0.25,-0.25,0],[-0.25,1,0,-0.25],[-0.25,0,1,-0.25],[0,-0.25,-0.
↪25,1]])
b = np.array([50,50,25,25])
x0 = np.ones(4)*100
Gauss_Seidel_iteration(A,b,x0,print_process=True)
```

```
[100. 100. 100. 100.]
[100.   100.    75.    68.75]
[93.75   90.625  65.625  64.0625]
[89.0625   88.28125  63.28125  62.890625]
[87.890625   87.6953125  62.6953125  62.59765625]
[87.59765625 87.54882812 62.54882812 62.52441406]
[87.52441406 87.51220703 62.51220703 62.50610352]
[87.50610352 87.50305176 62.50305176 62.50152588]
[87.50152588 87.50076294 62.50076294 62.50038147]
[87.50038147 87.50019073 62.50019073 62.50009537]
```

[10]:
```
array([87.50009537, 87.50004768, 62.50004768, 62.50002384])
```

[11]:
```
# A comparison between Gauss Elimination and Gauss-Seidel Iteration
N = 100
A = np.random.rand(N,N) + 100*np.identity(N)
b = np.random.rand(N)*100
x0 = np.random.rand(N)

%timeit Gauss_Seidel_iteration(A,b,x0)
%timeit Gauss_elimination(A,b)

x_gs = Gauss_Seidel_iteration(A,b,x0)
x_ge = Gauss_elimination(A,b)
print('Maximum Error of iteration solution: ',np.max(np.abs(x_gs - x_ge)))
```

