Topic 1: Computer Arithmetic and Algorithms

Numeral Systems:

Decimal Numeral System:

- **Base**: 10 (Use subscript 10 to represent base 10, 1_{10})
- **Digits**: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Form**: $d_m d_{m-1} \cdots d_0 . d_{-1} d_{-2} \cdots d_{-n}$
  - **Computation:** $d_m \times 10^m + d_{m-1} \times 10^{m-1} + \cdots + d_0 \times 10^0 + d_{-1} \times 10^{-1} + \cdots + d_{-n} \times 10^{-n}$
- **Integer Part**: Before the '.' separator
- **Fractional Part**: After the '.' separator

Binary Numeral System:

- **Base**: 2 (Use subscript 2 to represent base 2, 1_2)
- **Digits**: 0, 1
- **Form**: $(b_m b_{m-1} \cdots b_0 . b_{-1} b_{-2} \cdots b_{-n})_2$
  - **Computation:** $b_m \times 2^m + b_{m-1} \times 2^{m-1} + \cdots + b_0 \times 2^0 + b_{-1} \times 2^{-1} + \cdots + b_{-n} \times 2^{-n}$
- **Integer Part**: Before the '.' separator
- **Fractional Part**: After the '.' Separator

Note: The base is always a decimal number and any number without a subscript is a decimal number

Conversion:

Binary to Decimal:
Multiply each of the digits with the respective powers of 2

Suppose that we have a binary number $b_m b_{m-1} \cdots b_0 . b_{-1} b_{-2} \cdots b_{-n}$

$$(b_m b_{m-1} \cdots b_0 . b_{-1} b_{-2} \cdots b_{-n})_2 = (b_m \times 2^m + b_{m-1} \times 2^{m-1} + \cdots + b_0 \times 2^0 + b_{-1} \times 2^{-1} + \cdots + b_{-n} \times 2^{-n})_{10}$$

Decimal to Binary:
Suppose that we have a decimal number $d_m d_{m-1} \cdots d_0 . d_{-1} d_{-2} \cdots d_{-n}$
Just have to combine the integer and fractional part together after conversion $(b_m b_{m-1} \cdots b_0 . b_{-1} b_{-2} \cdots b_{-n})_2$

Integer Part:

1) Divide $d_m d_{m-1} \cdots d_0$ by 2
2) Resultant Remainder is b_0 (1st position of integer part of binary)
3) Divide the quotient by 2
4) Resultant Remainder is b_1 (2nd position of integer part of binary)
5) Repeat until the quotient is 0
6) We will have the sequence of $(b_m b_{m-1} \cdots b_0)_2$

Fractional Part:

1) Multiply $0. d_{-1} d_{-2} \cdots d_{-n}$ by 2
2) Resultant Integer part is b_{-1} (1st position of frac part of binary)
3) Multiply the fractional part by 2
4) Resultant Integer part is b_{-2} (2nd position of frac part of binary)
5) Repeat until the result is 0
6) We will have the sequence of $(0. b_{-1} b_{-2} \cdots b_{-n})_2$

Floating Point Formats:

Normalized Form:
For any nonzero binary number, we can write it in the normalized form (scientific notation)

$$Binary\ Number: (b_m b_{m-1} \cdots b_0 . b_{-1} b_{-2} \cdots b_{-n})_2$$
$$Normalized\ Form: \pm (1. s_1 s_2 \cdots s_N)_2 \times 2^k$$

Where $s_i \in \{0, 1\}$ and $k - Exponent$

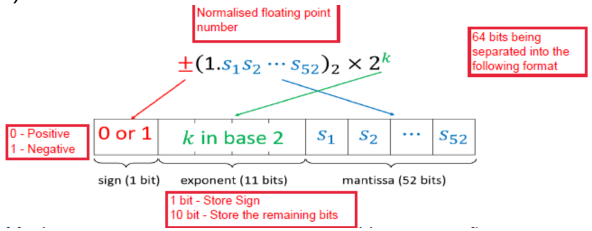
Conversion: Just move the separator forward or backwards until we have $(1. s_1 \cdots s_N)_2$ and depending on the direction and number of times, it will be the power of the exponent. (Similar to converting decimal numbers just that it will be converted to powers of 2 for each shift)

Pieces of Information to be stored by computer for numbers:

1) Digits $s_1, s_2, \cdots, s_N \in \{0, 1\}$
2) Exponent $k \in \mathbb{Z}$
3) Sign (Positive / Negative)

IEEE Standard:

1) **Single Precision**: 32 Bits (Less Memory Storage, Less Precise)
2) **Double Precision**: 64 Bits
   a. Used by Python by default
3) **Long Double Precision**: 80 Bits (More Memory Store, More Precise)



Normalised floating point number
64 bits being separated into the following format
$\pm(1.s_1 s_2 \cdots s_{52})_2 \times 2^k$
0 - Positive
1 - Negative
0 or 1
k in base 2
s_1
s_2
$\cdots$
s_{52}
sign (1 bit)
exponent (11 bits)
mantissa (52 bits)
1 bit - Store Sign
10 bit - Store the remaining bits

Figure 1 Machine Representation of Normalized double precision floating point numbers

	Number of Bits		
Precision	Sign	Exponent	Mantissa (N)
Single	1	8	23
Double	1	11	52
Long Double	1	15	64

Figure 2 Bit Distribution

Important Numbers:
Note that $(10)_2$ is 2 in base 2 and not 10, therefore the exponents are all in terms of power 2.

- Smallest positive normalized double precision floating-point number: $+(1.00 \ldots 00 \times 10^{-1111111110})_2 = 2^{-1022} \approx 2.22 \times 10^{-308}$
- Largest positive normalized double precision floating-point number: $+(1.11 \ldots 11 \times 10^{+1111111111})_2 = (2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}$

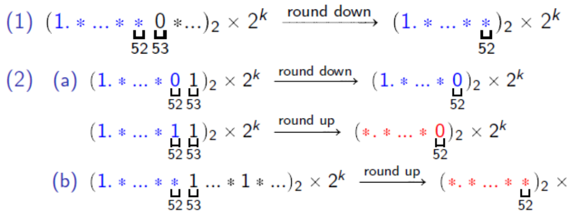
Machine Epsilon:
Distance between 1 and the smallest floating point number greater than 1

$$\epsilon_{mach} = 2^{-52}$$

This is for IEEE double precision floating point standard

IEEE Rounding to Nearest Rule:
Note that the following is for double precision

1) If 53rd bit to the right is 0, then round down (cut off after the 52nd bit)
2) If the 53rd bit to the right is 1:
   a. If all the known bits to the right of the 53rd bit are 0 (i.e. no more bit after 53rd bit):
      i. Round down if 52nd is 0
      ii. Round up if 52nd is 1
   b. Round up if there is at least one more bit that is 1 on the right of 53rd bit



(1)
$(1. * \ldots * \underset{52}{*}\ \underset{53}{0} * \ldots)_2 \times 2^k \xrightarrow{\text{round down}} (1. * \ldots * \underset{52}{*})_2 \times 2^k$
(2)
(a)
$(1. * \ldots * \underset{52}{0}\ \underset{53}{1})_2 \times 2^k \xrightarrow{\text{round down}} (1. * \ldots * \underset{52}{0})_2 \times 2^k$
$(1. * \ldots * \underset{52}{1}\ \underset{53}{1})_2 \times 2^k \xrightarrow{\text{round up}} (*. * \ldots * \underset{52}{0})_2 \times 2^k$
(b)
$(1. * \ldots * \underset{52}{*}\ \underset{53}{1} \ldots * 1 * \ldots)_2 \times 2^k \xrightarrow{\text{round up}} (*. * \ldots * \underset{52}{*})_2 \times 2^k$

Figure 3 Visual Representation of Rounding Rule

Subnormal Floating Point Number:
Used to represent numbers represent numbers smaller than 2^{-1022}

$$\pm(0. s_1 s_2 \cdots s_{52})_2 \times 2^{-1022}$$

Smallest representable double precision number $= 2^{-52} \times 2^{-1022} = 2^{-1074}$

Note that these numbers are still machine representable just that adding 1 to them will not make a difference since it will get rounded off and becomes 0

Rounding rule still applies the same way for subnormal, we just need to make the exponent of 2 to be 2^{-1022} and then carry out the rounding rule the same way

Computer Arithmetic:

Numbers store by computer: $fl(x)$ (After rounding off)

Operations:
Note that there will be 2 rounds of rounding off, we will round off the number first before the operation and round off again after the operation

1) $x \oplus y = fl(fl(x) + fl(y))$
2) $x \ominus y = fl(fl(x) - fl(y))$
3) $x \otimes y = fl(fl(x) \times fl(y))$
4) $x \oslash y = fl(fl(x) \div fl(y))$

Note: The operations can be carried out the normal way and note that it is in base 2 so if we add, 1+1 then it will be 0 carry forward 1

Topic 2: Computer Arithmetic, Computational Error and Algorithms

Dangerous Operations for Computer Arithmetic

Most of the issues occur due to rounding error or the representability of the number

1) Adding up 2 numbers with significantly different magnitudes

$$2^{-1} \oplus 2^{52} = 2^{52}$$

- *Issue*: Smaller number will get rounded off due to the rounding rule

2) Taking the difference of 2 nearly equal numbers

$$(2^{51} + 2^{-3}) \ominus 2^{51} = 0$$

- *Issue*: The smaller number will get rounded off first before we carry out the difference. Therefore, the result is not 2^{-3}. It will be better if we take the difference between those with similar magnitudes first

3) Product of 2 small numbers

$$2^{-520} \otimes 2^{-560} = 0$$

- *Issue*: $2^{-520} \otimes 2^{-560}$ is smaller than the smallest representable number

Computational Error:

Definition 3.2:
Suppose p^* is an approximate of p

Absolute Error of Approximation:

$$|p^* - p|$$

- Can vary a lot as it considers the absolute terms of the magnitude of p

Relative Error:

$$\frac{|p^* - p|}{|p|}$$

provided $p \neq 0$

- Tells us the significant level with respect to the actual number p of the error of approximation
- This is relatively more stable as it considers in terms of the value p rather than the absolute terms of the magnitude of p

Relative Rounding Error:
This is for an IEEE machine arithmetic model

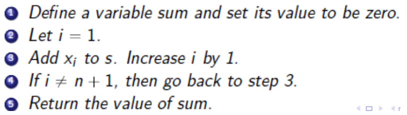
$$\frac{|fl(p) - p|}{|p|} \leq \frac{1}{2}\epsilon_{mach}$$

$\epsilon_{mach} = 2^{-52}$ – Machine Epsilon
p – Number that we are representing
$fl(p)$ – Floating point representation

Terms:

Algorithms:

- Set of unambiguous instructions to solve a class of problems or perform a computation

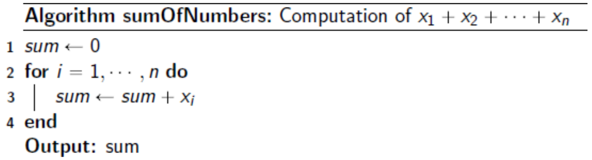


1. *Define a variable sum and set its value to be zero.*
2. *Let $i = 1$.*
3. *Add x_i to s. Increase i by 1.*
4. *If $i \neq n + 1$, then go back to step 3.*
5. *Return the value of sum.*

Figure 4 Example of Algorithm

Pseudocode:

- Informal description of normal programming language, that is intended to be read by human rather than machine



```
Algorithm sumOfNumbers: Computation of $x_1 + x_2 + \cdots + x_n$
sum ← 0
for $i = 1, \cdots, n$ do
    sum ← sum + $x_i$
end
Output: sum
```

Figure 5 Example of Pseudocode

Computational Cost:

Focus:

- Number of floating-point operations since they are the slowest part of the algorithm in most computers

Note:

- We can regard every *for* loop as a summation sign since we are repeating the same operation for the number of times the for loop operates

Big-O Notation:

Definition 3.4:

- We say $f(n) = O(g(n))$ if there exists a positive constant M and a real number N such that:
$$|f(n)| \leq Mg(n), \quad \forall n > N$$
In other words, so long as for a value of n that is large, if we can find a function $Mg(n)$ to bound it, it will be $O(g(n))$ complexity

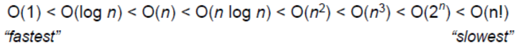
To find the Big-O:

- Compute the complexity, find the most dominant term and take away the coefficient

Computing Time Complexity:

- Find out the number of floating-point operations that occurs
- Find the big O → Find the most dominant term and take away the coefficient

Analysis of Time Complexities:



O(1) < O(log n) < O(n) < O(n log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)
"fastest" "slowest"

Figure 6 Common Growth Terms

- Note that the higher the power, the slower the time complexity because the time required will increase much faster and therefore it will be a slower time complexity

Estimation of Time Taken:

Suppose that we have a simulation for a size m and we know the computational time. If we know the time complexity, we can estimate the computational time for all different sizes

$$T(n) = f\left(\frac{n}{m}\right) \times t_m$$

$T(n)$ – Estimated time taken for size n
$f(x)$ – Time complexity for the operation (Function of the Big-O)
n – Size that we want to estimate
m – Original size that we took the computation
t_m – Time required for the original size m

Useful Formulas for Computation of Time Complexities:

Arithmetic Series:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

If $a_n = a_{n-1} + c$, where c is a constant, then

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n = \frac{n(a_n + a_1)}{2}$$

Geometric Series:

$$\sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$$

If $a_n = ca_{n-1}$, where $c \neq 1$ is a constant, then

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n = a_1\frac{c^n - 1}{c - 1}$$

If 0 < c < 1, then the sum of the infinite geometric series is

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1 - c}$$

Harmonic Series:

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx \ln(i + 1)$$

Sum of Squares:

$$\sum_{i=1}^{n} i^2 = 1 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

Topic 3: Matrix Multiplication

Matrix Multiplication:
Given $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times p}$

$$C = AB \text{ where } C \in \mathbb{R}^{m\times p}$$

Representation:

$$A = \left(a_{ij}\right)_{m\times n} \in \mathbb{R}^{m\times n}$$
$$B = \left(b_{ij}\right)_{n\times p} \in \mathbb{R}^{n\times p}$$
$$C = \left(c_{ij}\right)_{m\times p} \in \mathbb{R}^{m\times p}$$

i – Row of the matrix
j – Column of the Matrix
$x \times y$ – Dimensions of the matrix (Rows × Columns)

Entries of C (Matrix after Multiplication):

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$$
$$= \sum_{k=1}^{n} a_{ik}b_{kj}\ , \qquad \forall i = 1, \cdots m, \qquad j = 1, \cdots, p$$

- We will take the cartesian product of the ith row of A and the jth row of B

Implementation of Matrix Multiplication:

Matrix Multiplication for $A = \left(a_{ij}\right)_{m\times n}, B = \left(b_{ij}\right)_{n\times p},\ C = AB$

Pseudocode:

```
1    set C = $\left(c_{ij}\right)_{m\times p}$ to be a zero matrix
2    for $i = 1, \cdots, m$ do
3        for $j = 1, \cdots, p$ do
4            $c_{ij} \leftarrow a_{i1}b_{1j}$
5            for $k = 2, \cdots, n$ do
6                $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
7            end
8        end
9    end
```

Time Complexity: $O(mnp)$

- We have 3 for loops and we can convert them into summations and we will get the time complexity to be $O(mnp)$
- Note that line 4 is an optimization step, even though it does not change the time complexity, it still improves the computational cost since for each p there will be 1 less computation to compute

Note:

- For line 4, we do this assignment first so that it reduces an iteration of line 6 where we have 0 + $a_{i1}b_{1j}$
- Line 2: Loop through all the rows of A (m rows)
- Line 3: Loop through all the columns of B (p columns)
- Line 5: Loop through all the elements of the current row and column of B (n elements each)

Special Matrices:

Triangular Matrices:
Suppose A and B are lower-triangular matrices:

$$a_{ij} = b_{ij} = 0, \qquad if\ i < j$$
$$A = \begin{bmatrix} a_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}, \qquad B = \begin{bmatrix} b_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ b_{n1} & \cdots & b_{nn} \end{bmatrix}$$

- If $row\ index < column\ index$, the entry will be 0
- $C = AB$ is also a lower-triangular matrix

Optimization:
Since C is lower triangular:

1) $c_{ij} = 0,\ \forall i < j$ (Do not need to compute this) Since it is already 0
2) For $i \geq j$: (**Only need to compute this case**)
   a. A is lower triangular: $a_{ik} = 0,\ \forall i < k$
   b. B is lower triangular: $b_{kj} = 0,\ \forall k < j$
   c. $a_{ik}b_{kj} = 0$ if $i < k\ or\ k < j$
      i. The resultant computation will be 0 if either value is 0, a_{ik} or b_{kj}. a_{ik} will be 0 when the current column that we are iterating is 0. b_{kj} will be 0 when the current row that we are iterating is 0

Matrix Multiplication that we must do:

$$c_{ij} = \sum_{k=j}^{i} a_{ik}b_{kj}\ , \qquad 1 \leq j \leq i \leq n$$

- Note the condition on $1 \leq j \leq i \leq n$ because we do not need to consider if $i < j$ since C is lower triangular

Visualisation:
Suppose we are considering the ith row of A to be multiplied with the jth column of B for c_{ij}

$$A_{i*} = \left[\boxed{a_{i1} \quad a_{i2} \quad \cdots} \quad \boxed{a_{ik} \quad \cdots \quad a_{in}}\right]$$

$$B_{*j} = \begin{bmatrix} \boxed{\begin{matrix} b_{1j} \\ \vdots \\ b_{kj} \end{matrix}} \\ \boxed{\begin{matrix} b_{(k+1)j} \\ \vdots \\ b_{nj} \end{matrix}} \end{bmatrix}$$

If we are doing the multiplication, we will multiply the element in the row vector of A that is of the same position with that of the element in the column vector of B

Example: If $i = 5, j = 2, n = 6$

$$A_{5*} = [a_{51} \quad a_{52} \quad a_{53} \quad a_{54} \quad a_{55} \quad a_{56} = 0], B_{*2} = \begin{bmatrix} b_{12} = 0 \\ b_{22} \\ b_{32} \\ b_{42} \\ b_{52} \\ b_{62} \end{bmatrix}$$

Note that $a_{51} \times b_{12}$ & $a_{56} \times b_{62}$ will be 0. The only multiplication that does not result in 0 is $j = 2 \leq k \leq i = 5$

Pseudo Code:
Matrix Multiplication of 2 lower triangular matrices $A = \left(a_{ij}\right)_{n\times n}$ and $B = \left(b_{ij}\right)_{n\times n}$

```
1  Set $c_{ij} = 0$ for all $1 \leqslant i < j \leqslant n$
2  for $i = 1, ..., n$ do
3      for $j = 1, \cdots, i$ do
4          $c_{ij} \leftarrow a_{ij}b_{jj}$
5          for $k = j + 1, \cdots, i$ do
6              $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
7          end
8      end
9  end
   Result: $C = (c_{ij})_{n\times n}$
```

Note:

- Line 2 is to consider all i
- Line 3 considers $j \leq i$ as only those values of c_{ij} are non zero because C is lower triangular
- Line 4 is an optimization step so that there isn't an extra addition for $0 + a_{ik}b_{kj}$
- Line 5 is to loop over all values of $j \leq k \leq i$ as only those values we do not have either of the values from row of A or column of B to be 0

Time Complexity:

Addition:

$$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=j+1}^{i}1=\sum_{i=1}^{n}\sum_{j=1}^{i}(i-(j+1)+1)$$

Total no. of elements = Last Index – First index + 1

$$=\sum_{i=1}^{n}\sum_{j=1}^{i}(i-j)$$

$$=\sum_{i=1}^{n}[(i-1)+\cdots+1]$$

Substituting the values inside

$$=\sum_{i=1}^{n}\frac{i(i-1)}{2}$$

Sum of Arithmetic Series until i -1

$$=\frac{1}{2}\left(\sum_{i=1}^{n}i^2-\sum_{i=1}^{n}i\right)$$

$$=\frac{1}{2}\left(\frac{n(n+1)(2n+1)}{6}-\frac{n(n+1)}{2}\right)$$

Sum of i and i^2 formula

$$=\frac{(n-1)n(n+1)}{6}$$

Multiplication:

$$\sum_{i=1}^{n}\sum_{j=1}^{i}\left(1+\sum_{k=j+1}^{i}1\right)=\sum_{i=1}^{n}\sum_{j=1}^{i}1+\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=j+1}^{i}1$$

$$=\sum_{i=1}^{n}i+Number\ of\ Additions$$

Notice that second part is number of additions

$$=\frac{n(n+1)}{2}+\frac{(n-1)n(n+1)}{6}$$

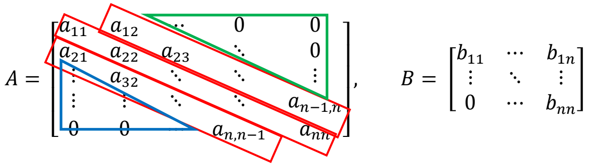
Sum of i and using the result from number of additions

$$=\frac{n(n+1)(n+2)}{6}$$

Time Complexity: $O(n^3)$ after adding the addition and multiplication

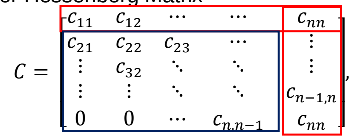
Tridiagonal Matrix:

Suppose $A=(a_{ij})_{n\times n}\in\mathbb{R}^{n\times n}$ is a tridiagonal matrix and $B=(b_{ij})_{n\times n}$ be an upper triangular matrix


$A=$
a_{11} a_{12} $\cdots$ 0 0
a_{21} a_{22} a_{23} $\ddots$ 0
$\vdots$ a_{32} $\ddots$ $\ddots$ $\vdots$
$\vdots$ $\vdots$ $\ddots$ $\ddots$ $a_{n-1,n}$
0 0 $\cdots$ $a_{n,n-1}$ a_{nn}
$B=$
b_{11} $\cdots$ b_{1n}
$\vdots$ $\ddots$ $\vdots$
0 $\cdots$ b_{nn}

- A is tridiagonal therefore we have only entries in the 3 diagonals

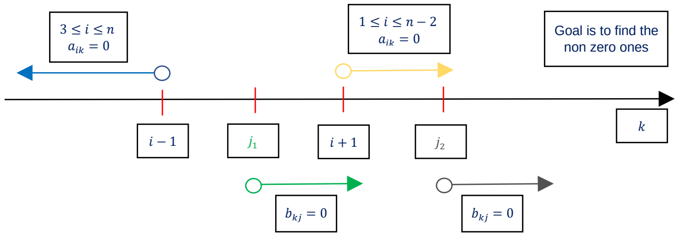
$C=AB$ is an upper Hessenberg Matrix


$C=$
c_{11} c_{12} $\cdots$ $\cdots$ c_{nn}
c_{21} c_{22} c_{23} $\cdots$ $\vdots$
$\vdots$ c_{32} $\ddots$ $\ddots$ $\vdots$
$\vdots$ $\vdots$ $\ddots$ $\ddots$ $c_{n-1,n}$
0 0 $\cdots$ $c_{n,n-1}$ c_{nn}

- C is an upper Hessenberg Matrix so the first row and last column will have entries and if we omit those 2, the remaining will be an upper triangular matrix (Blue Box)

Optimization:

- C is an upper Hessenberg: $c_{ij}=0,\ \forall j<i-1\,,3\le i\le n$ (Do not need to compute since all 0)
  - $3\le i\le n$ Because the first row will all contain non zero entries by definition and the second row will all contain non zero entries since it is the first row of an upper triangular matrix. It is only when we get to the 3rd row where it is possible to get a 0 entry
  - $j<i-1$ Because it is an upper triangular matrix, so the zero entries occur when the row index is more than the column index. However, we note that the row index starts from $i-1$ because we need to exclude the first row of non zero entries
- For $j\ge i-1$: (Need to compute)
  - A is tridiagonal:
    $a_{ik}=0,\forall k<i-1,3\le i\le n$ or $k>\ i+1,1\le i\le n-2$
    - **Case 1**: $\forall k<i-1,3\le i\le n$. This accounts for the lower triangle that contains zero in blue. We will need to skip the first 2 rows first and it is like the upper triangular computation for the zero entries. However, we will need to account for the fact that the diagonal starts only from $i-1$ as we need to exclude the first row
    - **Case 2**: $\forall k>\ i+1,1\le i\le n-2$. This accounts for the upper triangle that contains zero in green. We will need to skip the last 2 rows and it is like the lower triangular computation or zero entries. However, we need to account for the fact that the columns start only from $k-1$ and rearranging we get $k>i+1$
  - B is upper triangular: $b_{kj}=0,\forall j<k$
    - The zero entries for an upper triangular is where the $row\ index>column\ index$
  - $a_{ik}b_{kj}=0,\forall k<i-1\,,3\le i\le n\ or\ k>\min(i+1,j)\,,1\le i\le n-2$
    - Case 1: $\forall k<i-1,3\le i\le n$. We note that we do not need to consider in this case because $\forall j<i-1$, $c_{ij}=0$ as per the first point. Therefore, when $k<i-1$, $c_{ij}=0$ Since a_{ik} will be 0 as well
    - Case 2: $\forall k>\ i+1,1\le i\le n-2$ and $\forall j<k$. Note that there are 2 subcases
      - Case 2a: $j<i+1$ (j_1). For this case, when $k>j_1$, $b_{kj}=0$
      - Case 2b: $j>i+1$ (j_2). For this case, when $k>j_2, b_{kj}=0$ but we also have $k>i+1, a_{ik}=0$. So the range will be reduced to $k>i+1$
      - Therefore, we will just need the minimum of either $j\ or\ i+1$ to cover both cases $k>\min(i+1,j)$



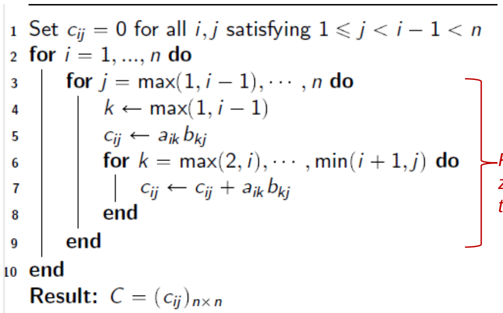
Matrix Multiplication that we must do:

$$c_{ij}=\sum_{k=\max(1,i-1)}^{\min(i+1,j)}a_{ik}b_{kj}\,,\qquad\forall i=1,\cdots,n\ ,\forall j=\max(1,i-1)\,,\cdots,n$$

- Note that $k=\max(1,i-1)$ is so that when $i=1$, we will not get $k=0$
- This is just taking the complement of the indices that will result in $a_{ik}=0$ or $b_{kj}=0$
- Note that $j\ge i-1$ but also must be more than the index 0 so we take the $\max(1,i-1)$. This is to account for $c_{ij}=0$ automatically when $j<i-1$

Pseudo Code:

Matrix Multiplication $C=AB$ for a tridiagonal matrix $A=(a_{ij})_{n\times n}$ and upper-triangular matrix $B=(b_{ij})_{n\times n}$



```
1  Set $c_{ij}=0$ for all $i,j$ satisfying $1\leqslant j<i-1<n$
2  for $i=1,\ldots,n$ do
3      for $j=\max(1,i-1),\cdots,n$ do
4          $k\leftarrow\max(1,i-1)$
5          $c_{ij}\leftarrow a_{ik}b_{kj}$
6          for $k=\max(2,i),\cdots,\min(i+1,j)$ do
7              $c_{ij}\leftarrow c_{ij}+a_{ik}b_{kj}$
8          end
9      end
10 end
   Result: $C=(c_{ij})_{n\times n}$
```

Set those that are zero first

Finding values that are non zero for i,j and considering the values k can take

Note:

- Line 2 to consider all values of i
- Line 3 to consider all values of $j\ge i-1$ for the condition $\forall j=\max(1,i-1)\,,\cdots,n$
- Line 4 we set k to the starting index
- Line 5 optimization step to reduce redundancy in the computation (Adding redundant 0)
- Line 6 loop over all $\max(1,i-1)+1=\max(2,i)\le k\le\min(i+1,j)$. Note that we add 1 to the first index because of Line 5

Time Complexity:

Addition:

$$\sum_{i=1}^{n}\sum_{j=\max(1,i-1)}^{n}\sum_{k=\max(2,i)}^{\min(i+1,j)} 1 = \sum_{i=1}^{n}\sum_{j=\max(1,i-1)}^{n}\overbrace{(\min(i+1,j)-\max(2,i)+1)}^{\boxed{\text{Last Index – First Index + 1}}}$$

$$= \underbrace{\sum_{j=1}^{n}(\min(2,j)-1)}_{} + \underbrace{\sum_{i=2}^{n}\sum_{j=i-1}^{n}(\min(i+1,j)-i+1)}_{}$$

$i\ = 1$ (Since this is the case, the first summation will only be for $i = 1$ so it disappears)
$\max(1, i-1) = 1$
$\min(i+1, j) = \min(2, j)$
$\max(2, i) = 2$
$-\max(2, i) + \ 1 = \ -1$

$i\ = 2, \cdots, n$ (First summation will start from $i\ = 2$)
$\max(1, i-1) = i-1$
$\max(2, i) = i$

$$= \overbrace{1}^{\boxed{j=1}} + \underbrace{\overbrace{2}^{\boxed{j=2}} + \overbrace{2}^{\boxed{j=3}} + \cdots + \overbrace{2}^{\boxed{j=n}}}_{\boxed{n-1\text{ terms}}} - \underbrace{n}_{\boxed{\text{From -1}}} + \sum_{i=2}^{n}(\underbrace{0}_{\boxed{j=i-1}} + \overbrace{1}^{\boxed{j=i}} + \overbrace{2}^{\boxed{j=i+1}} + \cdots + \overbrace{2}^{\boxed{j=n}})$$

$$= 1 + 2(n-1) - n + \sum_{i=2}^{n}\big(1 + 2(n-(i+1)+1)\big)$$

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

$$= n - 1 + \sum_{i=2}^{n} 1 + \sum_{i=2}^{n} 2n - \sum_{i=2}^{n} 2i \quad \boxed{\text{Splitting the summation}}$$

$$= \ n - 1 + n - 1 + 2n(n-1) - 2\left(\frac{n(n+1)}{2} - 1\right)$$

$$= n^2 - n$$

Last Index – First Index +1

Sum of i and we -1 because of the starting index $i\ = 2$ so we have to remove 2(1) = 2 which is 2(-1)

Multiplication:

$$\frac{1}{2}(3n-2)(n+1)$$

Time Complexity: $O(n^2)$ after adding the addition and multiplication

Special Matrix Summary:

Matrix Type	Entries
Full	$a_{ij} \neq 0 \qquad for\ i = 1,2,\dots,n$ $for\ j = 1,2,\dots,n$
Upper Triangular	$a_{ij} \neq 0 \qquad for\ i = 1,2,\dots,n$ $for\ j = i,(i+1),\dots,n\ (j \geq i)$
Lower Triangular	$a_{ij} \neq 0 \qquad for\ i = 1,2,\dots,n$ $for\ j = 1,2,\dots,i\ (j \leq i)$
Diagonal	$a_{ii} \neq 0 \qquad for\ i = 1,2,\dots,n,\ i = j$
Tridiagonal	$a_{ij} \neq 0 \qquad for\ i = 1,2,\dots,n$ $for\ j = \max(1, i-1),\dots,\min(\mathrm{i}+1,\mathrm{n})$ $\max(1, i-1) \leq j \leq \min(i+1, n)$
Upper Hessenberg	$a_{ij} \neq 0 \qquad for\ i = 1,2,\dots,n$ $for\ j = \max(1, i-1),\dots,n$ $j \geq \max(1, i-1)$
Lower Hessenberg	$a_{ij} \neq 0 \qquad for\ i = 1,2,\dots,n$ $for\ j = 1,\dots,\min(\mathrm{i}+1,\mathrm{n})$ $j \leq \min(i+1, n)$

Topic 4: Solving equations – Bisection & Fixed-point iteration

Intermediate Value Theorem

Theorem 5.3:
Let f be a continuous function on $[a,b]$, satisfying $f(a)f(b) < 0$. Then f has a root between a and b, that is, there exists a number r satisfying $a < r < b$ and $f(r) = 0$

Bisection Method

Algorithm:

First, find an interval $[a,b]$ satisfying IVT

Then run the following algorithm to solve for root:

```
Set $a \leftarrow a_0,\ b \leftarrow b_0$
while $(b-a)/2 > TOL$ do
    $c \leftarrow (a+b)/2$
    if $f(c) = 0$ then
        stop
    end
    if $f(a)f(c) < 0$ then
        $b \leftarrow c$
    else
        $a \leftarrow c$
    end
end
Result: The approximate root is $(a+b)/2$.
```

Terminating Condition: $\frac{b-a}{2} \leq TOL$

Finding the midpoint

Found exact root c such that $f(c) = 0$

Sign of $f(c)$ is different from $f(a)$, set b as c

Sign of $f(c)$ is different from $f(b)$, set a as c

When we terminate, we either have the exact root or a root within TOL

- Note that the updating of c is done to making the range smaller

Accuracy & Speed:

After n bisection steps:

- the interval $[a_n, b_n]$ length $\frac{b-a}{2^n}$
- midpoint $x_c = \frac{a_n+b_n}{2}$ gives best estimate of the solution r
- solution error satisfies:

This 1/2 is to check the half after the n bisection steps

Approx. - Actual

$$\boxed{|x_c - r| \leq \frac{1}{2}\left(\frac{b-a}{2^n}\right) = \frac{b-a}{2^{n+1}}}$$

- number of function evaluation is $n+2$

Convergence rate is $\frac{1}{2}$ since the interval is cut by $\frac{1}{2}$ for each step

Stopping tolerance (TOL):

A solution is correct within p decimal places if the error is less than 0.5×10^{-p} (TOL).

We can estimate the number of steps of bisection required to find a root within the *TOL*.

$$\boxed{\frac{b-a}{2^{n+1}} < \overset{TOL}{0.5 \times 10^{-p}}}$$

Given p, solve for n
$n =$ number of steps needed

Limitations:

Bisection method is prone to inaccuracy due to rounding error if the $TOL \leq 10^{-16}$ *(relative error of floating point)*

Fixed-point Theorem

Theorem 6.5:
Let g to be a continuous function in $[a,b]$ such that $g(x) \in [a,b]$, for all x in $[a,b]$. Suppose, in addition, that g' exists on (a,b) and that a constant $0 \leq k < 1$ exists with

$$|g'(x)| \leq k, \qquad \forall x \in (a,b)$$

Then for any number x_0 in $[a,b]$, the sequence defined by $x_n = g(x_{n-1})$, $n \geq 1$, converges to the unique fixed point r in $[a,b]$.

Fixed-point Iteration Method

Deriving fixed-point function:

Given $\boldsymbol{f}$, we want to find the root $\boldsymbol{f(x)}$:

$\boldsymbol{f(x) = 0}$, where $\boldsymbol{f}$ is a continuous function

Transforming the problem into a fixed-point problem:

$\boldsymbol{g(x) = x}$, for some function $\boldsymbol{g}$ derived from $\boldsymbol{f}$

Ensure that $\boldsymbol{g}$ satisfies the following conditions:

- $\boldsymbol{g}$ is continuous and $\boldsymbol{a \leq g(x) \leq b}$ for all $\boldsymbol{x \in [a,b]}$
- $\boldsymbol{|g'(x)| < 1}$ for all $\boldsymbol{x \in [a,b]}$

Methods of deriving fixed-point function:

1) $f(x) = 0 \quad \Rightarrow \quad f(x) + x = x$

2) Derive using each degree of polynomial:

eg. $f(x) = x^3 + x^2 + x + 2 = 0$

- 3rd degree: $x^3 = -(x^2 + x + 2)$
  $\Rightarrow \quad x = \sqrt[3]{-(x^2+x+2)}$
- 2nd degree: $x^2 = -(x^3 + x + 2)$
  $\Rightarrow \quad x = \sqrt[2]{-(x^3+x+2)}$

eg. $g(x) = x^3 + x + 2 = 0$

- 2nd degree: $x^2 = \frac{-(x+2)}{x}$

  $\Rightarrow \quad x = \sqrt[2]{\frac{-(x+2)}{x}}$

Algorithm:

First, find a suitable function $\boldsymbol{g(x) = x}$

Then run the following algorithm to solve for fixed-point of $\boldsymbol{g}$:

```
Set an initial guess $x_0$.
for $i = 0, 1, 2, \cdots$ do
    $x_{i+1} \leftarrow g(x_i)$
    if terminating condition holds then
        $x_{fp} \leftarrow g(x_{i+1})$
        break
    end
end
Result: The approximate root is $x_{fp}$.
```

Applying function to previous x value

Terminating Condition: $|g(x_{i+1}) - x_{i+1}| < TOL$

- Note that the quantity $|\boldsymbol{g(x_{i+1}) - x_{i+1}}|$ is also known as the **backward error**.

Definition 6.2:
Assume that f is a function and that r is a root, meaning that it satisfies $f(r) = 0$. Assume that $x_a$is an approximation to r. For the root-finding problem, the **backward error** of the approximation $x_a$is $|f(x_a)|$ and the **forward error** is $|r - x_a|$.

- **Forward Error**: Analytical analysis (When we want to create a new numerical method). When we know what is the value of r and to check how good our approximate is, we see how close it is to the actual solution r
- **Backward Error**: Implementation analysis. After the computation, we see how close our solution is to 0

Convergence & Convergence rate:

Theorem 6.4
Assume that g is a continuously differentiable, that $g(r) = r$, and that $S \coloneqq |g'(r)| < 1$. Then fixed-point iteration **converges linearly with rate** S to the fixed point r for initial guesses sufficiently close to r.

Suppose x^* is the actual solution and x_i is the i-th iterate of an iterative method

$$\boxed{\begin{gathered} e_i \coloneqq |x_i - x^*| \\ \lim_{i\to\infty} \frac{e_{i+1}}{e_i} = S < 1 \end{gathered}}$$

Given $g(r) = r$,

If $S = |g'(r)| > 1$, this means error increases and no convergence.

If $S = |g'(r)| < 1$, this means convergence and with rate S.

If $S = |g'(r)| = 0$, this is as small as S can get, leading to very fast convergence (**Fastest Possible**)

If $S = |g'(r)| = 1$, this does not necessarily mean convergence
For example:

Given 2 function with fixed point $x = 0$

$$g_1(x) = x - x^3$$
$$g_2(x) = x + x^3$$

If $0 < x_0 < 1$, the fixed-point iteration with g_1 gives the iterates $x_0 > x_1 > \cdots > 0$

⇒ g_1 converges to the fixed point $x = 0$, while $g_1'(x) = 1$.

If $0 < x_0 < 1$, the fixed-point iteration with g_2 gives the iterates $x_0 < x_1 < \cdots$

⇒ g_1 diverges from the fixed point $x = 0$, while $g_1'(x) = 1$.

Two fixed points:

Given a function g has 2 fixed points x_a & x_b,

Check the gradient at the fixed points:

- If $|g'(x_a)| < 1$ while $|g'(x_b)| > 1$

⇒ g converges to fixed-point x_a because g is convergent at fixed point x_a while it is divergent at fixed point x_b.

- If $|g'(x_a)| < |g'(x_b)| < 1$

⇒ g converges to fixed-point x_a as g is convergent at both fixed point but it converges at a faster rate for x_a as seen by $|g'(x_a)| < |g'(x_b)|$.

Function Evaluation:

For fixed-point iteration, the main computational cost of the algorithm (function evaluation) is in evaluating $g(x)$.

How to calculate computational cost of each $g(x)$?

Function g is commonly a polynomial. A polynomial of degree m has the following form:

$$P_m(x) = a_0 + a_1 x + \cdots + a_{m-1}x^{m-1} + a_m x^m$$

where $a_0, a_1, \ldots, a_{m-1}, a_m$ are the coefficients

Horner's method can be used to evaluate polynomial functions, which gives the optimal computational cost.

Horner's method:

Given x, the value of the polynomial $P_m(x)$ can be found using only m additions and m multiplications.

$$\begin{aligned} P_m(x) &= a_0 + a_1 x + \cdots + a_{m-1}x^{m-1} + a_m x^m \\ &= a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{m-1} + a_m x^m) \ldots))) \end{aligned}$$

$$\left.\begin{aligned} p_m &= a_m \\ p_{m-1} &= a_{m-1} + x \cdot p_m \\ p_{m-2} &= a_{m-2} + x \cdot p_{m-1} \\ \ldots &= \cdots \\ p_1 &= a_1 + x \cdot p_2 \\ p_0 &= a_0 + x \cdot p_1 \end{aligned}\right\}$$

The above formula written iteratively

Algorithm:

```
P ← $a_m$
for $k = m - 1, \cdots, 0$ do
    P ← $a_k + xP$
end
Result: P
```

Time complexity of Horner's method: $O(2m) \approx O(m)$

Time complexity of fixed-point iteration:
$O(mn)$, where n is the number of iterations

Summary

Bisection method: To find a root for $f(x) = 0$ in the interval $[a, b]$

- Cut the interval $[a, b]$ by $\frac{1}{2}$ for each step to find a small interval in the end that contains the root of f.
- Pro:
  - guarantee to converge linearly
- Con:
  - need to predefine an initial interval
  - convergence rate is $\frac{1}{2}$ (might be slower than fixed-point iteration)

Fixed-point iteration: To find a root for g$(x) = x$]

- Keep applying the function g to the iterates to find a solution.
- Pro:
  - when convergence rate $S < \frac{1}{2}$, it might converge much faster than bisection method
- Con:
  - might be hard to formulate g from f

Topic 5: Methods for solving linear systems – Gaussian elimination

Linear Systems

A system of n equations in unknown $x_1, x_2, \dots, x_n \in \mathbb{R}$:

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

In matrix form:

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

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

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right]$$

(Augmented matrix)

Gaussian Elimination

To solve linear systems by reducing the number of unknown variables by linear combination of the equations. Gaussian Elimination that solves the linear system includes 2 steps:

1) Elimination
2) Backwards substitution

Elimination

Steps:

To eliminate each variable, using row operations, and obtain an upper-triangular matrix by the end of the process.

$$\left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} & b_n \end{array}\right]$$

$$\xrightarrow{\substack{R_2 \leftarrow R_2 - \frac{a_{21}}{a_{11}}R_1 \\ R_3 \leftarrow R_3 - \frac{a_{31}}{a_{11}}R_1 \\ \vdots \\ R_n \leftarrow R_n - \frac{a_{n1}}{a_{11}}R_1}} \left[\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ \boxed{0} & a_{22} - \frac{a_{21}}{a_{11}}a_{12} & \cdots & a_{2n} & b_2 - \frac{a_{21}}{a_{11}}b_1 \\ \boxed{\vdots} & \vdots & \ddots & \vdots & \vdots \\ \boxed{0} & a_{n2} - \frac{a_{n1}}{a_{11}}a_{12} & \cdots & a_{nn} & b_n - \frac{a_{n1}}{a_{11}}b_1 \end{array}\right] \; \left.\vphantom{\begin{array}{c} a \\ a \\ a \end{array}}\right\} \textit{Removing } x_1$$

$$\xrightarrow{\substack{R_3 \leftarrow R_3 - \frac{a_{32}}{a_{22}}R_2 \\ \vdots \\ R_n \leftarrow R_n - \frac{a_{n2}}{a_{22}}R_2}} \left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & b_1 \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} & b_2 \\ \vdots & \boxed{0} & a_{33} - \frac{a_{32}}{a_{22}}a_{23} & \ddots & \vdots & b_3 - \frac{a_{32}}{a_{22}}b_2 \\ \vdots & \boxed{\vdots} & \vdots & \ddots & \vdots & \vdots \\ 0 & \boxed{0} & a_{n3} - \frac{a_{n2}}{a_{22}}a_{23} & \cdots & a_{nn} & b_n - \frac{a_{n2}}{a_{22}}b_2 \end{array}\right] \; \left.\vphantom{\begin{array}{c} a \\ a \\ a \\ a \\ a \end{array}}\right\} \textit{Removing } x_2$$

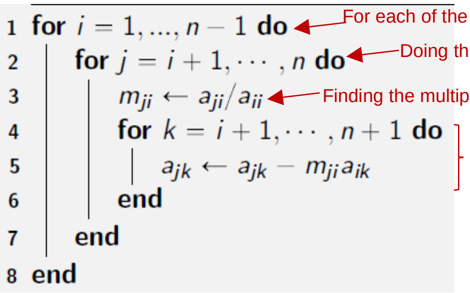
$$\vdots$$

$$\xrightarrow{\substack{R_{i+1} \leftarrow R_{i+1} - \frac{a_{i+1,i}}{a_{ii}}R_i \\ \vdots \\ R_n \leftarrow R_n - \frac{a_{ni}}{a_{ii}}R_i}} \left[\begin{array}{cccccccc|c} a_{11} & a_{12} & \cdots & \cdots & \cdots & \cdots & a_{1n} & & b_1 \\ 0 & a_{22} & \cdots & \cdots & \cdots & \cdots & a_{2n} & & b_2 \\ \vdots & 0 & \ddots & \cdots & \cdots & \ddots & \vdots & & \vdots \\ 0 & \cdots & a_{ii} & \cdots & \cdots & \vdots & \vdots & & \vdots \\ 0 & \cdots & 0 & a_{i+1,i+1} - \frac{a_{i+1,i}}{a_{ii}}a_{i,i+1} & \cdots & \vdots & a_{i+1,n} - \frac{a_{i+1,i}}{a_{ii}}a_{in} & & b_{i+1} - \frac{a_{i+1,i}}{a_{ii}}b_i \\ \vdots & \vdots & \boxed{\vdots} & \vdots & \cdots & \ddots & \vdots & & \vdots \\ 0 & 0 & \boxed{0} & a_{n,i+1} - \frac{a_{ni}}{a_{ii}}a_{i,i+1} & \cdots & \cdots & a_{nn} - \frac{a_{ni}}{a_{ii}}a_{in} & & b_n - \frac{a_{i+1,i}}{a_{ii}}b_i \end{array}\right] \; \left.\vphantom{\begin{array}{c} a \\ a \\ a \\ a \\ a \end{array}}\right\} \textit{Removing } x_i$$

$$\vdots$$

- We will go through the process until we eliminate $n-1$ variables. For the last variable, we will not need to remove it since there will only be 1 equation that contains it

Pseudocode:



```
for $i = 1, \dots, n-1$ do
    for $j = i+1, \cdots, n$ do
        $m_{ji} \leftarrow a_{ji}/a_{ii}$
        for $k = i+1, \cdots, n+1$ do
            $a_{jk} \leftarrow a_{jk} - m_{ji}a_{ik}$
        end
    end
end
```

For each of the $n-1$ variables

Doing the elimination step for the remaining rows other than the pivot row

Finding the multiplier

Doing the minus-ing step for all elements in the row with the multiplier

Time Complexity: $O(n^3)$

Backward Substitution

We will use the augmented matrix from the end of the elimination step

Solving for x_n:

$$\left(\begin{array}{cccc|c} a_{11} & a_{12} & \cdots & a_{1n} & a_{1,n+1} \\ & a_{22} & \cdots & a_{2n} & a_{2,n+1} \\ & & \ddots & \vdots & \vdots \\ & & & a_{nn} & a_{n,n+1} \end{array}\right)$$

Using last equation:

$$a_{nn}x_n = a_{n,n+1}$$
$$\boxed{\Rightarrow x_n = \frac{a_{n,n+1}}{a_{nn}}}$$

Solving for x_{n-1}:

$$\left(\begin{array}{cccccc|c} a_{11} & a_{12} & \cdots & \cdots & & a_{1n} & a_{1,n+1} \\ & a_{22} & \cdots & \cdots & & a_{2n} & a_{2,n+1} \\ & & \ddots & \vdots & & \vdots & \vdots \\ & & & a_{n-1,n-1} & & a_{n-1,n} & a_{n-1,n+1} \\ & & & & & a_{nn} & a_{n,n+1} \end{array}\right)$$

Using the second last equation:

$$a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = a_{n-1,n+1}$$
$$\Rightarrow \boxed{x_{n-1} = \frac{1}{a_{n-1,n-1}}\left(a_{n-1,n+1} - a_{n-1,n}x_n\right)}$$

Solving for x_i:

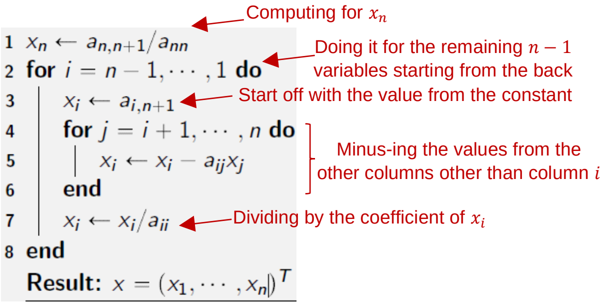
$$\left(\begin{array}{cccccccc|c} a_{11} & a_{12} & \cdots & \cdots & \cdots & & \cdots & a_{1n} & a_{1,n+1} \\ & a_{22} & \cdots & \cdots & \cdots & & \cdots & a_{2n} & a_{2,n+1} \\ & & \ddots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ & & & a_{i,i} & \cdots & & \cdots & a_{i,n} & a_{i,n+1} \\ & & & & \ddots & & \vdots & \vdots & \vdots \\ & & & & & & a_{n-1,n-1} & a_{n-1,n} & a_{n-1,n+1} \\ & & & & & & & a_{nn} & a_{n,n+1} \end{array}\right)$$

Using the $i-th$ equation:

$$a_{ii}x_i + \cdots + a_{i,n-1}x_{n-1} + a_{in}x_n = a_{i,n+1}$$
$$\boxed{\Rightarrow x_i = \frac{1}{a_{ii}}\left(a_{i,n+1} - \sum_{j=i+1}^{n} a_{ij}x_j\right)}$$

- We will start from the last variable and go backwards until the first variable

Pseudocode:



```
$x_n \leftarrow a_{n,n+1}/a_{nn}$
for $i = n-1, \cdots, 1$ do
    $x_i \leftarrow a_{i,n+1}$
    for $j = i+1, \cdots, n$ do
        $x_i \leftarrow x_i - a_{ij}x_j$
    end
    $x_i \leftarrow x_i/a_{ii}$
end
Result: $x = (x_1, \cdots, x_n)^T$
```

Computing for x_n

Doing it for the remaining $n-1$ variables starting from the back

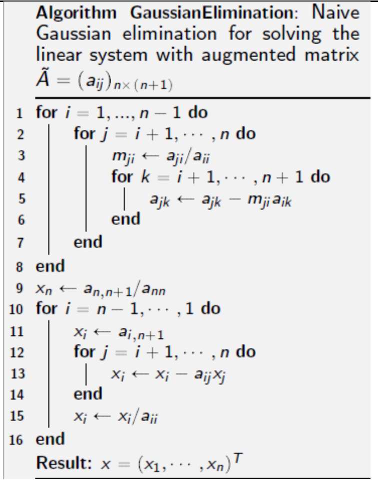
Start off with the value from the constant

Minus-ing the values from the other columns other than column i

Dividing by the coefficient of x_i

Time Complexity: $O(n^2)$

Naïve Gaussian Elimination Complete Pseudocode



Algorithm GaussianElimination: Naive Gaussian elimination for solving the linear system with augmented matrix $\tilde{A} = (a_{ij})_{n\times(n+1)}$

```
for $i = 1, ..., n-1$ do
    for $j = i+1, \cdots, n$ do
        $m_{ji} \leftarrow a_{ji}/a_{ii}$
        for $k = i+1, \cdots, n+1$ do
            $a_{jk} \leftarrow a_{jk} - m_{ji}a_{ik}$
        end
    end
end
$x_n \leftarrow a_{n,n+1}/a_{nn}$
for $i = n-1, \cdots, 1$ do
    $x_i \leftarrow a_{i,n+1}$
    for $j = i+1, \cdots, n$ do
        $x_i \leftarrow x_i - a_{ij}x_j$
    end
    $x_i \leftarrow x_i/a_{ii}$
end
Result: $x = (x_1, \cdots, x_n)^T$
```

Number of Operations:

Subtraction:

$$\boxed{\frac{1}{6}n(n-1)(2n+5)}$$

Multiplication:

$$\boxed{\frac{1}{6}n(n-1)(2n+5)}$$

Division:

$$\boxed{\frac{1}{2}n(n+1)}$$

Total Number of Operations:

$$\frac{1}{6}n(n-1)(2n+5) + \frac{1}{6}n(n-1)(2n+5) + \frac{1}{2}n(n+1)$$
$$= \boxed{\frac{2}{3}n(n-1)(2n+5) + \frac{1}{2}n(n+1)}$$

Time Complexity: $O(n^3)$

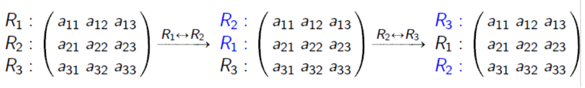
Note that when n is large, the total number of operations is dominated by $\frac{2n^3}{3}$

Division By Zero Handling

Row Labelling

Make use of an array $[r_1, r_2, r_3]$ to keep track of the row labels

r_i : Index of physical row with label R_i



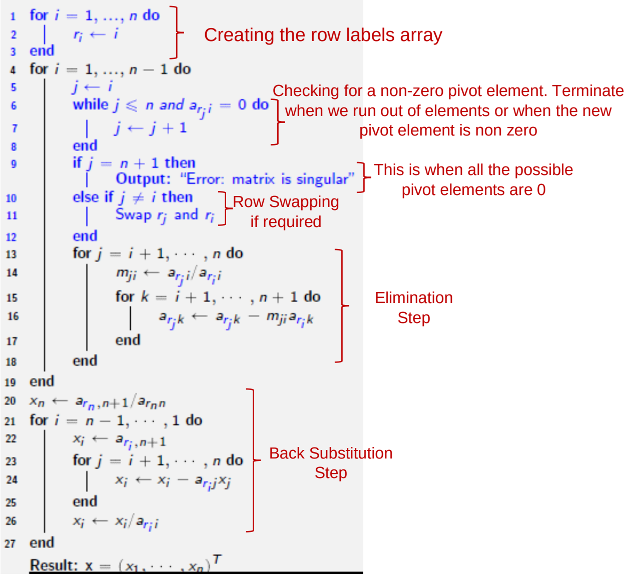
$$\begin{array}{l} R_1: \\ R_2: \\ R_3: \end{array}\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \xrightarrow{R_1 \leftrightarrow R_2} \begin{array}{l} R_2: \\ R_1: \\ R_3: \end{array}\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \xrightarrow{R_2 \leftrightarrow R_3} \begin{array}{l} R_3: \\ R_1: \\ R_2: \end{array}\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Figure 7 Example of Row Swapping with Row Labels

Management of Zero Division

1) Check if the pivot element, $a_{r_i,i} = 0$
2) If $a_{r_i,i} = 0$, then we just need to replace the pivot element by exchanging r_i with some r_j such that j
    a. Greater than i to ensure that the first $i-1$ variables are already eliminated in the row labelled by j
    b. Satisfy that $a_{r_j,i} \neq 0$ to avoid getting another zero pivot element

Pseudocode



```
for $i = 1, ..., n$ do
    $r_i \leftarrow i$
end
for $i = 1, ..., n-1$ do
    $j \leftarrow i$
    while $j \leqslant n$ and $a_{r_j i} = 0$ do
        $j \leftarrow j+1$
    end
    if $j = n+1$ then
        Output: "Error: matrix is singular"
    else if $j \neq i$ then
        Swap $r_j$ and $r_i$
    end
    for $j = i+1, \cdots, n$ do
        $m_{ji} \leftarrow a_{r_j i}/a_{r_i i}$
        for $k = i+1, \cdots, n+1$ do
            $a_{r_j k} \leftarrow a_{r_j k} - m_{ji}a_{r_i k}$
        end
    end
end
$x_n \leftarrow a_{r_n,n+1}/a_{r_n n}$
for $i = n-1, \cdots, 1$ do
    $x_i \leftarrow a_{r_i,n+1}$
    for $j = i+1, \cdots, n$ do
        $x_i \leftarrow x_i - a_{r_i j}x_j$
    end
    $x_i \leftarrow x_i/a_{r_i i}$
end
Result: $x = (x_1, \cdots, x_n)^T$
```

Creating the row labels array

Checking for a non-zero pivot element. Terminate when we run out of elements or when the new pivot element is non zero

This is when all the possible pivot elements are 0

Row Swapping if required

Elimination Step

Back Substitution Step

Time Complexity: $O(n^3)$

- Note that this code is like the Naïve Gaussian Elimination, and we just need to consider the row swapping and it will take at most $O(n^2)$. Therefore, the time complexity is still the same

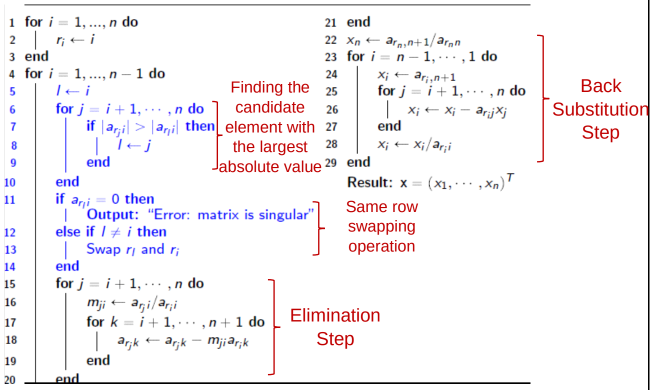
Topic 6: Methods for solving linear systems – Gaussian elimination with pivoting strategies

Partial Pivoting

Main Idea: Choose the pivot element as the one with the largest absolute value

Reason: When we are doing the elimination step, for instance we are doing $a_{22} \leftarrow a_{22} - \frac{a_{21}}{a_{11}} a_{12}$ and a_{11} is the pivot element and is small. If we divide by a small coefficient, $\frac{a_{21}}{a_{11}} a_{12}$ will be big and cause a loss of significant digits in the original system

Pseudocode:



```
for $i = 1, ..., n$ do
    $r_i \leftarrow i$
end
for $i = 1, ..., n-1$ do
    $l \leftarrow i$
    for $j = i+1, \cdots, n$ do
        if $|a_{r_j i}| > |a_{r_l i}|$ then
            $l \leftarrow j$
        end
    end
    if $a_{r_l i} = 0$ then
        Output: "Error: matrix is singular"
    else if $l \neq i$ then
        Swap $r_l$ and $r_i$
    end
    for $j = i+1, \cdots, n$ do
        $m_{ji} \leftarrow a_{r_j i}/a_{r_i i}$
        for $k = i+1, \cdots, n+1$ do
            $a_{r_j k} \leftarrow a_{r_j k} - m_{ji} a_{r_i k}$
        end
    end
end
$x_n \leftarrow a_{r_n, n+1}/a_{r_n n}$
for $i = n-1, \cdots, 1$ do
    $x_i \leftarrow a_{r_i, n+1}$
    for $j = i+1, \cdots, n$ do
        $x_i \leftarrow x_i - a_{r_i j} x_j$
    end
    $x_i \leftarrow x_i/a_{r_i i}$
end
Result: $\mathbf{x} = (x_1, \cdots, x_n)^T$
```

Finding the candidate element with the largest absolute value

Same row swapping operation

Elimination Step

Back Substitution Step

Time Complexity: $O(n^3)$

- Note that since this is only a slight modification, this just ensures that we will run through all the possible candidates as compared to the one that handles zero division. Therefore, the time complexity does not change

Scaled Partial Pivoting

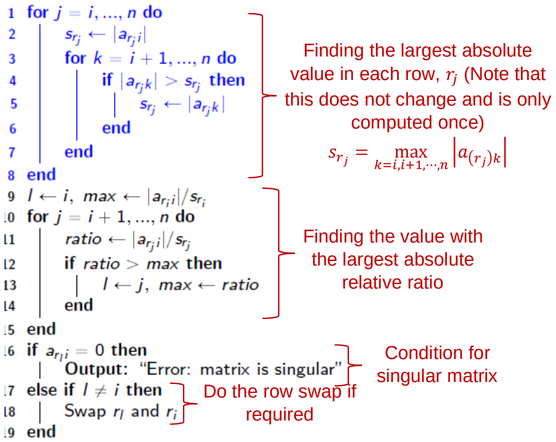
Main Idea: Choose the pivot element as the one with the largest absolute relative ratio

Reason: This is building upon partial pivoting's reason and its flaw. When one of the values is relatively small as compared to the rest of the elements in the rows, it will cause the row to be selected as the pivot row. $a_{22} \leftarrow a_{22} - \frac{a_{21}}{a_{11}} a_{12}$. Suppose a_{11} is very small as compared to a_{12}, this will cause the loss of significant digits from the original system since we are taking the difference with a much larger number

Way to find the Pivot Element:

- Find the elements with the largest absolute value for each of the row and record them as s_{r_i} where r_i is the row number (*Note that the largest absolute values follow row swapping, and it is only computed once*)
- For each of the values in the column that we are trying to eliminate, we find the value that has the largest absolute relative ratio taking a_{ii}/s_{r_i} to find the pivot element
- Continue with the normal Gaussian Elimination after the pivot element is determined

Pseudocode



```
for $j = i, ..., n$ do
    $s_{r_j} \leftarrow |a_{r_j i}|$
    for $k = i+1, ..., n$ do
        if $|a_{r_j k}| > s_{r_j}$ then
            $s_{r_j} \leftarrow |a_{r_j k}|$
        end
    end
end
$l \leftarrow i,\ max \leftarrow |a_{r_i i}|/s_{r_i}$
for $j = i+1, ..., n$ do
    $ratio \leftarrow |a_{r_j i}|/s_{r_j}$
    if $ratio > max$ then
        $l \leftarrow j,\ max \leftarrow ratio$
    end
end
if $a_{r_l i} = 0$ then
    Output: "Error: matrix is singular"
else if $l \neq i$ then
    Swap $r_l$ and $r_i$
end
```

Finding the largest absolute value in each row, r_j (Note that this does not change and is only computed once)

$$s_{r_j} = \max_{k=i,i+1,\cdots,n} \left|a_{(r_j)k}\right|$$

Finding the value with the largest absolute relative ratio

Condition for singular matrix

Do the row swap if required

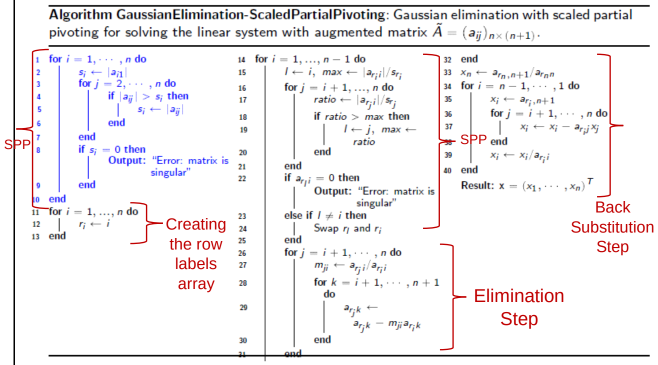
Divisions: $\boxed{\frac{(n+2)(n-1)}{2}}$

Comparisons: $\boxed{\frac{3}{2}n(n-1)}$

Note that if the largest absolute value is constantly computed, the comparisons will increase to $\frac{n(n-1)(2n+5)}{6}$

Time Complexity (for Scaled Partial Pivoting Part): $O(n^2)$

Overall Pseudocode for Gaussian Elimination with Scaled Partial Pivoting



Algorithm GaussianElimination-ScaledPartialPivoting: Gaussian elimination with scaled partial pivoting for solving the linear system with augmented matrix $\tilde{A} = (a_{ij})_{n \times (n+1)}$.

```
for $i = 1, \cdots, n$ do
    $s_i \leftarrow |a_{i1}|$
    for $j = 2, \cdots, n$ do
        if $|a_{ij}| > s_i$ then
            $s_i \leftarrow |a_{ij}|$
        end
    end
    if $s_i = 0$ then
        Output: "Error: matrix is singular"
    end
end
for $i = 1, ..., n$ do
    $r_i \leftarrow i$
end
for $i = 1, ..., n-1$ do
    $l \leftarrow i,\ max \leftarrow |a_{r_i i}|/s_{r_i}$
    for $j = i+1, ..., n$ do
        $ratio \leftarrow |a_{r_j i}|/s_{r_j}$
        if $ratio > max$ then
            $l \leftarrow j,\ max \leftarrow ratio$
        end
    end
    if $a_{r_l i} = 0$ then
        Output: "Error: matrix is singular"
    else if $l \neq i$ then
        Swap $r_l$ and $r_i$
    end
    for $j = i+1, \cdots, n$ do
        $m_{ji} \leftarrow a_{r_j i}/a_{r_i i}$
        for $k = i+1, \cdots, n+1$ do
            $a_{r_j k} \leftarrow a_{r_j k} - m_{ji} a_{r_i k}$
        end
    end
end
$x_n \leftarrow a_{r_n, n+1}/a_{r_n n}$
for $i = n-1, \cdots, 1$ do
    $x_i \leftarrow a_{r_i, n+1}$
    for $j = i+1, \cdots, n$ do
        $x_i \leftarrow x_i - a_{r_i j} x_j$
    end
    $x_i \leftarrow x_i/a_{r_i i}$
end
Result: $\mathbf{x} = (x_1, \cdots, x_n)^T$
```

SPP

Creating the row labels array

SPP

Elimination Step

Back Substitution Step

Overall Time Complexity: $O(n^3)$

- Note that since the elimination step still takes $O(n^3)$ time to complete, the overall time complexity will still be $O(n^3)$ since the scaled partial pivoting part only takes $O(n^2)$

Topic 7: Methods for solving linear systems – LU Factorization

LU Factorization

Usage:

When the linear systems have the same coefficient matrix but different right-hand side

$$\boxed{Ax_1 = b_1, Ax_2 = b_2, \cdots, Ax_p = b_p}$$

Where $b_1, b_2, \cdots, b_p$ are given, and $x_1, x_2, \cdots x_p$ are unknown vectors

Matrices that are used:

A – Original Coefficient Matrix
U – Coefficient Matrix after the row elimination (**Upper Triangular Matrix**)
b – Right hand side vector
L – Matrix with elementary steps that converts the row eliminated b' vector back to b (**Lower Triangular Matrix**)

Algorithm:

[Step 1] Preprocess the coefficient matrix A to obtain

- The multipliers $m_{ji}, i = 1, \cdots, n-1, j = i+1, \cdots, n$
- Coefficient Matrix U after the elimination step

[Step 2] For each $k = 1, 2, \cdots, p$

- Apply elimination to b_k by using multipliers $m_{ji}, i = 1, \cdots, n-1, j = i+1, \cdots, n$ stored previously to get b'_k.
- Perform backward substitution by using the linear system $(U|b'_k)$

Proof for LU factorization:

The original linear system can be rewritten as:

$$Ax = b = Lb'$$

We also know that the matrix relation after the row elimination:

$$Ux = b'$$

Left Multiplying both sides of the equation by L

$$LUx = Lb' = Ax$$
$$\Rightarrow LU = A$$

Solving using LU factorization:

From the above relation:

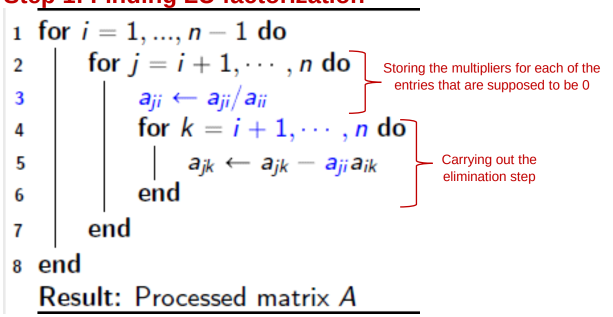
$$\boxed{Ax = b \Leftrightarrow LUx = b}$$

Forward substitution: Solve $\boxed{Ly = b}$ for y (Taking $Ux = y$)
Time Complexity: $O(n^2)$ due to triangular structure

Backward substitution: Solve $\boxed{Ux = y}$ for x
Time Complexity: $O(n^2)$ due to triangular structure

Pseudocode:

Step 1: Finding LU factorization



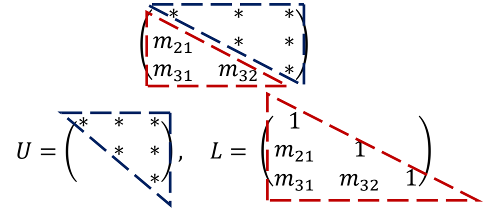
```
for i = 1, ..., n − 1 do
    for j = i + 1, ··· , n do
        $a_{ji} \leftarrow a_{ji}/a_{ii}$
        for k = i + 1, ··· , n do
            $a_{jk} \leftarrow a_{jk} - a_{ji}a_{ik}$
        end
    end
end
Result: Processed matrix A
```

Storing the multipliers for each of the entries that are supposed to be 0

Carrying out the elimination step

Suppose $A = \begin{pmatrix} * & * & * \\ * & * & * \\ * & * & * \end{pmatrix}$

After this algorithm, we will get the following matrix



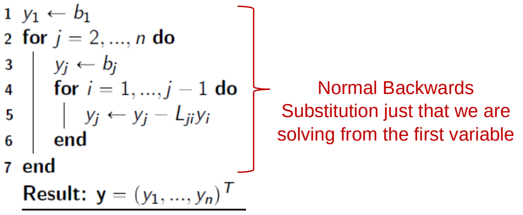
$$\begin{pmatrix} * & * & * \\ m_{21} & * & * \\ m_{31} & m_{32} & * \end{pmatrix}$$

$$U = \begin{pmatrix} * & * & * \\ & * & * \\ & & * \end{pmatrix}, \quad L = \begin{pmatrix} 1 & & \\ m_{21} & 1 & \\ m_{31} & m_{32} & 1 \end{pmatrix}$$

- We can see that with this algorithm, we will be able to get the following LU matrix out. Note that the m_{ij} entries are supposed to be 0 after the elimination step but we are storing the multipliers here

Time Complexity: $O(n^3)$

Step 2: Forward Substitution



```
$y_1 \leftarrow b_1$
for $j = 2, ..., n$ do
    $y_j \leftarrow b_j$
    for $i = 1, ..., j - 1$ do
        $y_j \leftarrow y_j - L_{ji}y_i$
    end
end
Result: $\mathbf{y} = (y_1, ..., y_n)^T$
```

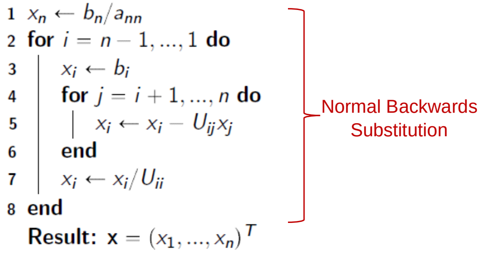
Normal Backwards Substitution just that we are solving from the first variable

We will be solving for $Ly = b$

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ L_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ L_{n1} & \cdots & L_{n,n-1} & 1 \end{pmatrix} \mathbf{y} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Time Complexity: $O(n^2)$

Step 3: Backward Substitution



```
$x_n \leftarrow b_n/a_{nn}$
for $i = n - 1, ..., 1$ do
    $x_i \leftarrow b_i$
    for $j = i + 1, ..., n$ do
        $x_i \leftarrow x_i - U_{ij}x_j$
    end
    $x_i \leftarrow x_i/U_{ii}$
end
Result: $\mathbf{x} = (x_1, ..., x_n)^T$
```

Normal Backwards Substitution

We will be solving for $Ux = y$. Note that y is being used from the original equation

Time Complexity: $O(n^2)$

Existence of LU Factorization:

NOT all matrices have LU factorization, we may need to do some row swapping before we can get the LU factorization

PA = LU Factorization

Permutation Matrix

Definition 9.4:
A permutation matrix is an $n \times n$ matrix consisting of all zeros, except for a single 1 in every row and column

It is like applying arbitrary row exchanges to the $n \times n$ identity matrix

Example:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \cdots$$

Fundamental Theorem of Permutation Matrices:

Theorem 9.5:
Let P be the $n \times n$ permutation matrix formed by a particular set of row exchanges applied to the identity matrix. Then, for any $n \times n$ matrix A, PA is the matrix obtained by applying exactly the same set of row exchanges to A

Solving for $PA = LU$ factorization

Keep track of the row swapping operations using a permutation matrix P

From the LU factorization, we know that the resultant matrix from the row elimination can be formulated as $A = LU$. However, since we did row swapping we will need to make it:

$$PA = LU$$

Solve for the following equation:

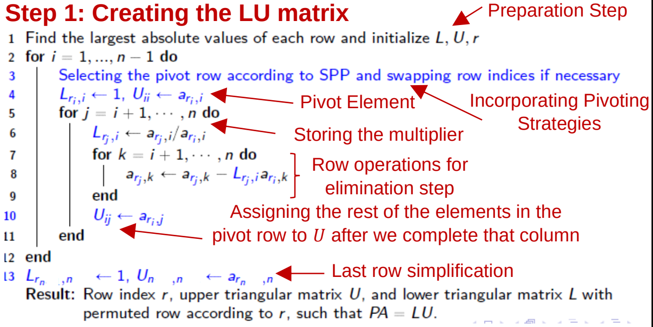
$$PAx = Pb$$
$$\Rightarrow LUx = Pb$$

Forward substitution: Solve $Ly = Pb$ for y (Taking $Ux = y$)
Time Complexity: $O(n^2)$ due to triangular structure

Backward substitution: Solve $Ux = y$ for x
Time Complexity: $O(n^2)$ due to triangular structure

Pseudocode

Step 1: Creating the LU matrix



Preparation Step

```
Find the largest absolute values of each row and initialize $L, U, r$
for $i = 1, ..., n-1$ do
    Selecting the pivot row according to SPP and swapping row indices if necessary
    $L_{r_i,i} \leftarrow 1,\ U_{ii} \leftarrow a_{r_i,i}$
    for $j = i+1, \cdots, n$ do
        $L_{r_j,i} \leftarrow a_{r_j,i}/a_{r_i,i}$
        for $k = i+1, \cdots, n$ do
            $a_{r_j,k} \leftarrow a_{r_j,k} - L_{r_j,i}a_{r_i,k}$
        end
        $U_{ij} \leftarrow a_{r_i,j}$
    end
end
$L_{r_n,n} \leftarrow 1,\ U_{n,n} \leftarrow a_{r_n,n}$
Result: Row index $r$, upper triangular matrix $U$, and lower triangular matrix $L$ with
        permuted row according to $r$, such that $PA = LU$.
```

Pivot Element

Incorporating Pivoting Strategies

Storing the multiplier

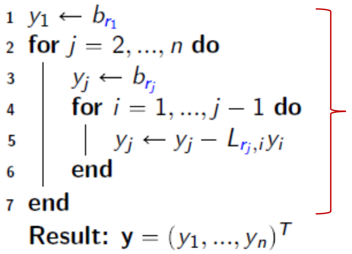
Row operations for elimination step

Assigning the rest of the elements in the pivot row to U after we complete that column

Last row simplification

Time Complexity: $O(n^3)$ due to the triple for loop

Step 2: Forward Substitution



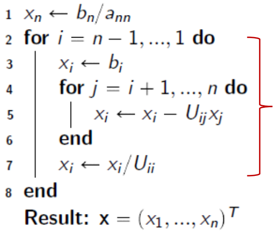
```
$y_1 \leftarrow b_{r_1}$
for $j = 2, ..., n$ do
    $y_j \leftarrow b_{r_j}$
    for $i = 1, ..., j-1$ do
        $y_j \leftarrow y_j - L_{r_j,i}y_i$
    end
end
Result: $\mathbf{y} = (y_1, ..., y_n)^T$
```

Forward Substitution as per the normal LU factorization just that we are using the row labels

- Note that the Pb is already considered with our label array and we do not need to explicitly create a P matrix since what it does essentially is to swap the rows which is given by the row labels

Time Complexity: $O(n^2)$ due to double for loop

Step 3: Backward Substitution



```
$x_n \leftarrow b_n/a_{nn}$
for $i = n-1, ..., 1$ do
    $x_i \leftarrow b_i$
    for $j = i+1, ..., n$ do
        $x_i \leftarrow x_i - U_{ij}x_j$
    end
    $x_i \leftarrow x_i/U_{ii}$
end
Result: $\mathbf{x} = (x_1, ..., x_n)^T$
```

Backward Substitution as per the normal LU factorization just that we are using the row labels

Time Complexity: $O(n^2)$ due to double for loop

Pros and Cons of LU or $PA = LU$

Pros:
Computational time can be reduced compared to Gaussian elimination. Once the LU factorization is obtained, just need to solve two triangular system with time complexity $O(n^2)$

Cons:
If n is large, the memory needed to store both L and Umight be consuming

Topic 8: Methods for solving linear systems – Cholesky Factorization

Symmetric Positive-Definite Matrices

Definition 10.1:
The $n \times n$ matrix A is symmetric if $A^T = A$. The matrix A is positive-definite if $x^T Ax > 0$ for all vectors $x \neq 0$

Proposition 1:
If the $n \times n$ matrix A is symmetric, then A is positive-definite if and only if all its eigenvalues are positive

Methods for checking if a matrix is symmetric positive definite

Method 1: Check $x^T Ax$ noting $x \neq 0$

- Expand out $x^T Ax$ and do completing the square to check if the whole expression > 0.
- Find a counter example if we want to argue that it is ≤ 0

Method 2: Finding the eigenvalues

- Use the determinant method to find out whether all the eigenvalues are positive
- $\det(A - \lambda I)$ – We will get a characteristic polynomial and we will solve for it.

Definition 10.4:
A principal submatrix of a square matrix A is a square submatrix whose diagonal entries are diagonal entries of A

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

Principal submatrices:

$$\underbrace{a_{11}, a_{22}, a_{33}}_{1 \times 1}, \underbrace{\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix}, \begin{pmatrix} a_{11} & a_{13} \\ a_{31} & a_{33} \end{pmatrix}}_{2 \times 2}$$

Example 10.6:
If the $n \times n$ matrix A is symmetric positive definite, the diagonals are positive.

However, note that the converse is not true. Which is that if the diagonals are positive, it may not mean that the matrix is symmetric positive-definite (We will need to check this)

Proposition 2:

Any principal submatrix of a symmetric positive-definite matrix is symmetric positive-definite

Cholesky Factorization

Every symmetric positive-definite matrix A can be factored as

$$\boxed{A = R^T R}$$

For an upper-triangular matrix R

Pros:
As compared to the LU factorization method, this method only requires us to store one upper-triangular matrix (saving almost half the storage)

Main Idea:
Use row/column operations to reduce a positive definite matrix into the identity matrix. Express the row/column operations in matrix form and do them symmetrically

Repeat Iteratively the following:
1a. Apply row and column operations to introduce zeros into the first column and row
1b. Apply appropriate row and column scaling so that the first diagonal entry becomes 1
2a. Apply row and column operations to introduce zeros into the second column and row
2b. Apply appropriate row and column scaling so that the second diagonal entry becomes 1
⋮
n. Apply row and column scaling so that the last diagonal entry becomes 1 (Note that we don't need to do any row/column operations for this)

Formula for Cholesky Factorization

Note that A can be decomposed into the following form

$$A = \underbrace{\left(\begin{array}{c|c} \sqrt{a_{11}} & \frac{\mathbf{u}_1^T}{\sqrt{a_{11}}} \\ \hline \mathbf{0} & I \end{array}\right)^T}_{:=R_1^T} \underbrace{\left(\begin{array}{c|c} 1 & \mathbf{0}^T \\ \hline \mathbf{0} & K_1 - \frac{1}{a_{11}}\mathbf{u}_1\mathbf{u}_1^T \end{array}\right)}_{:=A_1} \underbrace{\left(\begin{array}{c|c} \sqrt{a_{11}} & \frac{\mathbf{u}_1^T}{\sqrt{a_{11}}} \\ \hline \mathbf{0} & I \end{array}\right)}_{:=R_1}$$

We can iteratively break down the row/column operations and scaling using this R matrix and our goal is to get to the point where A is I

Steps:

Step 1: Computing the R Matrix
Suppose:

$$\boxed{A = \left(\begin{array}{c|c} a_{ii} & \boldsymbol{u_i^T} \\ \hline \boldsymbol{u_i} & \boldsymbol{K_i} \end{array}\right)}$$

Note that u_i could be a vector and not scalar, K_i could be a matrix and not a scalar

Compute the following:

$$\boxed{\begin{gathered} R_{ii} = \sqrt{a_{ii}} \\ \frac{u_i^T}{R_{ii}} \\ \tilde{A}_i = K_i - \frac{u_i}{R_{ii}} \frac{u_i^T}{R_{ii}} \end{gathered}}$$

$$R = \left(\begin{array}{c|c} R_{ii} & \frac{u_i^T}{R_{ii}} \\ \hline & \end{array}\right)$$

$$\tilde{A} = \left(\begin{array}{c|c} & \\ \hline & \tilde{A}_i \end{array}\right)$$

Do this iteratively until the $\widetilde{A_\iota}$ is a scalar and we just need to compute R_{ii} and put into the matrix R

Step 2: Solving the linear system
We know that

$$\boxed{\begin{gathered} A = R^T R \\ \Rightarrow R^T Rx = b \end{gathered}}$$

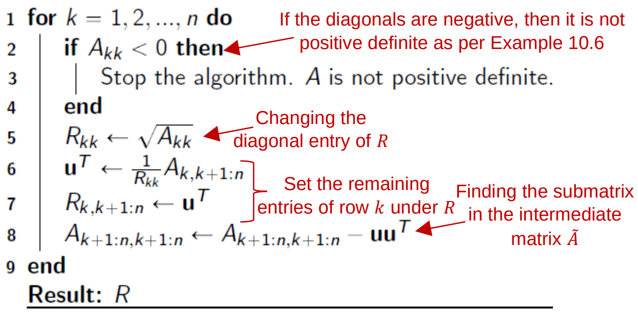
Forward Substitution: Solve $\boxed{R^T y = b}$ for y (Taking $Rx = y$)

Time Complexity: $O(n^2)$ same as the forward substitution for LU factorization since we are solving a lower triangular matrix as well

Backward Substitution: Solve $\boxed{Rx = y}$ for x (Noting that y is from the forward substitution step)

Time Complexity: $O(n^2)$ same as backward substitution for LU factorization since we are solving an upper triangular matrix

Pseudocode



```
for $k = 1, 2, ..., n$ do
    if $A_{kk} < 0$ then
        Stop the algorithm. $A$ is not positive definite.
    end
    $R_{kk} \leftarrow \sqrt{A_{kk}}$
    $\mathbf{u}^T \leftarrow \frac{1}{R_{kk}} A_{k,k+1:n}$
    $R_{k,k+1:n} \leftarrow \mathbf{u}^T$
    $A_{k+1:n,k+1:n} \leftarrow A_{k+1:n,k+1:n} - \mathbf{u}\mathbf{u}^T$
end
Result: $R$
```

If the diagonals are negative, then it is not positive definite as per Example 10.6

Changing the diagonal entry of R

Set the remaining entries of row k under R

Finding the submatrix in the intermediate matrix $\tilde{A}$

Time Complexity: $O(n^3)$

Topic 9: Methods for solving linear systems – Iterative Methods

Decomposition of coefficient matrix, A

Suppose $Ax = b$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

It can be decomposed into $A = \tilde{L} + D + \tilde{U}$

$$\tilde{L} = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{pmatrix}$$

$$D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

$$\tilde{U} = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_{n-1,n} \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

$\tilde{L}$ – Lower triangle of A (entries below the main diagonal)
D – Main diagonal of A
$\tilde{U}$ – Upper triangle of A (entries above the main diagonal)

Jacobi Method

Main Equation:
We know that

$$\begin{aligned} & Ax = b \\ \Leftrightarrow\ & (\tilde{L} + D + \tilde{U})x = b \\ \Leftrightarrow\ & Dx = b - (\tilde{L} + \tilde{U})x \\ \Leftrightarrow\ & \boxed{x = D^{-1}(b - (\tilde{L} + \tilde{U})x)} \end{aligned}$$

$$\boxed{D^{-1} = \begin{pmatrix} \frac{1}{a_{11}} & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{11}} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \frac{1}{a_{nn}} \end{pmatrix}}$$

- This is taking the inverse of all the diagonal entries of D and if diagonal entries are non zero

Computation:

Note that this is an *iterative process* (similar to fixed-point iteration)

From the equation:

$$\boxed{x^{(k+1)} = D^{-1}(b - \tilde{L}x^{(k)} - \tilde{U}x^{(k)})}$$

We will solve for the equations element-wise and we will only use specific elements from the $D^{-1}, \tilde{L}, \tilde{U}$ matrices

Keep iterating until we get closer to the final solution

$$\begin{array}{lllllll} x_1^{(k+1)} = & \frac{1}{a_{11}} & [b_1 & & -a_{12}x_2^{(k)} & & - \cdots & & -a_{1n}x_n^{(k)}] \\ x_2^{(k+1)} = & \frac{1}{a_{22}} & [b_2 & -a_{21}x_1^{(k)} & & -a_{23}x_3^{(k)} & - \cdots & & -a_{2n}x_n^{(k)}] \\ \vdots & \vdots & & \vdots & \ddots & & \ddots & \vdots & \\ x_{n-1}^{(k+1)} = & \frac{1}{a_{n-1,n-1}} & [b_{n-1} & -a_{n-1,1}x_1^{(k)} & - \cdots & & -a_{n-1,n-2}x_{n-2}^{(k)} & & -a_{n-1,n}x_n^{(k)}] \\ x_n^{(k+1)} = & \frac{1}{a_{nn}} & [b_n & -a_{n1}x_1^{(k)} & - \cdots & & & -a_{n,n-1}x_{n-1}^{(k)} &] \end{array}$$

Notation:
$x_i^{(k)}$: $i - th$ entry of the solution vector at the k-th iterate

Pseudocode

```
Set an initial vector $\mathbf{x}^{(0)}$
$\mathbf{d} \leftarrow diag(A)$    (extract the diagonal of A as column vector)
$R \leftarrow A - diag(\mathbf{d})$    $(\tilde{L} + \tilde{U} = A - D)$
for $k = 0, 1, 2, \cdots$ do
    $\mathbf{x}^{(k+1)} = (\mathbf{b} - R\mathbf{x}^{(k)}) \oslash \mathbf{d}$    (vectorized/parallelized operations)
end
Result: Solution of the linear system $\mathbf{x}$
```

R is $\tilde{L} + \tilde{U}$ which is A without the diagonals

Computing the solution vector

Note that $\oslash$ means element-wise division

Element Wise Division:

- Element 1 will work on element 1 of the other matrix

Example:

$$\boxed{\begin{pmatrix} 2 \\ 3 \end{pmatrix} \oslash \begin{pmatrix} 2 \\ 6 \end{pmatrix} = \begin{pmatrix} \frac{2}{2} = 1 \\ \frac{3}{6} = \frac{1}{2} \end{pmatrix}}$$

Time Complexity: $O(n^2)$ per iteration of k assuming A is dense

Stopping Condition

1) Maximum Number of Iterations reach

2) $x^{(k+1)}$ approximately solves the linear system, $i.e. Ax^{(k+1)} \approx b$. E.g. for a pre-defined tolerance, ϵ, check if the following condition is satisfied

$$\boxed{\max_{1 \le i \le n} \left\{ \left| Ax^{(k+1)} - b \right|_i \right\} < \epsilon}$$

- This is to see what the maximum difference between each of the elements and the original solution is. If the maximum is less than the pre-defined tolerance, we can terminate the algorithm

When Jacobi Method works

Definition: The $n \times n$ matrix $A = (a_{ij})$ is strictly diagonally dominant if for each $1 \le i \le n$, $|a_{ii}| > \sum_{j \ne i} |a_{ij}|$

- This is saying that the main diagonal entry has a larger magnitude (absolute value) than the sum of the magnitudes (absolute value) of the remainder of the entries in its row

Theorem 11.3:
If the $n \times n$ matrix A is strictly diagonally dominant, then

1) A is a nonsingular matrix
2) For every vector b and every starting guess, the Jacobi Method applied to $Ax = b$ converges to the (unique solution)

- Note that Strict Diagonal Dominance is only a sufficient condition, without it Jacobi Method may still converge but we are not able to ascertain it (The theorem only tells us one direction)

Iterative Methods for Solving $Ax = b$

Note that we can break up $Ax = b$ into the following form:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

It can be decomposed into $A = \tilde{L} + D + \tilde{U}$

$$\tilde{L} = \begin{pmatrix} 0 & 0 & \cdots & 0 \\ a_{21} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_{n1} & \cdots & a_{n,n-1} & 0 \end{pmatrix}$$

$$D = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}$$

$$\tilde{U} = \begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ 0 & 0 & \cdots & \vdots \\ \vdots & \vdots & \ddots & a_{n-1,n} \\ 0 & 0 & \cdots & 0 \end{pmatrix}$$

Jacobi Method

Jacobi method: $$\boxed{\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - \tilde{L}\mathbf{x}^{(k)} - \tilde{U}\mathbf{x}^{(k)})}$$

or

$$\forall i = 1, ..., n: \quad x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - a_{i1}x_1^{(k)} - \cdots - a_{i,i-1}x_{i-1}^{(k)} - a_{i,i+1}x_{i+1}^{(k)} - \cdots - a_{in}x_n^{(k)}\right]$$

Pros: Able to be parallelized since we are not using any of the current computations to compute the rest of the values

Cons: Might be the slowest among the 3 iterative methods

Gauss-Seidel Method

$$\boxed{\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - \tilde{L}\mathbf{x}^{(k+1)} - \tilde{U}\mathbf{x}^{(k)})}$$

or

$$\forall i = 1, ..., n: \quad x_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - a_{i1}x_1^{(k+1)} - \cdots - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - \cdots - a_{in}x_n^{(k)}\right]$$

Note that the difference from Jacobi is that our L matrix we are making use of the values that are computed from the current iteration. It tries to make it converge faster but we will not be able to parallelize it

Pros: Faster than the Jacobi Method

Cons: Hard to be parallelized since we make use of entries in the current iteration to compute for the future values.

SOR Method

$$\boxed{\mathbf{x}^{(k+1)} = (1-\omega)\mathbf{x}^{(k)} + \omega D^{-1}(\mathbf{b} - \tilde{L}\mathbf{x}^{(k+1)} - \tilde{U}\mathbf{x}^{(k)})}$$

or

$$\forall i = 1, ..., n: \quad x_i^{(k+1)} = (1-\omega)x_i^{(k)} + \frac{\omega}{a_{ii}}\left[b_i - a_{i1}x_1^{(k+1)} - \cdots - a_{i,i-1}x_{i-1}^{(k+1)} - a_{i,i+1}x_{i+1}^{(k)} - \cdots - a_{in}x_n^{(k)}\right]$$

Note that this method tries to take the Gauss-Seidel direction towards the solution and overshoots it.
When $\omega > 1$, it is an over-relaxation. ω is a relaxation parameter

Pros: Might be the fastest among the 3 iterative methods

Cons: Need to choose the parameter ω wisely

Theorem to prove the convergence of Jacobi & Gauss-Seidel method:

Strictly Diagonally Dominant:

Definition: The $n \times n$ matrix $A = (a_{ij})$ is strictly diagonally dominant if for each $1 \le i \le n$, $|a_{ii}| > \sum_{j \ne i} |a_{ij}|$

- This is saying that the main diagonal entry has a larger magnitude (absolute value) than the sum of the magnitudes (absolute value) of the remainder of the entries in its row

Theorem:

If the $n \times n$ matrix A is strictly diagonally dominant, then

- A is a nonsingular matrix
- For every vector b and every starting guess, the Jacobi and Gauss-Seidel Method applied to $Ax = b$ converges to a solution

Note that strict diagonal dominance is only a sufficient condition. The Jacobi Method may still converge in the absence of the strict diagonal dominance.

Spectral Radius Theorem:

Definition: Spectral radius $\rho(B)$ of a square matrix B is the maximum magnitude of its eigenvalues.

Theorem: If the $n \times n$ matrix B has spectral radius $\rho(B) < 1$, and c is arbitrary, then, for any vector x_0, the iteration $x_{k+1} = Bx_k + c$ converges. In fact, there exists a unique x_* such that $\lim_{k\to\infty} x_k = x_*$ and $x_* = Bx_* + c$

- This theorem can be used to prove the convergence of the Jacobi, Gauss-Seidel Method
- Note that we just need to write the $x^{k+1} = D^{-1}(b - Lx^k - Ux^k)$ in the form of $x_* = Bx_* + c$
- Refer to Tutorial 6 Question 3

Eigenvector

Let λ be an eigenvalue of A with corresponding eigenvector v, then we have that:

$$\boxed{Av = \lambda v}$$

Eigenvalues:

Let A be an $n \times n$ matrix and let $f(\lambda) = \det(A - \lambda I)$ be its characteristic polynomial. Then a number λ_0 is an eigenvalue of A if and only if $f(\lambda_0) = 0$

To solve for the eigenvalues, we just need to solve for the characteristic polynomial:

$$\boxed{\det(A - \lambda I) = 0}$$

Properties:

Sum and Product of Eigenvalues:
If A is an $n \times n$ matrix, then the sum of the n eigenvalues of A is the trace of A and the product of the n eigenvalues is the determinant of A

- Note that this only holds if the eigenvalues are all distinct

Positive Eigenvalues of Positive Definite Matrices:
If the $n \times n$ matrix A is symmetric, then A is positive-definite if and only if all its eigenvalues are positive

Mean of the eigenvalues is the mean of the diagonals

$$\frac{a+d}{2} = \frac{\lambda_1 + \lambda_2}{2}$$

Product of the eigenvalues is the det(A)

$$\det(A) = \lambda_1 \lambda_2$$

To find the eigenvalues of a 2 × 2: Using the 2 above facts

$\lambda_1, \lambda_2 = m \pm \sqrt{m^2 - p}$

$$\det(A) = \det\begin{bmatrix} \cancel{a} & \cancel{b} & \textcircled{c} \\ d & e & \cancel{f} \\ g & h & \cancel{i} \end{bmatrix} = a \cdot \det\begin{bmatrix} e & f \\ h & i \end{bmatrix} - b \cdot \det\begin{bmatrix} d & f \\ g & i \end{bmatrix} + c \cdot \det\begin{bmatrix} d & e \\ g & h \end{bmatrix}$$

$$\det\begin{bmatrix} 1 & 3 & 2 \\ -3 & -1 & -3 \\ 2 & 3 & 1 \end{bmatrix} = 1 \cdot \det\begin{bmatrix} -1 & -3 \\ 3 & 1 \end{bmatrix} - (3) \cdot \det\begin{bmatrix} -3 & -3 \\ 2 & 1 \end{bmatrix} + 2 \cdot \det\begin{bmatrix} -3 & -1 \\ 2 & 3 \end{bmatrix}$$

Interpolation Problem:

Problem: Given the pairs of datapoints $(x_0, f(x_0)), (x_1, f(x_1)), \cdots, (x_n, f(x_n))$. We want to find a polynomial of degree n, $P_n(x)$ to connect these points to restore the original function f.
i.e.

$$\boxed{P_n(x_i) = f(x_i), \forall i = 0, 1, \cdots, n}$$

Note that all our interpolating functions must pass through the original interpolating nodes. Note that we have $n+1$ interpolating nodes since the i starts from 0

Weierstrass Approximation Theorem:

Let f be a continuous function $[a, b]$. For any $\epsilon > 0$, there exists a polynomial $P(x)$ such that

$$\boxed{|f(x) - P(x)| < \epsilon}$$

Note:

- Every continuous function can be approximated using polynomial up to any precision
- To get better precision, (smaller ϵ), we can just make use of a higher degree polynomial

Method 1: Normal Method and we just solve from the Gaussian Elimination

[Method 1.] Write $\boxed{P_n(x) = a_0 + a_1 x + \ldots + a_n x^n}$. Solve the linear system:

$$\boxed{\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}}$$

to obtain the coefficients $a_0, a_1, \ldots, a_n$.

Pros: Direct Method (Very intuitive)
Cons: Computation might be tedious

Method 2: Finding the Lagrange Basis Polynomial

[Method 2.] Find the Lagrange basis polynomial.

$$\boxed{L_k(x) = \prod_{j=0,\, j\neq k}^{n} \frac{x - x_j}{x_k - x_j} = \frac{(x - x_0)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_0)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)}}$$

Then the Lagrange interpolating polynomial is

$$\boxed{P(x) = f(x_0)L_0(x) + f(x_1)L_1(x) + \cdots + f(x_n)L_n(x),}$$

Pros: Good for analysis, if we need to interpolate many functions with the same set of interpolating nodes
Cons: Extension for extra data points is not obvious

Uniqueness of Lagrange Interpolating Polynomial (Theorem 13.4):

If $x_0, x_1, \cdots, x_n$ are $n+1$ distinct numbers and f is a function whose values are given at these numbers, then a unique polynomial $P_n(x)$ of degree at most n exists with

$$\boxed{f(x_k) = P_n(x_k), \qquad \forall k = 0, 1, \cdots, n}$$

The polynomial is given by the Lagrange Interpolating Polynomial:

$$\boxed{P_n(x) = f(x_0)L_0(x) + f(x_1)L_1(x) + \cdots + f(x_n)L_n(x)}$$

Where L_k is the kth Lagrange basis polynomial

Note that this uniqueness theorem can be used for any interpolating polynomial. Once we find an interpolating polynomial, it will be at most degree n and if we can find a lower degree interpolating polynomial, that will be the unique interpolating polynomial

Method 3: Computed the divided differences and write down the Newton's polynomial

Divided Differences (Definition):

$$\boxed{f[x_i] = f(x_i), \qquad f[x_0, x_1, \cdots, x_n] = \sum_{k=0}^{n} f(x_k) \prod_{j=0, j\neq k} \frac{1}{x_k - x_j}}$$

Newton's Polynomial:

$$\boxed{P_n(x) = \sum_{k=0}^{n} f[x_0, \cdots, x_k](x - x_0)(x - x_1)\cdots(x - x_{k-1})}$$

Theorem:

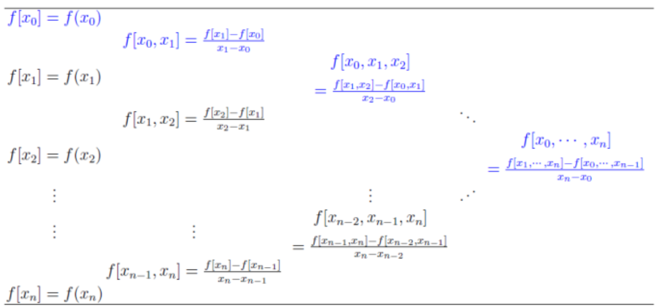
Gives us a method for computing the Newton's Divided Differences using the values from those of smaller values

2^{nd} element to the last element

1^{st}element to the 2^{nd} last element

$$f[x_0, x_1, \cdots, x_n] = \frac{\boxed{f[x_1, x_2, \cdots, x_n]} - \boxed{f[x_0, x_1, \cdots, x_{n-1}]}}{x_n - x_0}$$

Table of Divided Differences:



$f[x_0] = f(x_0)$

$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$

$f[x_1] = f(x_1)$

$f[x_0, x_1, x_2] = \frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$

$f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1}$

$f[x_2] = f(x_2)$

$f[x_0, \cdots, x_n] = \frac{f[x_1, \cdots, x_n] - f[x_0, \cdots, x_{n-1}]}{x_n - x_0}$

$\vdots$

$f[x_{n-2}, x_{n-1}, x_n] = \frac{f[x_{n-1}, x_n] - f[x_{n-2}, x_{n-1}]}{x_n - x_{n-2}}$

$f[x_{n-1}, x_n] = \frac{f[x_n] - f[x_{n-1}]}{x_n - x_{n-1}}$

$f[x_n] = f(x_n)$

To construct the interpolating polynomial:

Take the top entries of each of the columns and write out with the corresponding x values except the last one

$$\boxed{\begin{aligned} P_n(x) = f[x_0] + f[x_0, x_1](x - x_0) + \cdots \\ + f[x_0, \cdots, x_n](x - x_0)(x - x_1)\cdots(x - x_n) \end{aligned}}$$

Note that when we sub in the values of each of the x we should get the corresponding $f(x)$ values since the interpolating polynomial should pass through the interpolating nodes

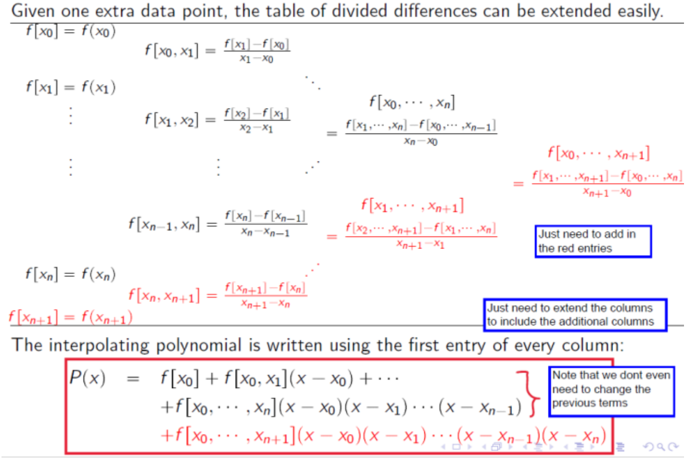
Error of interpolation:

$$\boxed{f(x) - P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n)}$$

- Note that this is the difference between the actual function and the interpolating polynomial

Adding extra interpolating nodes:

Note that this method is the best when we want to add extra interpolating nodes



Given one extra data point, the table of divided differences can be extended easily.

$f[x_0] = f(x_0)$

$f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0}$

$f[x_1] = f(x_1)$

$f[x_0, \cdots, x_n] = \frac{f[x_1, \cdots, x_n] - f[x_0, \cdots, x_{n-1}]}{x_n - x_0}$

$f[x_1, x_2] = \frac{f[x_2] - f[x_1]}{x_2 - x_1}$

$f[x_0, \cdots, x_{n+1}] = \frac{f[x_1, \cdots, x_{n+1}] - f[x_0, \cdots, x_n]}{x_{n+1} - x_0}$

$f[x_1, \cdots, x_{n+1}] = \frac{f[x_2, \cdots, x_{n+1}] - f[x_1, \cdots, x_n]}{x_{n+1} - x_1}$

$f[x_{n-1}, x_n] = \frac{f[x_n] - f[x_{n-1}]}{x_n - x_{n-1}}$

Just need to add in the red entries

$f[x_n] = f(x_n)$

$f[x_n, x_{n+1}] = \frac{f[x_{n+1}] - f[x_n]}{x_{n+1} - x_n}$

$f[x_{n+1}] = f(x_{n+1})$

Just need to extend the columns to include the additional columns

The interpolating polynomial is written using the first entry of every column:

$$\boxed{\begin{aligned} P(x) \quad = \quad & f[x_0] + f[x_0, x_1](x - x_0) + \cdots \\ & + f[x_0, \cdots, x_n](x - x_0)(x - x_1)\cdots(x - x_{n-1}) \\ & + f[x_0, \cdots, x_{n+1}](x - x_0)(x - x_1)\cdots(x - x_{n-1})(x - x_n) \end{aligned}}$$

Note that we dont even need to change the previous terms

Chebyshev Nodes:

On the interval of $[-1,1]$

$$\boxed{x_k = \cos\left(\frac{\left(k+\frac{1}{2}\right)\pi}{n+1}\right), \qquad k = 0,1,\cdots,n}$$

For general interval $[a,b]$

$$\boxed{x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{\left(k+\frac{1}{2}\right)\pi}{n+1}\right), k = 0,1,\cdots,n}$$

Runge's Phenomenon:

Polynomial wiggle near the ends of the interpolation interval. We note that with more data points, it yields worse approximation for the region close to the boundary of the domain

Upper Bound:

Note that this is useful when we want to find the error of interpolation. As per given by the theorem for the error of interpolation

For equally spaced interpolation nodes, we note that the multiplication of all of them has the same effect as the interpolation error. When we are in the middle, x is small so the distance is small, but nearing the end, the x values gets larger and the multiplication of their distance will be larger.

On the interval of $[-1,1]$

$$\boxed{|(x-x_0)(x-x_1)\cdots(x-x_n)| \le \frac{1}{2^n}}$$

For general interval $[a,b]$

$$\boxed{\left|\prod_{k=0}^{n}(x-x_k)\right| \le \frac{\left(\frac{b-a}{2}\right)^{n+1}}{2^n}}$$

Chebyshev Polynomial:

$$\boxed{T_{n+1}(x) = \cos\big((n+1)\,arccos(x)\big)}$$

- Polynomial with Chebyshev nodes being the roots
- We note that T_{n+1} will be a polynomial of degree $n+1$. When $n \ge 0$, the leading coefficient is 2^n

Properties:

1. $g_n(x)$ can be upper bounded

$$\boxed{g_n(x) = |\frac{1}{2^n}\cos\big((n+1)\,arccos\,x\big) \le \frac{1}{2^n}}$$

2. $g_n \to 0\ as\ n \to \infty$
3. $g_n(x)$ attains its maxima if and only if $\cos\big((n+1)\,arccos\,x\big) = \pm 1$
4. Note that the maxima are attained at the boundary points $-1, 1$

Least Square Solution

Problem:

Given the inconsistent system $Ax = b$ $(A \in \mathbb{R}^{m\times n}, m \geq n)$, solve the normal equations:

Method 1: Construct and Solve the Normal Equations

$$\boxed{A^TA\bar{x} = A^Tb}$$

For the least squares solution $\bar{x}$ that minimizes the Euclidean norm of the residual

$$\boxed{r = b - Ax}$$

Note that the residual can be found by taking the $L2$ norm of the residual

$$\boxed{||r||_2}$$

Cholesky Factorization:

Note that this can be used when we have multiple systems of equations with the same A but different b

A^TA is symmetric positive-definite and by Cholesky factorization theorem, there exists an upper triangular matrix R

$$\boxed{A^TA = R^TR}$$

We can solve the inconsistent linear system $Ax = b$ via Cholesky factorization:

1. Compute A^TA, A^Tb
2. Find the Cholesky factorization of A^TA
3. Solve $R^Ty = A^Tb$ for y
4. Solve $R\bar{x} = y$ for $\bar{x}$

Method 2: QR factorization

Problem: Given linearly independent vectors $a_1, a_2, \cdots, a_n \in \mathbb{R}^m$, where $m \geq n$. We want to find n mutually perpendicular unit vectors q_i, $i = 1, \cdots, n$ spanning the same subspace as $a_i, i = 1, \cdots, n$

More numerically stable as compared to the normal equations way,

$$\boxed{A = QR}$$

Where

$Q \in \mathbb{R}^{m\times n}$ – A matrix (might not be square) with orthonormal columns (i.e. $||q_i|| = 1, q_i^Tq_j = 0\ if\ i \neq j$)

$R \in \mathbb{R}^{n\times n}$ – A square upper triangular matrix

$$\underbrace{\begin{pmatrix} \mathrm{a}_1 & \mathrm{a}_2 & \cdots & \mathrm{a}_n \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} \mathrm{q}_1 & \mathrm{q}_2 & \cdots & \mathrm{q}_n \end{pmatrix}}_{Q} \underbrace{\begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \cdots & r_{2n} \\ & & \ddots & \vdots \\ & & & r_{nn} \end{pmatrix}}_{R}$$

Construction of the values of Q and R via the Gram-Schmidt Process:

[Construction of q_1]

Note that the first column of q is basically making the first column of a to be a unit vector and that is the first orthonormal vector and the rest is just trying to make them perpendicular to each other and also noting that they are unit vectors

$$\boxed{y_1 = a_1, \qquad r_{11} = ||y_1||_2, \qquad q_1 = \frac{y_1}{r_{11}}}$$

[Construction of q_2]

$$\boxed{r_{12} = q_1^Ta_2, \qquad y_2 = a_2 - r_{12}q_1, \qquad r_{22} = ||y_2||_2, \qquad q_2 = \frac{y_2}{r_{22}}}$$

Note that the r_{ij} where $i \neq j$ is so that we can construct an orthogonal vector with respect to the other previous orthogonal vectors and using a_i

[Construction of q_j]:

$$\boxed{r_{ij} = q_i^Ta_j, \qquad y_j = a_j - r_{1j}q_1 - r_{2j}q_2 - \cdots - r_{j-1}q_{j-1}, \qquad r_{jj} = ||y_j||_2, \qquad q_j = \frac{y_j}{r_{jj}}}$$

Solving the Least Squares via QR factorization:

$$\boxed{\begin{aligned} &A^TAx = A^Tb \\ &\Leftrightarrow \hat{R}x = \hat{Q}^Tb \end{aligned}}$$

Note that the normal equation can be expressed in the above form using the QR factorization.

Derivation can be found on page 18 of the Lecture Slide 17

Method to solve:

1. Compute the reduce QR factorization of A
2. Compute $\hat{Q}^Tb$
3. Solve $\hat{R}x = \hat{Q}^Tb$ by backwards substitution

Use backwards substitution to solve x with the coefficient matrix as $\hat{R}$ and the values as $\hat{Q}^Tb$

Differentiation:

Taylor's Theorem:

$$f(x) = \overbrace{f(a) + f'(a)(x-a) + \frac{1}{2!}f''(a)(x-a)^2 + \cdots + \frac{1}{n!}f^{(n)}(a)(x-a)^n}^{n^{th}\ \text{order Taylor Polynomial}} + \underbrace{\frac{1}{(n+1)!}f^{(n+1)}(c)(x-a)^{n+1}}_{\text{Error Term}}$$

For some c between a and x

- Note that this is especially useful when we want to create new difference formulas

Expanding out Taylor's Theorem:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)(x-a)^2}{2} + \frac{f'''(c_1)}{3!}(x-a)^3$$

Suppose we want to get $f(x-h)$

We can let $x = x - h$ and we let $a = x$

Trick: Let x be the value that we want to estimate and let a be the original x

$$f(x-h) = f(x) + f'(x)(-h) + \frac{f''(x)(-h)^2}{2} + \frac{f'''(c_1)}{3!}(-h)^3$$

Lagrange Interpolating Polynomial:

Note that we can also make use of the Lagrange Interpolating Polynomial to find an approximation

$$L_k(x) = \prod_{j=0, j\neq k}^{n} \frac{x - x_j}{x_k - x_j}$$
$$P_n(x) = f(x_0)L_0(x) + \cdots + f(x_n)L_n(x)$$

We can differentiate this to find the first order derivation and also make use of the error of interpolation theorem

Error of Interpolation

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)(x-x_1)\cdots(x-x_n)$$

We can make use of both of these to construct an approximation as per Tutorial 11 Question 2

Order of method:

Defined by the **order of the h term** in the error term. Suppose the error consists of a h^2 term, then we have that the order of the method is $O(h^2)$.

Note the higher the order of the method, the more accurate the method. Because when we reduce the value of h, it will reduce the error much more as compared to a method with a lower order

Generalized Intermediate Value Theorem:

Let f be a continuous function on the interval $[a,b]$. Let $x_1, \cdots, x_n$ be points in $[a,b]$ and $a_1, \cdots, a_n > 0$. Then there exists a number c between a and b such that:

$$(a_1 + \cdots + a_n)f(c) = a_1 f(x_1) + \cdots + a_n f(x_n)$$

Note that this is useful when we want to combine the error terms together. When the functional values are different but we have the same function. We can sum the coefficients together and have a functional value that is somewhere between the range

Two-point forward-difference formula (with error term):

$$f'(x) = \overbrace{\frac{f(x+h) - f(x)}{h}}^{\text{Approximation}} - \overbrace{\frac{h}{2}f''(c)}^{\text{Error Term}}$$

Two-point backward-difference formula (with error term):

$$f'(x) = \overbrace{\frac{f(x) - f(x-h)}{h}}^{\text{Approximation}} - \overbrace{\frac{h}{2}f''(c)}^{\text{Error Term}}$$

Three-point centered-difference formula (with error term):

$$f'(x) = \overbrace{\frac{f(x+h) - f(x-h)}{2h}}^{\text{Approximation}} - \overbrace{\frac{h^2}{6}f'''(c)}^{\text{Error Term}}$$

Three-point centered-difference formula for second derivative (with error term):

$$f''(x) = \overbrace{\frac{f(x-h) - 2f(x) - f(x-h)}{h^2}}^{\text{Approximation}} - \overbrace{\frac{h^2}{6}f^{(4)}(c)}^{\text{Error Term}}$$

Optimal Choice of h:

When we are looking at the three-point centered-difference formula

$$h = \sqrt[3]{\frac{3\epsilon}{M}}$$

Where ϵ is the machine epsilon (machine rounding error)

M is the third derivative

Extrapolation or Richardson Extrapolation Formula:

Compute $Q = F_n(h) + K(h)h^n$ but to get a higher order formula for approximating Q

$$F_{n+1}(h) = \frac{\left(2^n F_n\left(\frac{h}{2}\right) - F_n(h)\right)}{2^n - 1}$$

Order of the resulting approximation:

$F_{n+1}(h)$ - Is **at least an order $n+1$ formula** for approximating Q

Suppose we want to create another extrapolation formula with a different $\frac{h}{2}$

For any $Q = F_n(h) + K(h)h^n$:

We can also represent it as:

$$Q = F_n(h) + Bh^n + O(h^{n+1})$$

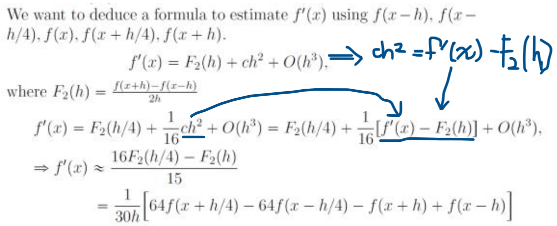
Where $B = K(0)$

Note that we make use of Taylor's Theorem to expand where $h = 0$ for $K(h)$

General Extrapolation Formula:

$$F_{n+1}(h) = \frac{\left(k^n F_n\left(\frac{h}{k}\right) - F_n(h)\right)}{k^n - 1}$$

If we want to have a 5 point centered difference formula with $x+h$, $x-h$, $x - \frac{h}{4}$, $x + \frac{h}{4}$



We want to deduce a formula to estimate $f'(x)$ using $f(x-h)$, $f(x-h/4)$, $f(x)$, $f(x+h/4)$, $f(x+h)$.

$$f'(x) = F_2(h) + ch^2 + O(h^3), \Longrightarrow ch^2 = f'(x) - F_2(h)$$

where $F_2(h) = \frac{f(x+h)-f(x-h)}{2h}$

$$f'(x) = F_2(h/4) + \frac{1}{16}\underline{ch^2} + O(h^3) = F_2(h/4) + \frac{1}{16}[\underline{f'(x) - F_2(h)}] + O(h^3),$$

$$\Rightarrow f'(x) \approx \frac{16F_2(h/4) - F_2(h)}{15}$$

$$= \frac{1}{30h}\left[64f(x+h/4) - 64f(x-h/4) - f(x+h) + f(x-h)\right]$$

Five-Point Centered-Difference Formula:

$$f'(x) \approx \frac{f(x-h) - 8f\left(x - \frac{h}{2}\right) + 8f\left(x + \frac{h}{2}\right) - f(x+h)}{6h}$$

Note that the extrapolation method was applied to the three-point centered difference formula (which is a second order method). Therefore, the order of the formula is at least 3

However, the order of the formula is 4 instead. This is because the error terms can only be even powers of h only

Integration:

Mean Value Theorem for Integrals:

Let f be a continuous function on the interval $[a, b]$ and let g be an integrable function that does not change sign on $[a, b]$. Then there exists a number c between a and b such that

$$\boxed{\int_a^b f(x)g(x)\ dx = f(c)\int_a^b g(x)\ dx}$$

- Can be used to express the error terms of the Trapezoid Rule
- This gives an approximation for a value c that it gives us the area of the rectangle to approximate the integral

Degree of Precision:

Definition: The degree of precision of a numerical integration is the greatest integer k for which all degree k or less polynomials are integrated exactly by this method

To check the degree of precision:

We can do a very brute force method and check through each of the various monomials and check the exact integral and the approximation value

Trapezoid Rule:

Note that this is the one panel version

$$\int_{x_0}^{x_1} f(x)\ dx = \overbrace{\frac{h}{2}(y_0 + y_1)}^{\text{Approximation}} - \overbrace{\frac{h^3}{12}f''(c)}^{\text{Error Term}}$$

where $h = x_1 - x_0$ and c is between x_0 and x_1

Degree of Precision: 1

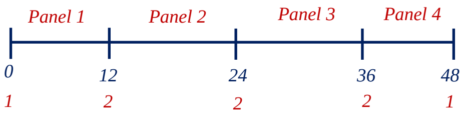
Composite Trapezoid Rule:

$$\int_a^b f(x)\ dx = \overbrace{\frac{h}{2}\left(y_0 + y_m + 2\sum_{i=1}^{m-1} y_i\right)}^{\text{Approximation}} - \overbrace{\frac{(b-a)h^2}{12}f''(c)}^{\text{Error Term}}$$

Where $h = \frac{b-a}{m}$ and $c \in [a, b]$

Order of Method: 2nd Order Method (Because of the h^2 under the error term)

For example, if $m = 4$ (which is where we have 4 panels), we have that $h = \frac{48}{4} = 12$, there will be points from 0 to 48 with spacing of 12 in between such that we can make 4 panels



Intuition: For the formula is basically the computation of a trapezoid where the heights for both sides are $f(x_0), f(x_n)$ and the base is h. Therefore, the form is in $\frac{1}{2}(h)(x_0 + x_n)$. But we note that since we are breaking into m subintervals, those middle nodes will be used to compute the trapezoid twice since it can be used as the left boundary and right boundary and therefore, we have the formula above

Simpson's Rule:

Note that this is the one panel version

$$\int_{x_0}^{x_2} f(x)\,dx = \overbrace{\frac{h}{3}(y_0 + 4y_1 + y_2)}^{\text{Approximation}} - \overbrace{\frac{h^5}{90}f^{(4)}(c)}^{\text{Error Term}}$$

Where $h = x_2 - x_1 = x_1 - x_0$ and c is between x_0 and x_2

Degree of Precision: 3

Composite Simpson's Rule:

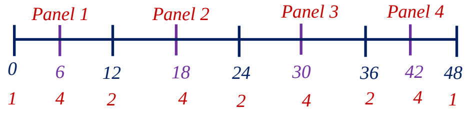
$$\int_a^b f(x)\ dx = \overbrace{\frac{h}{3}\left[y_0 + y_{2m} + 4\sum_{i=1}^{m} y_{2i-1} + 2\sum_{i=1}^{m-1} y_{2i}\right]}^{\text{Approximation}} - \overbrace{\frac{(b-a)h^4}{180}f^{(4)}(c)}^{\text{Error Term}}$$

Where $h = \frac{b-a}{2m}$ and $c \in [a, b]$

Order of Method: 4th Order Method (Because of the h^4 under the error term)

Note that the difference for the Composite Simpson's Rule is that we have to take note of the midpoints between the left and right boundaries of a subinterval

Suppose that we have 4 panels on the interval $[0, 48]$. Note that we have to take into account the midpoints into the calculations



Midpoint Rule:

$$\int_{x_0}^{x_1} f(x)\,dx = \overbrace{hf(w)}^{\text{Approximation}} + \overbrace{\frac{h^3}{24}f''(c)}^{\text{Error Term}}$$

Where $h = (x_1 - x_0)$, w is the midpoint $w_0 + \frac{h}{2}$ and c is between x_0 and x_1

Composite Midpoint Rule:

*Note that this can be used when the **endpoints are not defined***

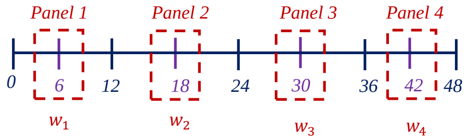
$$\int_a^b f(x)\ dx = \overbrace{h\sum_{i=1}^{m} f(w_i)}^{\text{Approximation}} + \overbrace{\frac{(b-a)h^2}{24}f''(c)}^{\text{Error Term}}$$

Where $h = \frac{b-a}{m}$ and $c \in [a, b]$. The w_i are the midpoints of the m equal subintervals of $[a, b]$. Note that m is the number of panels

Order of Method: 2nd Order Method (Because of the h^2 under the error term) s

Note that as compared to the Composite Simpon's Rule and Composite Trapezoid Rule, we will need to compute the w_i where after we compute h, we find the middle values between them so that those are the midpoints

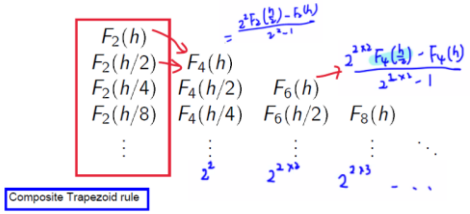
Suppose that we have 4 panels on the interval $[0, 48]$. Note that we have to take into account the midpoints into the calculations



Romberg Integration

Note that this is an extrapolation of the Composite Trapezoid Rule to make it more accurate

$$F_{2k}(h) = \frac{2^{2(k-1)} F_{2(k-1)}\left(\frac{h}{2}\right) - F_{2(k-1)}(h)}{2^{2(k-1)} - 1}, \qquad k > 1$$



Romberg Triangle:

Note that each of the rows is like the number of panels of the Composite Trapezoid Rule used

$$\begin{array}{lllll|l}
R_{11} & & & & & \textit{1 Panel} \\
R_{21} & R_{22} & & & & \textit{2 Panels} \\
R_{31} & R_{32} & R_{33} & & & \textit{4 Panels} \\
R_{41} & R_{42} & R_{42} & R_{44} & & \textit{8 Panels} \\
\vdots & \vdots & \vdots & \vdots & \ddots &
\end{array}$$

$$R_{i2} = \frac{\left(4R_{i,(k-1)} - R_{i-1,k-1}\right)}{3} \quad R_{i3} = \frac{\left(16R_{i,(k-1)} - R_{i-1,k-1}\right)}{15} \quad R_{i4} = \frac{\left(64R_{i,(k-1)} - R_{i-1,k-1}\right)}{63}$$

$$R_{jk} = \frac{2^{2(k-1)} R_{j,k-1} - R_{j-1,k-1}}{2^{2(k-1)} - 1}$$

Note that R_{nn} (the bottom rightmost entry computed) is the best approximation for the definite integral Q, which is a ***2nth-order approximation***

This is a much better approximation as compared to the original Composite Trapezoid Rule which is only a order 2 approximation