```
7.97 ms ś 187 ţs per loop (mean ś std. dev. of 7 runs, 100 loops each)
65.9 ms ś 1.83 ms per loop (mean ś std. dev. of 7 runs, 10 loops each)
Maximum Error of iteration solution:  1.9403455026323346e-05
```

## 0.7 Norms and Condition Number

A computational problem is called **ill-conditioned** (or ill-posed) if "small" changes in the data (the input) cause "large" changes in the solution (the output). In contrast, a problem is called **well-conditioned** (or well-posed) if "small" changes in the data cause only "small" changes in the solution. Keep in mind these concepts are qualitative. The definition of "small" and "large" depends on the accuracy of the data and the error tolerance of the solution.

Let's take a simple linear system as an example. Say we would like to calculate the crossing point location of two straight lines. If the two lines are nearly parallel to each other, this problem is ill-conditioned. Because a small change in the intercept can change the location of the crossing point dramatically, as shown in the figure.

Here is an example of ill-conditioned linear system:

$$0.9999x_1 + 1.0001x_2 = 1x_1 - x_2 = 1$$

By inducing a small error $\epsilon$ to the intercept on the second equation, we have

$$0.9999x_1 + 1.0001x_2 = 1x_1 - x_2 = 1 + \epsilon.$$

This system has the solution $x_1 = 0.5 + 5000.5\epsilon$, $x_2 = -0.5 + 4999.5\epsilon$.

Our goal is to show that ill-conditioning of a linear system and of its coefficient matrix $A$ can be measured by a number, the **condition number** $\kappa(A)$. Other measures for ill-conditioning have also been proposed, but $\kappa(A)$ is probably the most widely used one. $\kappa(A)$ is defined in terms of norm. To reach our goal, we discuss in three steps: 1. **Vector norms** 2. **Matrix norms** 3. **Condition number**

### 0.7.1 Vector Norms

A **vector norm** for column vectors $x = [x_j]$ with $n$ components is a generalized length of the vector. It is denoted by $\|x\|$ and is defined by four properties, 1. $\|x\|$ is a nonnegative real number. 2. $\|x\| = 0$ if and only if $x = 0$. 3. $\|kx\| = |k|\|x\|$ for all $k$. 4. $\|x + y\| \leq \|x\| + \|y\|$

The most important norms in connection with computations is the **p-norm** defined by

$$\|x\|_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

where $p$ is a fixed number. The most commonly used ones are

$$\begin{aligned} \|x\|_1 &= |x_1| + |x_2| + ... + |x_n| && (l_1\text{-norm}) \\ \|x\|_2 &= \sqrt{|x_1|^2 + |x_2|^2 + ... + |x_n|^2} && (\text{Euclidean or } l_2\text{-norm}) \\ \|x\|_\infty &= max_j|x_j| && (l_\infty\text{-norm}) \end{aligned}$$

**Exercise:** What is the norms of vector $x^T = [2 \quad -3 \quad 0 \quad 1 \quad -4]$?

### 0.7.2 Matrix Norm

If $A$ is an $n \times n$ matrix and $x$ any vector with $n$ components, then $Ax$ is a vector with $n$ components. We now take a vector norm and consider $\|x\|$ and $\$ \|Ax\|\$.$*Onecanprovethatthereisanumberc* (depending on $A$) such that

$$\|Ax\| \leq c\|x\|, \quad \text{for all } x$$

The **matrix norm of** $A$ is defined as the smallest possible $c$ valid for all $x$ ($\neq 0$). Thus

$$\|A\| = \min(c) = \max \frac{\|Ax\|}{\|x\|}$$

or

$$\|A\| = \max_{\|x\|=1} \|Ax\|$$

Note carefully that $\|A\|$ depends on the vector norm that we selected. In particular, one can show that + for the $l_1$-norm one gets the column sum norm of $A$. + for the $l_\infty$-norm one gets the row sum norm of $A$.

More importantly, by the norm definition, we have

$$\|Ax\| \leq \|A\|\|x\|$$

### 0.7.3 Condition Number of a Matrix

We are now ready to introduce the key concept in our discussion of ill-conditioning, the **condition number** $\kappa(A)$ of a (nonsingular) square matrix $A$, defined by

$$\kappa(A) = \|A\|\|A^{-1}\|$$

For a linear system of equations $Ax = b$, if the **condition number of $A$ is small, then the problem is well-conditioned.**

To prove this, let's assume the observation $b$ has a small error $\epsilon$, which causes an inaccurate solution $\tilde{x}$.

$$A\tilde{x} = b + \epsilon$$

By substituting $Ax = b$, we have

$$\epsilon = A(x - \tilde{x})$$
$$x - \tilde{x} = A^{-1}\epsilon$$

Taking the norm on both side yields

$$\|x - \tilde{x}\| \leq \|A^{-1}\epsilon\| \leq \|A^{-1}\|\|\epsilon\|$$

Division by $\$\|x\|\$onbothsidefinallygives\frac{\|x-\tilde{x}\|}{\|x\|} \leq \frac{1}{\|x\|}\|A^{-1}\|\|\epsilon\|$Because $b = Ax$, $\|b\| = \|Ax\| \leq \|A\|\|x\|$, we have

$$\frac{1}{\|x\|} \leq \frac{\|A\|}{\|b\|}$$

Thus,

$$\frac{\|x - \tilde{x}\|}{\|x\|} \leq \frac{\|A\|}{\|b\|}\|A^{-1}\|\|\epsilon\| = \kappa(A)\frac{\|\epsilon\|}{\|b\|}$$

Hence if $\kappa(A)$ is small, a small error $\|\epsilon\|$ implies a small relative error $\$ \{\|x - \tilde{x}\|\}/\{\|x\|\}\$, so that the system is well-conditioned.

**Exercise**

1. Calculate the condition number of matrix

$$A = \begin{bmatrix} 5 & 1 & 1 \\ 1 & 4 & 2 \\ 1 & 2 & 4 \end{bmatrix}, \qquad \text{which has the inverse} \qquad A^{-1} = \frac{1}{56}\begin{bmatrix} 12 & -2 & -2 \\ -2 & 19 & -9 \\ -2 & -9 & 19 \end{bmatrix}.$$

2. Calculate the condition number of matrix

$$A = \begin{bmatrix} 0.9999 & -1.0001 \\ 1.0000 & -1.0000 \end{bmatrix}, \qquad \text{which has the inverse} \qquad A^{-1} = \begin{bmatrix} -5000 & 5000.5 \\ -5000 & 4999.5 \end{bmatrix}.$$

**Further comments on Condition Numbers**

1. There is no sharp dividing line between "well-conditioned" and "ill-conditioned", but generally the situation will get worse as we go from systems with small $\kappa(A)$ to systems with larger $\kappa(A)$.
2. Ill-conditioned matrix takes longer to converge using Gauss-Seidel iteration.
3. With modern computer science development, numerical algorithms can handle ill-conditioned matrix much better than they used to.

# 1 Acknowledgement

Most of this teaching material is based on:
Kreyszig, E., 2018. Advanced Engineering Mathematics, 10-th edition.

[ ]: