



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

Today

LECTURE 1

Introduction

Staff

- Course Coordinator and Lecturer:
Aman G. Kidanemariam (Mechanical Engineering)
- Tutors:
David Rodriguez-Sanchez (Head tutor)
Tony Zahtila
Sheikh Khaleduzzaman Shah
Cas James Eliot Kent
- Contact information of lecturer/tutors is available on Canvas LMS (under the 'Welcome Module')

Critical Information

- Lecture slides and recording will be on Canvas LMS
- Workshops commence in the first Week
- These textbooks will be used as reference:
 - Numerical Recipes in C, Second Edition (1992) (available online)
 - Programming, Problem Solving and Abstraction with C
- You must have taken one of the following prerequisite sets.

Either

- 1, 3 and 4
- 2, 3 and 4
- 1, 3, 5 and 6

1. COMP20005 Engineering Computation
2. COMP10002 Foundations of Algorithms
3. ENGR20004 Engineering Mechanics
4. MAST20029 Engineering Mathematics
5. MAST20009 Vector Calculus
6. MAST20030 Differential Equations

Timetable

All lectures and workshops will be delivered online.

Lectures:

- Wednesday 15:15 – 16:15
- Thursday 13:00 – 14:00

Workshops:

- Tuesday 09:00 – 11:00
- Thursday 09:00 – 11:00
- Friday 09:00 – 11:00
- Friday 11:00 – 13:00

Subject Overview

Numerical Programming for Engineers provides further programming, using the language C, with an emphasis on fundamental algorithms, data structures and numerical problem solving techniques.

Part 1: Weeks 1 – 5

Topics that will be covered include dynamic data structures, and the algorithms that manipulate them (lists, trees); searching and sorting algorithms

Part 2: Weeks 6 – 12

Numerical solutions to problems such as root finding, solving linear equations, interpolation, regression, differentiation, integration and ODEs

Workload

Two one-hour lectures, and a two-hour workshop.

Plus:

- One preview hour for each week (two lectures), including reading the text
- One review hour for each hour of lectures, including reading the text
- Two preparation hours for the workshop
- Three hours of general review/reading

In total, around **12 hours** per week per subject is required, starting immediately.

Getting help

There are a range of mechanisms to use when you need help:

- Ask the lecturer during or after the lecture session
- Post your query to the Canvas LMS discussion forum.
Read other posts and responses while you wait for a response to your query.
- Ask tutors in the workshops, or email them
- Email the lecturer

Assessment

Your final mark is the combination of three components.

Task	Due	Marks
Assignment 1	Week 5	25%
Assignment 2	Week 11	35%
Examination		40%

C programming – What you need to know

- Basic Types and Operators
(int, float, truncation, type casting etc.)
- Control Structures
(if, while, switch, do-while, break, continue)
- Complex data types
(structs, arrays, pointers, dereferencing, strings)
- Functions
(pass by value versus pass by reference, const, return types)

Sample C code

```
1 #include <stdio.h>
2
3 /* define a constant */
4 #define MAX_POINTS 10
5
6
7 /* define custom type */
8 typedef struct {
9     float x;
10    float y;
11 } point_t;
12
13 int main()
14 {
15     /* declare variables */
16     point_t A[MAX_POINTS];
17     point_t *ptr = &A[0];
18     int pointsRead = 0;
19 }
```

Sample C code

```
20
21     /* read floats from stdin */
22     printf("Input x/y value pairs [type non-numeric value to
23             finish]\n");
24     while (pointsRead < MAX_POINTS && scanf("%f %f", &(ptr->x)
25         ,&(ptr->y)) == 2) {
26         printf("read (%f,%f)\n", ptr->x, ptr->y);
27         pointsRead++;
28         ptr++;
29     }
30
31     if (pointsRead == 0) {
32         printf("no points read\n");
33         return 0;
34     } else {
35         printf("%d points read:\n",pointsRead);
36         for (int i = 0; i < pointsRead; i++)
37             printf(" (%f,%f)\n", A[i].x, A[i].y);
38     }

```

Sample C code

```
39
40     /* find */
41     float max_x = A[0].x;
42     float min_y = A[0].y;
43     float sum = 0.0;
44
45     for (int i = 0; i < pointsRead; i++) {
46         if (max_x < A[i].x)
47             max_x = A[i].x;
48         if (min_y > A[i].y)
49             min_y = A[i].y;
50         sum += A[i].x + A[i].y;
51     }
52
53     /* print stats */
54     printf("Points= %d, max X=% .3f, min Y=% .3f, sum= % .3f\n",
55           pointsRead, max_x, min_y, sum);
56
57     return 0;
58 }
59
```

C Programming – Resources

If you have problems understanding this program you must practice your C programming **NOW!**

Resources:

- COMP20005 Textbook (Chapters 1-9): Moffat, A. (2012).
- Programming, Problem Solving, and Abstraction with C, Revised Edition. Pearson. ISBN 9781486010974
- Essential C (45 pages free PDF, see Canvas Resources)
- Numerical Recipes in C (available online, see Canvas)

Expectations

- Self-study is expected, and be proactive.
- Practice programming a lot!
- Do the workshop exercises before you attend workshops. The workshops sessions are the opportunity for you to ask the tutors questions, not for you to do the exercises. If you have no question after doing the exercises, then you may not need to attend the workshops. But workshops still provide you an opportunity to interact with other students (unfortunately virtually this semester) and learn from the tutors.
- Do all the exercises and project assignments.

Academic honesty

All assessed work in this subject is **individual**.

We routinely run sophisticated similarity checking software over all submissions. If you are clever enough to outsmart this software, you are also clever enough to do your own project.

The University's Academic Honesty policy will be applied if duplicate work is detected. Penalties go as far as subject failure, or even termination of enrolment.

Next

- Dynamic memory allocation
- Linked Lists
- Binary Search Trees



THE UNIVERSITY OF

MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

Today

LECTURE 2

Dynamic Memory, Linked Lists, Stacks and
Queues

Dynamic Memory

Often it is unclear at **compile time** of a program, how much memory is required to process or store data.

New chunks of memory, sized according to **run-time** values, can be requested via the function `malloc()`.

The allocated memory is manipulated via a **pointer** variable.

When no longer required, the memory can be handed back via the function `free()`.

Amounts that have already been allocated can be resized using function `realloc()`.

Memory allocation - Example

pointalloc.c

```
1 int main() {
2     point_t* points = NULL;
3     int points_to_read = 0;
4     /* read number of points */
5     scanf("%d",&points_to_read);
6     points = (point_t*)
7         malloc(points_to_read*sizeof(point_t));
8     if(points == NULL) {
9         printf("could not allocate %d bytes.\n",
10             points_to_read*sizeof(point_t));
11         exit(EXIT_FAILURE);
12     }
13     /* do stuff */
14     /* free the allocated memory */
15     free(points);
16 }
```

Examples for `sizeof.c` and `realloc.c` (+ data: `file_of_ints.txt`).
Copyright © The University of Melbourne

Memory allocation - Usage

- Always use `sizeof()` to determine the size of a data type
- Always check the return value of all functions and handle errors
- Garbage collection is your responsibility. Always match a `malloc()` with the corresponding `free()`
- Read the `man` pages for all three commands
- Use `realloc()` to grow multiplicatively, not additively

Memory allocation - Exercise I

Exercise 2

Write a program that can read a $n \times m$ matrix of integers from stdin with the following format:

`n = 2`

`m = 4`

`4 812 94 24`

`42 43 31 5`

- use dynamic memory allocation to store the matrix in memory
- output the largest and smallest value in each column of the matrix

Sample data file: `matrix.txt`

Memory allocation - Exercise II

Exercise 3

Write a function `char *string_dupe(char *s)` that creates a copy of the string `s` and returns a pointer to it.

Hint: Strings (`char*`) are always terminated with a 0 byte which requires additional space.

Linked Lists I

Idea: define a `struct` type that includes a pointer to itself (rather than a pointer to, say, a string):

```
typedef struct node node_t;

struct node {
    data_t data;
    node_t *next;
};
```

Note the need for the forward declaration - the type `node_t` is declared before it is defined, the same as is done via function prototypes.

Linked Lists II

A sequence of elements of type `data_t` are threaded together in a chain of pointers.

Last item in chain has a `NULL` pointer.

If `p` is a pointer to such a chain, we can sequentially process each element in the chain.

And if `p` is a pointer to such a chain, we can `push` a new element into the front of the chain.

Linked Lists III

linkedlist.c

```
1 void
2 process_each( node_t *p) {
3     while (p) {
4         process(p->data );
5         p = p->next;
6     }
7 }
8
9 node_t*
10 push( data_t stuff , node_t *p) {
11     node_t* new = (node_t*) malloc(sizeof(node_t));
12     assert(new);
13     new->data = stuff ;
14     new->next = p;
15     return new;
16 }
```

Linked Lists IV

Similarly, if `p` is a pointer to a non-empty chain, we can `pop` the front element off the chain and discard it:

`linkedlist.c`

```
1 node_t*
2 pop(node_t *p) {
3     node_t* old;
4     assert(p);
5     old = p;
6     p = p->next;
7     free(old);
8     return p;
9 }
```

In this form, it must be used as `list = pop(list)` and not `newlist = pop(oldlist)`. (Why?)

Linked Lists V

The other option is to add new items at the `tail` of the list. Could traverse list to reach the insertion point. But better to maintain another level of abstraction that keeps pointers to the first and last item in the list; this is the purpose of the `list_t` type.

linkedlist.c

```
1 typedef struct {
2     int num_elements;
3     node_t* head;
4     node_t* tail;
5 } list_t;
6
7 data_t pop_front(list_t* list);
8 void push_front(list_t* list, data_t d);
9 void push_back(list_t* list, data_t d);
```

Linked Lists - Exercise

Exercise 4

Download [linkedlist.c](#) and implement the function
`insert_after(list_t* l, node_t* n, data_t d)` which
inserts element `d` after node `node` in the list.

Write some test code to make sure your implementation is correct.

Stacks and Queues

If insert at tail and extract from head, have a [queue](#), or a first-in first-out (FIFO) structure.

If insert and head and also extract from head, have a [stack](#), or last-in first-out (LIFO) structure.

Stacks and queues are fundamental data structures that are used in a wide range of algorithms. They allow data to be processed systematically in orders other than it was received in.

Next

Binary Search Trees



THE UNIVERSITY OF

MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

Today

LECTURE 3

Binary Search Trees (BST)

Binary Search Tree (BST)

Next step — nodes with two pointers:

```
1 typedef struct node node_t;
2
3 struct node {
4     void* data;
5     node_t* left;
6     node_t* right;
7 };
```

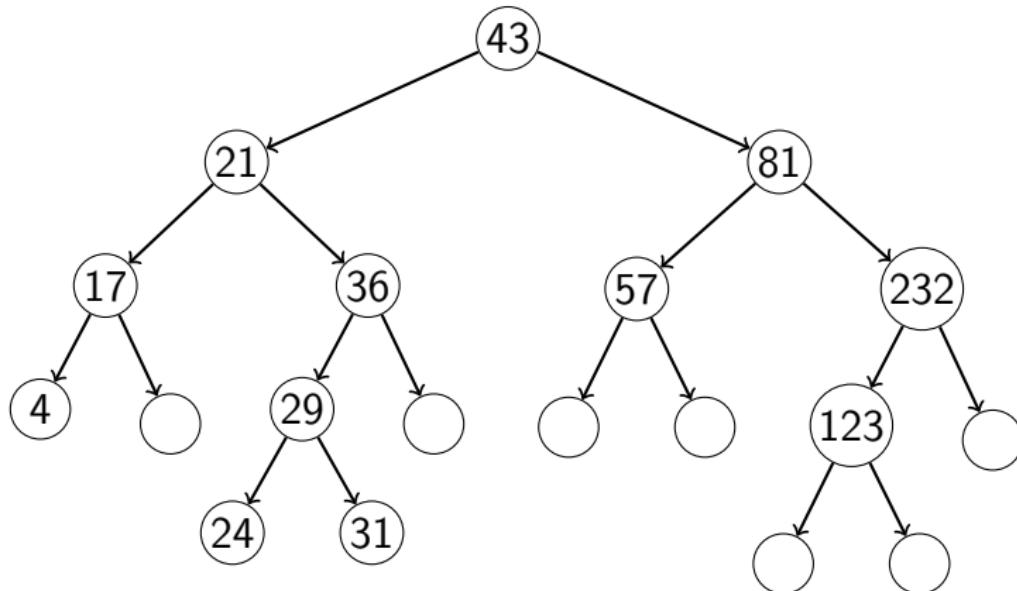
Note also (as an independent change) that `data` is stored via an anonymous pointer. This data type is now **polymorphic**.

Binary Search Trees

A node in a binary tree has two pointers which, if non-NULL, point to the left and right children of that node.

A Binary Search Tree is a binary tree in which the objects are ordered from left to right across the tree. Elements of the **left** subtree are **smaller** than current node. Elements of the **right** subtree are **larger** than the current node.

Binary Search Trees – Structure



Binary Search Trees – Properties

- Shape of the tree depends on insertion order.
- Efficient search if tree is **balanced**. A perfectly balanced BST containing n items has a **height** of $\lceil \log_2(n + 1) \rceil$.
- Height of the tree describes the length of the longest path from root to any leaf node.
- If items are inserted in increasing or decreasing order the tree becomes a linked list.

Binary Search Trees – Abstraction

To allow type-free functions, the comparison and delete functions must also be polymorphic, and are captured at the time the tree is created, rather than being passed in to every tree manipulation function.

```
1 typedef struct {
2     node_t* root;
3     int (*cmp)(void*, void*);
4     void (*del)(void*);
5 } tree_t;
```

Binary Search Trees – Abstraction

bst.c

```
1 tree_t*
2 make_tree(int fcmp(void*, void*), void fdel(void*))
3 {
4     tree_t *tree;
5     tree = malloc(sizeof(*tree));
6     assert(tree!=NULL);
7     /* initialize tree to empty */
8     tree->root = NULL;
9     /* and save the supplied function pointer */
10    tree->cmp = fcmp;
11    tree->del = fdel;
12    return tree;
13 }
```

Abstraction at its best: with these declarations, can have one tree of strings in ascending order, another of some data type in descending order.

Binary Search Trees – Searching

BST search is implemented using **recursive** functions:

bst.c

```
1 node_t*
2 tree_search(tree_t* t, node_t* node, data_t d)
3 {
4     if(node) {
5         /* go left as d < node->data */
6         if(t->cmp(d, node->data) == -1) {
7             return tree_search(t, node->left, d);
8         }
9         /* go right as d > node->data */
10        if(t->cmp(d, node->data) == 1) {
11            return tree_search(t, node->right, d);
12        }
13        /* must be equal */
14    }
15    return node;
16 }
```

Binary Search Trees – Processing

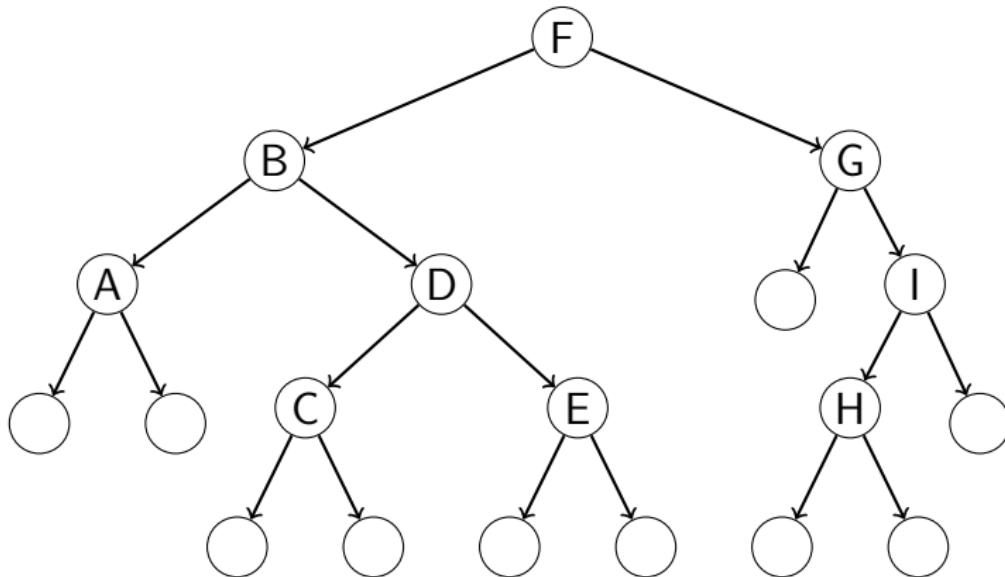
Processing the complete tree is also implemented using **recursive** functions:

bst.c

```
1 void
2 traverse_inorder(node_t* node)
3 {
4     if(node) {
5         traverse_inorder(node->left);
6
7         /* do stuff for this node */
8         process(node->data);
9
10        traverse_inorder(node->right);
11    }
12 }
```

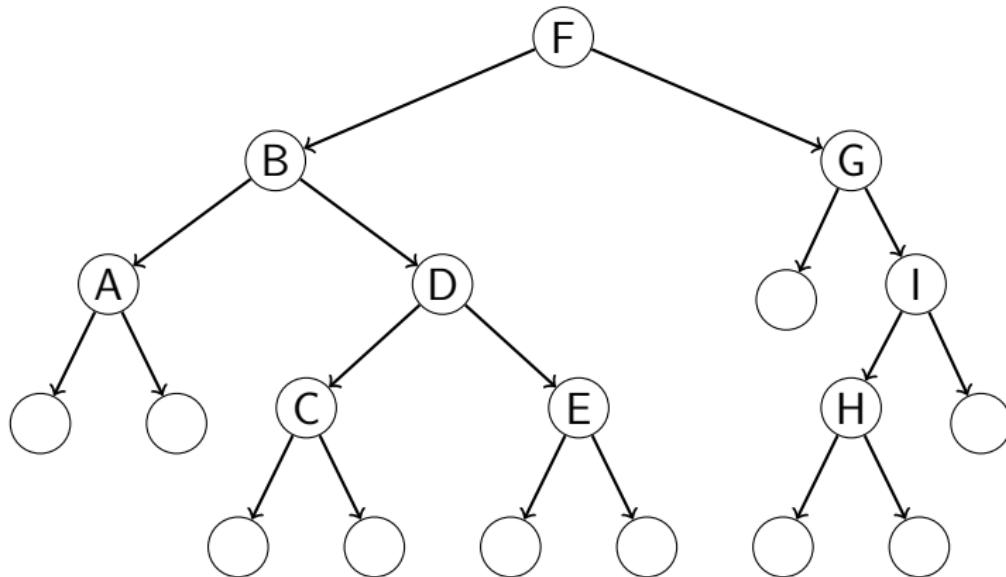
Processing can be **preorder**, **inorder** or **postorder** depending on when the data at the current node is processed relative to its subtrees.

Pre-order (NLR)



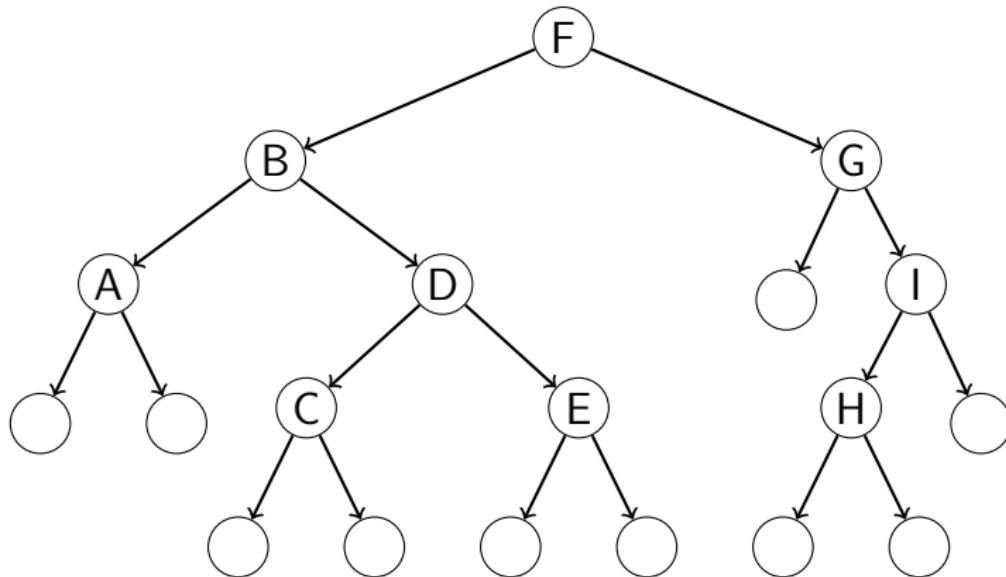
F → B → A → D → C → E → G → I → H

In-order (LNR)



A → B → C → D → E → F → G → H → I

Post-order (LRN)



A → C → E → D → B → H → I → G → F

Exercise

Exercise 7

Given a BST, how would you implement outputting the values in the tree in **descending** order?

Exercise 8

Given a BST, write a function which computes the **average depth** of objects stored in the tree.

Exercise

Exercise 9 (advanced)

If all n items to be installed in a BST are available in advance, a *balanced* tree can be constructed, in which no object has a depth greater than $\lceil \log_2(n + 1) \rceil$. In such a tree, searching is guaranteed to be fast.

Write a function that accepts as arguments an array of n objects assumed to be in sorted order, and constructs and returns a balanced binary search tree.

Dynamic Structures - Program examples

File	Content
file_io.c	Toy program explaining reading/writing content from/to a file
dynmem.c	Toy program explaining dynamic memory allocation
funcpoint.c, funcarg.c callqsort.c	Function pointer toy program Standard library sorting function using function pointers
linkedlist.c	Linked list implementation
bst.c, bst.h, bst_main.c	Binary search tree implementation (data: city.csv)

Dynamic Structures - Summary

Dynamic memory allows run-time construction of linked data structures.

Lists and trees are very powerful algorithmic techniques.

Next

Performance evaluation of algorithms and data structures.



THE UNIVERSITY OF
MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

Today

LECTURE 4

Evaluating the performance of algorithms

What is Faster? Public Transport

Let's say you want to go from campus to the MCG. You can either take a tram or train (or a combination of both). Which is faster?

Answer: Depends on walking times to tram stops, train and tram schedules or many other factors such as the direction of the city loop.

Another example: Going from the city to Richmond - train, bus or tram?

What is Faster? Algorithms

Problem: Find the k -th longest line in a text file.

Approach one: Read all the lines sort them in decreasing length order using a simple sorting algorithm such as Insertion Sort and output the k -th element.

Approach two: Keep reading lines. Keep track of the k longest lines seen so far and replace elements if necessary.

Which approach is faster? How do you know?

What is Faster? Algorithms

For small text files: Both algorithms work OK.

For large text files: None of the algorithms is efficient “enough”.

A faster sorting algorithm such as QuickSort would solve the problem much faster.

Lesson: Often it is not sufficient to write an algorithm that **can** solve a problem. If the program is to be run on a very large data set, then the **efficiency** of your program becomes an important issue.

Efficiency of Algorithms

Suppose you are given specific problem and you come up with an algorithm to solve the problem. However, if your algorithm

- Runs for two years on the fastest computer you have available, or
- Requires 10 TB of RAM

it might not be useful.

This week, we will discuss ways of estimating the resource requirements of a program and how to compare the resource requirements of programs solving a problem **without ever implementing them**.

Efficiency of Algorithms - Resources

Consider the following example of two algorithms **A** and **B** which require the following resources to solve an instance of a specific problem:

Resource	Algorithm	
	A	B
Peak Memory Usage [MB]	123	2923
Bandwidth [MB]	55	50
Run Time [sec]	512	43
Disk Storage Space [MB]	5	500
Cloud Computing Cost [\$]	5	10

What are the important metrics? How would you measure the resource requirements of an algorithm or data structure?

Efficiency of Algorithms - Resources

Consider the following example of two algorithms **A** and **B** which require the following resources to solve an instance of a specific problem:

Resource	Algorithm	
	A	B
Peak Memory Usage [MB]	123	2923
Bandwidth [MB]	55	50
Run Time [sec]	512	43
Disk Storage Space [MB]	5	500
Cloud Computing Cost [\$]	5	10

What are the important metrics? How would you measure the resource requirements of an algorithm or data structure?

Theoretical Analysis of Runtime efficiency

We will focus on the run time efficiency of an algorithm. Specifically, we are interested in the **basic operation** of an algorithm. The basic operation of an algorithm is the main contributor towards the runtime.

Examples:

- Sorting and Searching: Item comparisons
- Matrix multiplication: Floating point multiplication

How often is the basic operation executed? Specifically, we are interested in the **rate of growth** depending on the input size, not the exact running time on any input.

Theoretical Analysis of Runtime efficiency

Thus, we formally approximate the runtime of an algorithm as

$$T(n) = c_{op} \times C(n)$$

where

- n is the input size
- $T(n)$ is the estimated runtime of the algorithm
- c_{op} is the cost of executing the basic operation once on a specific machine
- $C(n)$ is the number of times the basic operation is executed

Describing Runtime efficiency

Recall that the performance of a binary search tree depends on the insertion sequence of the elements. Thus, we differentiate between

- Worst Case – Given an input of n items, what is the maximum cost for any possible input?
- Average Case – Given an input of n items, what is the average cost across all possible inputs?
- Best Case – Given an input of n items, what is the minimum cost for any possible input?

Example: Searching a linked list

Searching for item x in a linked list of length n :

- Basic Operation?

Example: Searching a linked list

Searching for item x in a linked list of length n :

- Basic Operation? Item comparison
- Best Case?

Example: Searching a linked list

Searching for item x in a linked list of length n :

- Basic Operation? Item comparison
- Best Case? $C(n) = 1$
- Worst Case?

Example: Searching a linked list

Searching for item x in a linked list of length n :

- Basic Operation? Item comparison
- Best Case? $C(n) = 1$
- Worst Case? $C(n) = n$
- Average Case?

Example: Searching a linked list

Searching for item x in a linked list of length n :

- Basic Operation? Item comparison
- Best Case? $C(n) = 1$
- Worst Case? $C(n) = n$
- Average Case? $C(n) = n/2$? Can be tricky!

Defining efficiency

While it might be tempting to hope for the best, it is more professional (and much safer) to instead plan for the **worst**.

If the program requires n steps on an input of size n , it is **linear**. If it takes one second when $n = 1,000$, it will take around two seconds if n is increased to 2,000.

It is the **rate of growth** that is important, not the exact running time on any input.

A different algorithm that requires $2n$ steps is still linear; as is a third method that requires $15n + 27\sqrt{n} - e \log n$ steps.

Formal Framework: Asymptotic Notation

To capture this idea, we define **sets of functions** that are all **asymptotically equivalent** in terms of their eventual long term growth rate:

$$f(n) \in O(g(n))$$

if and only if

$$\exists n_0, c > 0 : \forall n > n_0, f(n) \leq c \cdot g(n).$$

This is a complex definition!

Asymptotic Notation Explained

In words:

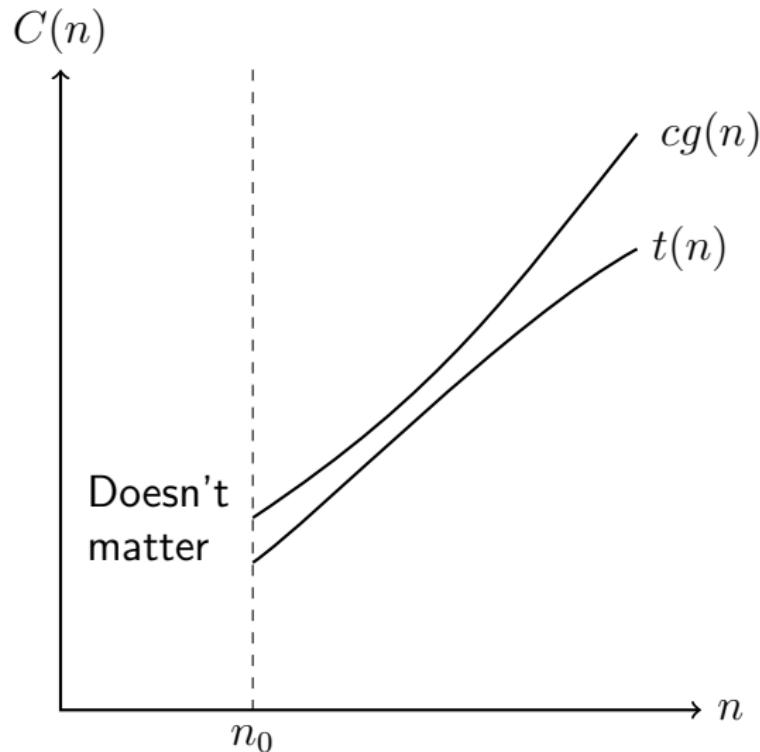
*$f(n)$ is a function that is **order $g(n)$** when a positive threshold n_0 and a positive constant c can be identified such that, for every n larger than n_0 , $f(n)$ is bounded above by c times $g(n)$.*

Example 1: Take $f_1(n) = 2n$. Then $f_1(n) \in O(n)$.

Demonstration: take $n_0 = 1$ and $c = 2$.

Example 2: Take $f_2(n) = 15n + 27\sqrt{n} - e \log n$. Then $f_2(n) \in O(n)$. Demonstration: take $n_0 = 1$ and $c = 42$.

Asymptotic Notation in one Picture!



Asymptotic Notation - More Examples

Example 3: Take $f_3(n) = f_2(n) \times f_1(n)$. Then $f_3(n) \in O(n^2)$. (And also $\in O(n^2 \log n)$, $O(n^3)$, and so on). Demonstration: take $n_0 = 1$ and $c = 84$.

Example 4: Take $f_4(n) = 5123 \times f_1(n)$. Then $f_4(n) \in O(n)$. (And also $\in O(n \log n)$, $O(n\sqrt{n})$, $O(n^2)$, and so on).

Asymptotic Notation - Continued

Note that $O(g(n))$ is a **set** of functions, including all functions that have the same or smaller growth rate.

Important: Given a function $f(n)$ to be categorised, it is usual to make use of the simplest such $g(n)$ function, by dropping constants and secondary terms; and also to choose $g(n)$ so that the smallest set $O(g(n))$ is generated.

So, while it is correct that $2n \log_2 n \in O(5n^2 - 3)$, it is usual to keep it simple, and say that $2n \log 2n \in O(n \log n)$.

Asymptotic Notation - Why

The “order” notation provides a tremendously useful abstraction that lets algorithms be compared.

If $f(n) \in O(g(n))$, and $g(n) \notin O(f(n))$, then an algorithm that requires $f(n)$ steps will, for large enough inputs, be faster than an algorithm that takes $g(n)$ steps.

Example: Algorithm A is $f(n) = n \log n$ and Algorithm B is $g(n) = n^2$ then $n \log n \in O(n^2)$ but $n^2 \notin O(n \log n)$. Thus algorithm A will be faster than B for large n .

Asymptotic Notation - Common Sets

Most algorithms can be classified into one of the following sets:

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n)$$

$$O(n \log n) < O(n\sqrt{n}) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

Time complexity and **Space complexity**. E.g., the time complexity of search in a balanced binary search tree is $O(\log n)$; space complexity?

Asymptotic Notation: What does it mean in practice?

n	$\log_2(n)$	n	$n \log_2 n$	n^2	n^3
10	4	10	34	10^2	10^3
100	7	100	665	10^4	10^6
1000	10	1000	9966	10^6	10^9
10k	14	10k	132k	10^8	10^{12}
100k	17	100k	1.6M	10^{10}	10^{15}
1M	20	1M	2×10^7	10^{12}	10^{18}

For small inputs the efficiency of an algorithm is not important.

Fast algorithms can solve problems on a slow machine, however a fast machine often does not help if you are using a slow algorithm.

Asymptotic Notation: What does it mean in practice? II

A current processor (Intel i7 3Ghz) can execute 240 billion (2.4×10^{11}) instructions per second.

Under this assumption, how would it take to solve a problem with an algorithm in a certain efficiency class?

n	$\log_2(n)$	n	$n \log_2 n$	n^2	n^3
100k	instant	instant	instant	instant	seconds
1M	instant	instant	instant	seconds	months
10M	instant	instant	instant	minutes	centuries
1B	instant	instant	seconds	days	never

Asymptotic Notation - Exercises

Exercise 10

Order the following functions according to their order of growth (from the lowest to the highest):

$$(n - 4)!, 2 \log_2(n + 53)^6, 2^{2^n}, 0.032n^4 + 2n^3 + 7, \log_e^2 n, \sqrt[3]{n}, 3^n$$

Exercise 11

Indicate whether the first function of each of the following pairs has a smaller, same or larger order of growth (to within a constant multiple) than the second function.

- a) $100n^2$ and $0.01n^3$
- b) $\log_2 n$ and $\log_e n$
- c) $(n - 1)!$ and $n!$

From code to asymptotic notation

- Determine the basic operation of the algorithm (usually the one in the inner most loop)
- As a function of the input size n , figure out how often it will be executed
- Sometimes the input size is defined by two or more variables. Example: a Matrix of size $n \times m$. In this case, the order of the algorithm is described by two variables. For example: $O(n \log m)$

From code to asymptotic notation: Example

```
1 /* determine if all elements in an
2     unsorted integer array are unique.
3     return 1 if true and 0 otherwise */
4 int check_if_unique(int* A, int n) {
5     for(int i=0;i<n-2;i++) {
6         for(int j=i+1;j<n-1;j++) {
7             if( A[ i ] == A[ j ] ) {
8                 return 0;
9             }
10        }
11    }
12    return 1;
13 }
```

From code to asymptotic notation: Example II

- Basic Operation?

From code to asymptotic notation: Example II

- Basic Operation? Comparison in line 7.
- Input Size?

From code to asymptotic notation: Example II

- Basic Operation? Comparison in line 7.
- Input Size? Size of the array n
- How often do we compare?

From code to asymptotic notation: Example II

- Basic Operation? Comparison in line 7.
- Input Size? Size of the array n
- How often do we compare? $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1.$

From code to asymptotic notation: Example II

- Basic Operation? Comparison in line 7.
- Input Size? Size of the array n
- How often do we compare? $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1.$
- Asymptotic Notation?

From code to asymptotic notation: Example II

- Basic Operation? Comparison in line 7.
- Input Size? Size of the array n
- How often do we compare? $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$.
- Asymptotic Notation?

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - 1 - i)$$

From code to asymptotic notation: Example II

- Basic Operation? Comparison in line 7.
- Input Size? Size of the array n
- How often do we compare? $C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$.
- Asymptotic Notation?

$$\begin{aligned}C(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \\&\sum_{i=0}^{n-2} (n-1) \sum_{i=0}^{n-2} i = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in O(n^2)\end{aligned}$$

From code to asymptotic notation: Exercise

Exercise 12

Determine the efficiency class of the following algorithm:

```
int ABC(int n) {  
    int a = 0;  
    for(int i=0;i<n;i++) {  
        for(int j=0;j<n;j++) {  
            a = a + sqrt(i+j+1);  
        }  
    }  
    return a;  
}
```

Theoretical Analysis: Overview

We can evaluate the performance of an algorithm without implementing it by determining its efficiency class.

Depending on the input size, efficiency is determined by the number of time the basic operation of an algorithm is executed.

Order of growth is more important than describing runtime exactly.

Next

Practical analysis of algorithm performance



THE UNIVERSITY OF

MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

Today

Empirical performance evaluation of
algorithms

O-Notation recap

$$f(n) = 12 \log_2 n + 51232n * \log_e n + 0.5n^2 + 34234242n\sqrt{5n}$$

$f(n)$ is in which sets?

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n)$$

$$O(n \log n) < O(n\sqrt{n}) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

Abstract Data Types

An **abstract data type** is a model which describes a data type by the **functionality** it provides. For example:

Data Type	Operations
Stack (LIFO)	push, pop
Queue (FIFO)	enqueue, dequeue
Dictionary / Set	insert(elem), search(elem)
Map	insert(key,value), lookup(key)

For example, in Python, C++ or Java you use data types such as **Maps**, **Sets**, **Queues** without knowing what kind of data structure is used “internally”.

Dictionary abstract data type

If we have an algorithm that requires the operations `insert()` and `search()`, we now have several possible structures:

- Unsorted / Sorted Arrays
- Unsorted / Sorted Linked Lists
- Binary Search Trees

How do we evaluate which data structure best fits the requirements of the problem we are trying to solve?

Empirical Evaluation

Idea

- Implement all relevant data structures and algorithms
- Gather several instances of your problem
- Run controlled experiments to determine experimentally which data structure or algorithm best suits the problem

What should we measure?

Empirical Evaluation - Measuring RAM Usage

Approach 1

The majority of memory used by a program is allocated using `malloc` and released using `free`

Idea: Create custom functions which encapsulate `malloc` and `free` to keep track of memory used by a program.

Approach 2

Use facilities provided by the operating system to measure peak memory usage

Toy program: `memusage.c`

Measuring RAM Usage - Exercise

Exercise 13

Copy the relevant functions from the `memusage.c` toy program and combine it with the `linkedlist.c` program. Keep inserting new dummy elements into a linked list. Output the current memory usage after each inserted item.

Measure the maximum amount of memory your program is allowed to consume before it fails. Try running the program on dimefox where you are only allowed to consume a limited amount of RAM.

Measuring Runtime Performance

Runtime performance may be measured for

- Whole programs
- Algorithms
- Functions
- CPU instructions or resource requests

Examples: Time to sort a set of strings, Time to perform a HTTP request, Time to compute the square root of a number.

Measuring Program Execution time

```
1 > time sort strings.txt > sorted_strings.txt
2 real      0m5.085s
3 user      0m4.014s
4 sys       0m0.071s
```

- Prefix executable by `time` command
- Measures reading input, program execution, writing output
- Breakdown into **real**, **user** and **system** time.

Measuring Function Execution time I

gettimeofday.c

```
1 #include <sys/time.h>
2 #include <stdio.h>
3 #include <unistd.h>
4 int main()
5 {
6     struct timeval start;
7     struct timeval stop;
8     gettimeofday(&start, NULL);
9     sleep(5); // call algorithm
10    gettimeofday(&stop, NULL);
11    double elapsed_ms =
12        (stop.tv_sec - start.tv_sec) * 1000.0;
13    elapsed_ms +=
14        (stop.tv_usec - start.tv_usec) / 1000.0;
15    printf("elapsed time = %.2f ms\n", elapsed_ms);
16 }
```

Measuring Function Execution time II

- Measure time of parts of a program using `gettimeofday`
- Other high resolution timer measurement functions exist but are platform specific
- Measures **real** time in seconds and microseconds
- To measure more precisely execute functions multiple times and report mean and median running times

Question: What can influence the runtime of an algorithm?

Measuring Runtime - Exercise

Exercise 14

Write a program using `gettimeofday` which measures the runtime of the following functions accurately:

- `sqrt()`
- `pow()`
- `qsort()` using an array of lengths $1M$, $10M$ and $100M$ containing random integers

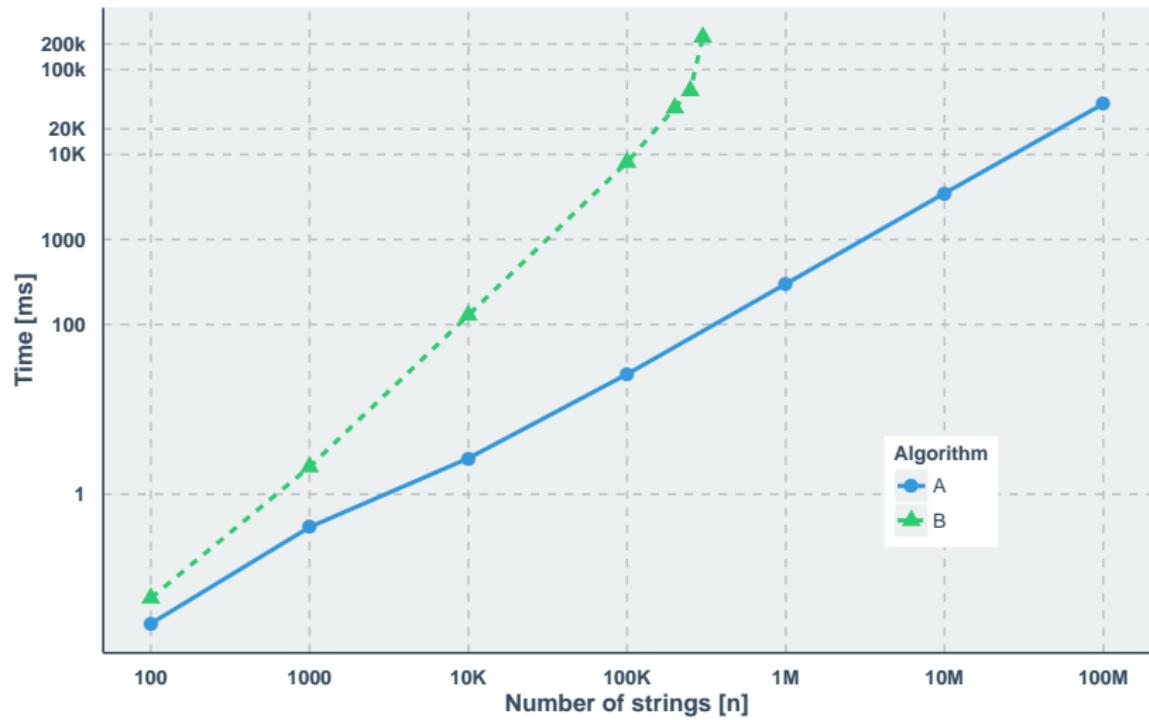
Some of these functions execute in nanoseconds so it might be necessary to report average runtime over many measurements.
Run your program multiple times and compare the results.

Sample empirical evaluation

Task: Sort n strings lexicographically using two standard sorting algorithms **A** and **B**:

n	Algorithm			
	A		B	
	Time	Memory	Time	Memory
100	0.03ms	4.7MB	0.06ms	4.7MB
1000	0.41ms	4.9MB	2.10ms	4.9MB
10k	2.65ms	6.9MB	128ms	6.9MB
100k	26ms	24MB	8s	24MB
200k	40ms	24MB	35s	24MB
300k	120ms	24MB	240s	24MB
1M	300ms	130MB		130MB
10M	3.5s	1GB		1GB
100M	39s	10.6GB		10.6GB

Graphical Evaluation of Algorithm Performance



Overview experimental evaluation

- Resource consumption of algorithms and data structures can be measured empirically
- Many tools exist to measure different kind of resources (time, memory usage, CPU instructions)
- While most algorithms are deterministic, the resources they consume can depend on many factors such as input types, input size or the computing environment
- Careful experimentation and evaluation of resources is required to draw conclusions about the performance of an algorithm

Next

Searching/Sorting algorithms



THE UNIVERSITY OF

MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

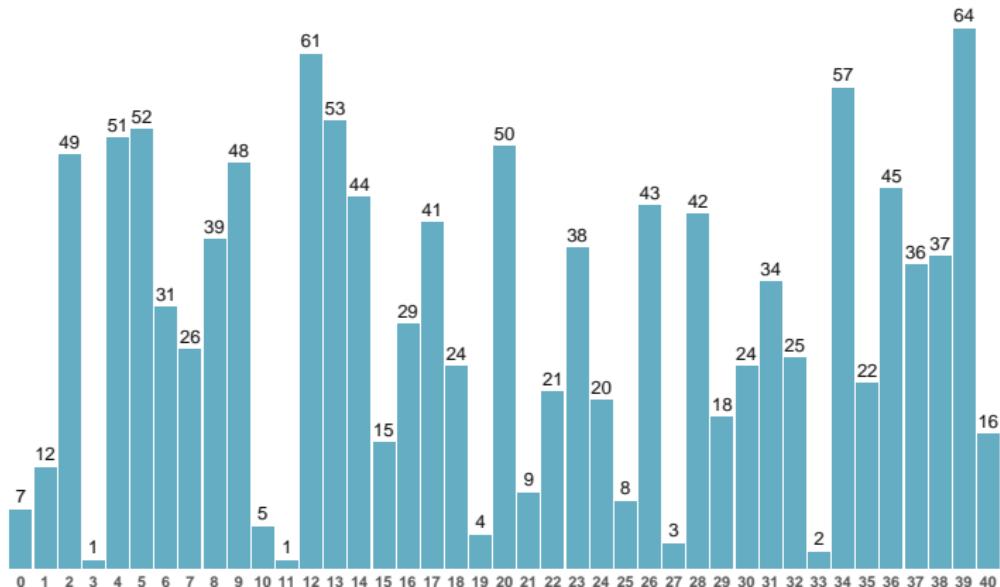
Today

Arrays and Searching/Sorting Algorithms

Searching in Arrays

Task: Determine if item x occurs in array $A[0, n - 1]$.

Search for $x=38$



Unsorted: In the worst case, $O(n)$ comparisons
Copyright © The University of Melbourne

Searching in Arrays

If array is **sorted**, can exit loop early:

```
1 // return item pos if found
2 // or -1 if not found
3 int i = 0;
4 while( i < n && A[ i ] < x )
5     i = i + 1
6 if( i == n || A[ i ] > x )
7     return -1;
8 return i;
```

Worst-case execution time is still linear.

Binary search

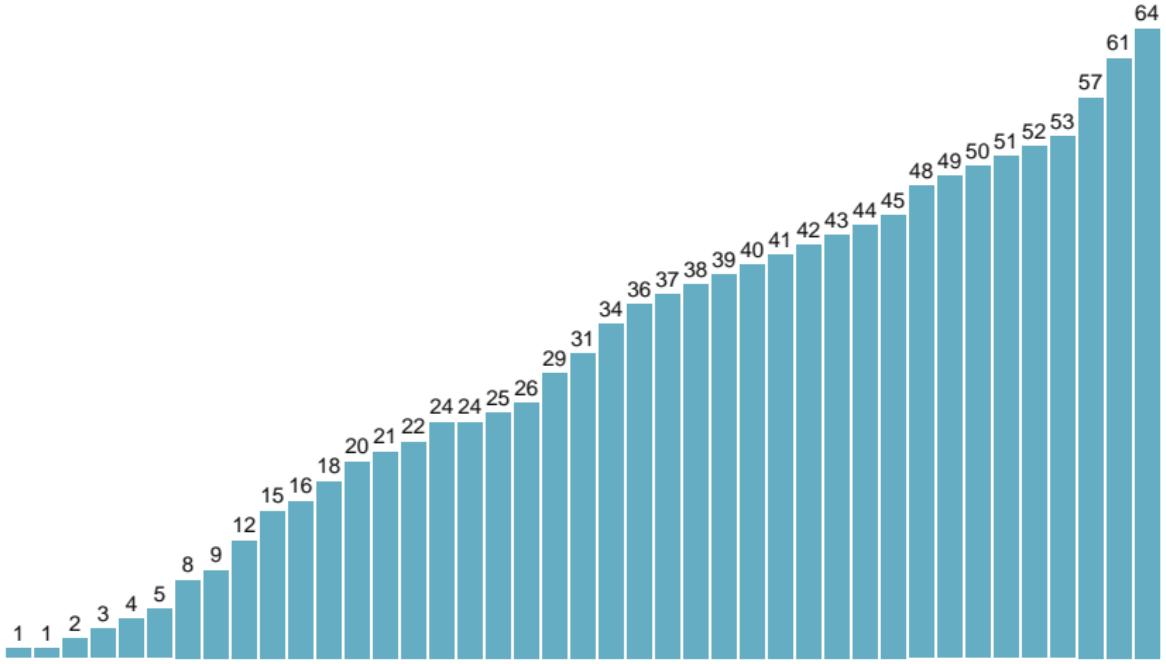
Idea: If array is sorted, search by repeated range halving is better.

Procedure:

- Compare item x with element in the middle element $A[m] = M$ of the current range $A[low, high]$. Here $m = (low + high)/2$.
- If $x < M$ we have to refine the range to $low = low, high = m - 1$.
- If $x > M$ we have to refine the range to $low = m + 1, high = high$.
- If $x == M$ we found the item

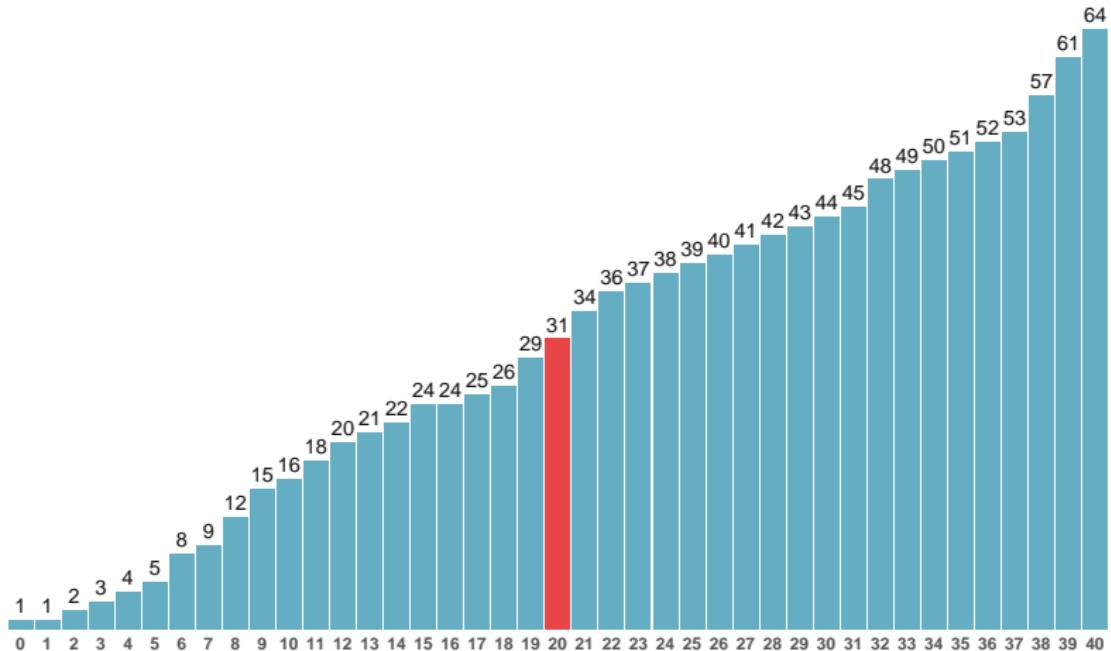
Binary search Example

Search for $x=38$



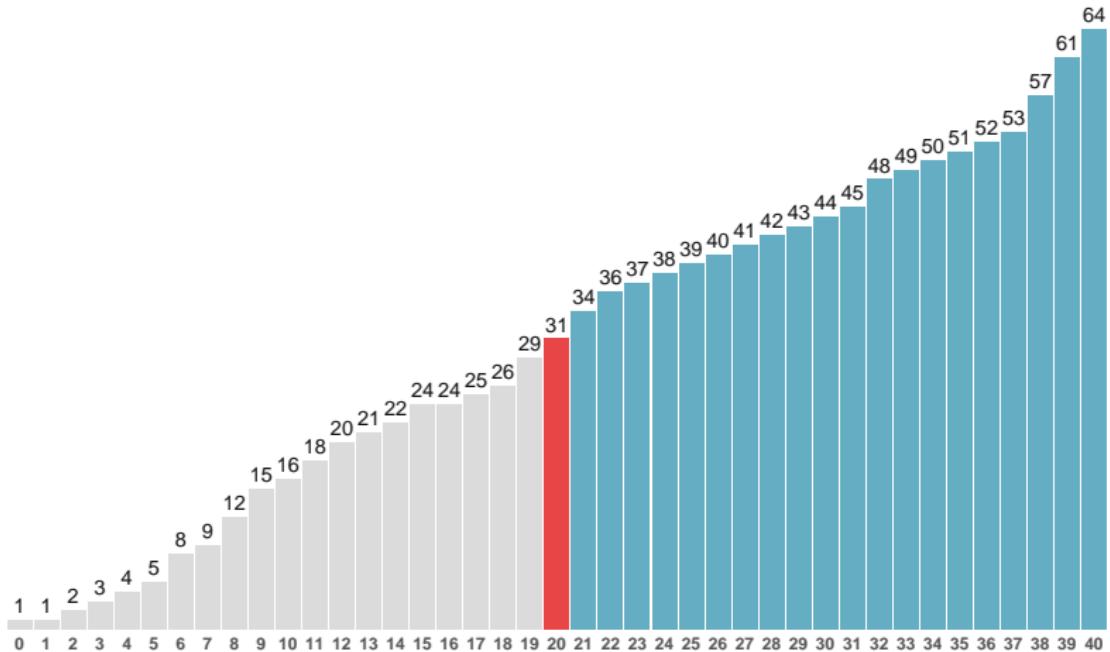
Binary search Example

Search for $x=38$



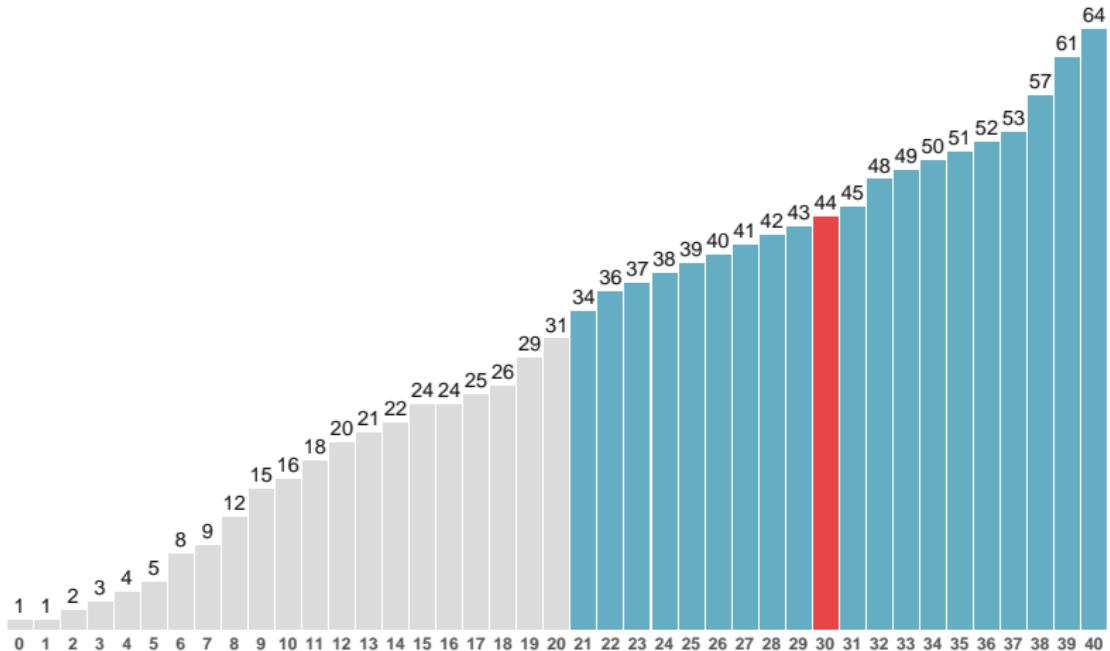
Binary search Example

Search for $x=38$



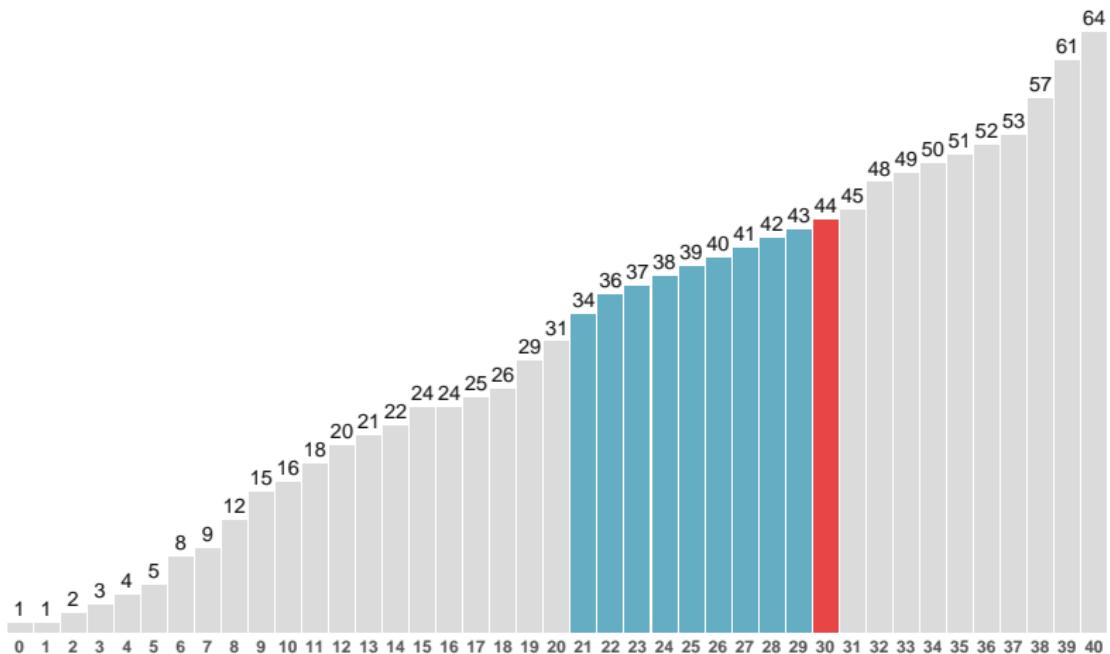
Binary search Example

Search for $x=38$



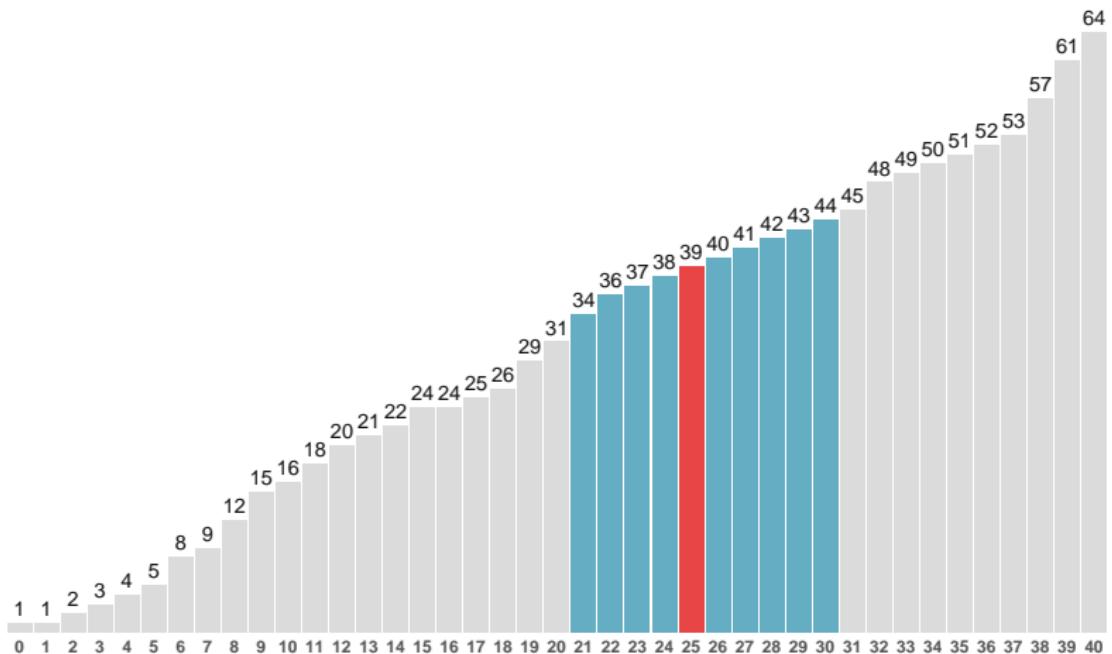
Binary search Example

Search for $x=38$



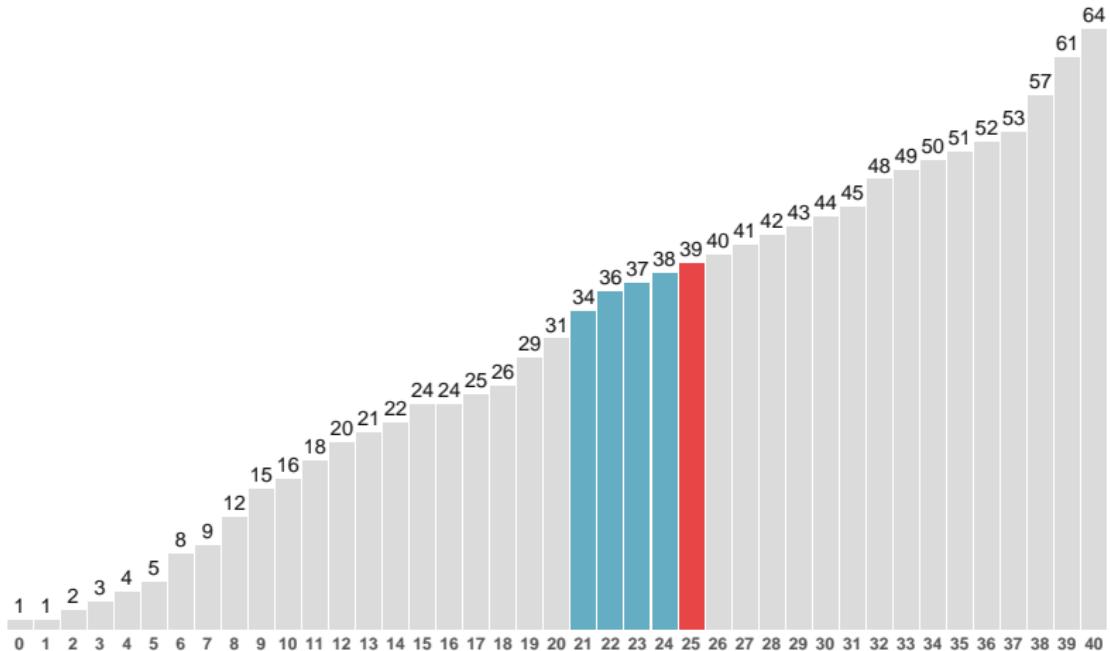
Binary search Example

Search for $x=38$



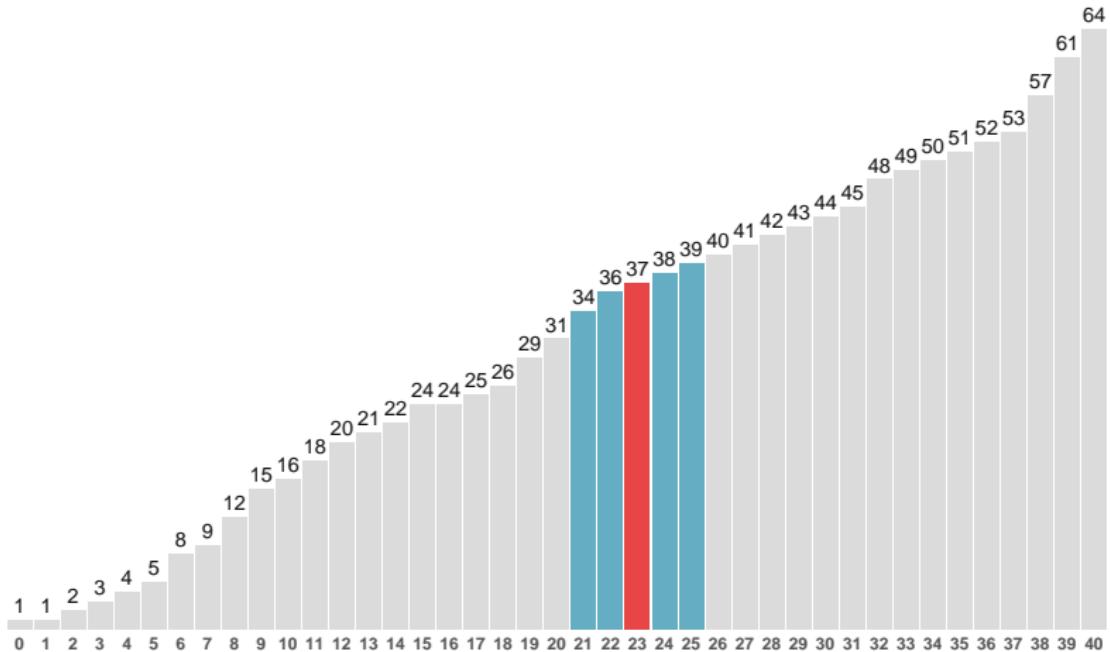
Binary search Example

Search for $x=38$



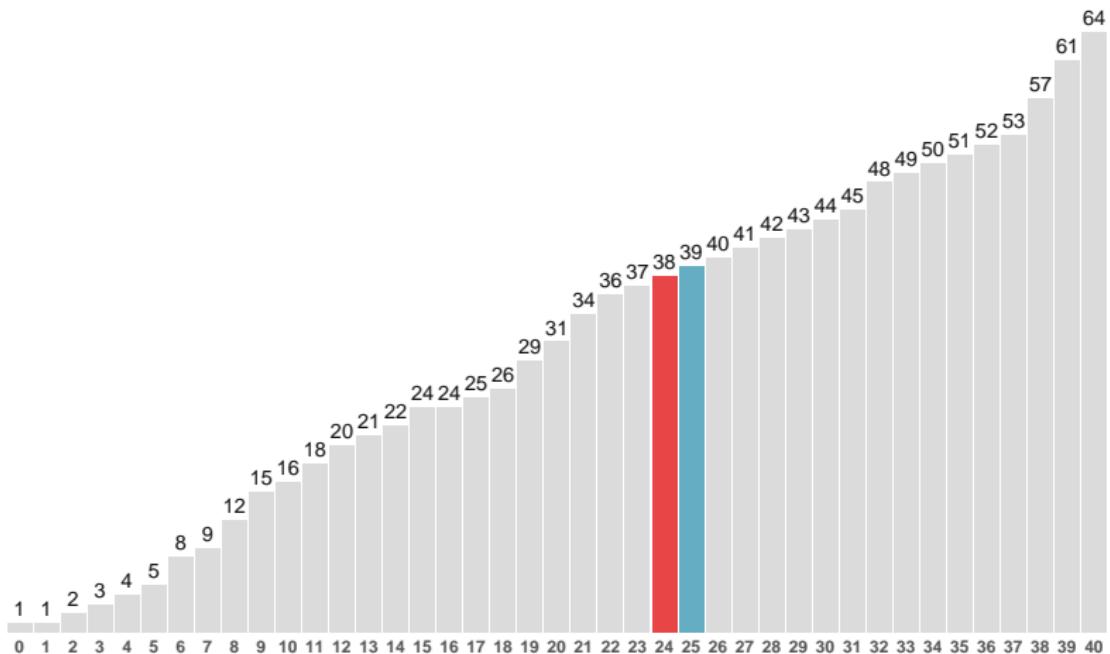
Binary search Example

Search for $x=38$



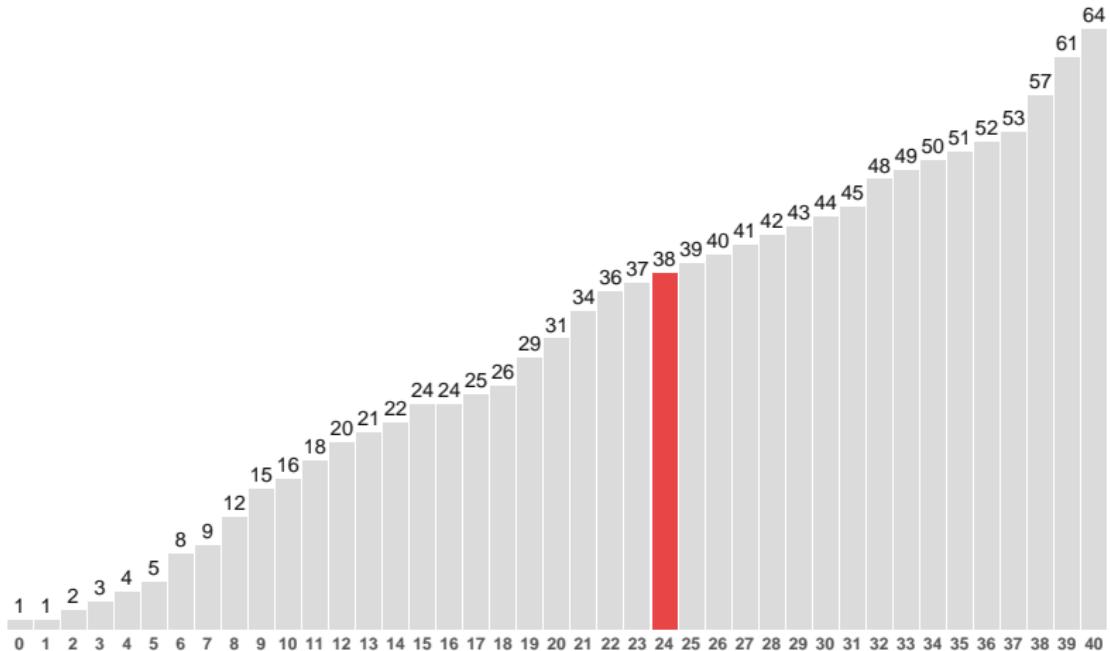
Binary search Example

Search for $x=38$



Binary search Example

Search for $x=38$



Binary search - Pseudocode Recursive

binary_search.c

```
1 int binary_search_r(int* A, int lo, int hi, int x)
2 {
3     if (lo >= hi) {
4         return -1;
5     }
6     int mid = (lo + hi) / 2;
7     int M = A[mid];
8     if (x < M) {
9         return binary_search_r(A, lo, mid, x);
10    } else if (x > M) {
11        return binary_search_r(A, mid + 1, hi, x);
12    } else {
13        return mid;
14    }
15 }
```

Binary search - Pseudocode Non-Recursive

binary_search.c

```
1 int binary_search(int* A, int lo, int hi, int x)
2 {
3     while (lo < hi) {
4         int m = (lo + hi) / 2;
5         if (x < A[m]) {
6             hi = m;
7         } else if (x > A[m]) {
8             lo = m + 1;
9         } else {
10            return m;
11        }
12    }
13    return -1;
14 }
```

Binary search - Function pointer comparison

```
1 int bs_funccmp(void* A, int lo, int hi, void* key,
2                 int (*cmp)(const void*a, const void*b))
3 {
4     if (lo >= hi) {
5         return -1;
6     }
7     int mid = (lo + hi) / 2;
8     void* M = A[mid];
9     int outcome = cmp(key, M);
10    if (outcome < 0) {
11        return bs_funccmp(A, lo, mid, x);
12    } else if (outcome > 0) {
13        return bs_funccmp(A, mid + 1, hi, x);
14    } else {
15        return mid;
16    }
17 }
```

Comparison Functions - A standard C paradigm I

The function `cmp` is a standard C paradigm.

It is usually passed in to a sorting or searching function as a **function argument**.

It compares (via pointers to underlying objects) two elements, and returns:

- –ve, if first item should come prior to second
- 0, if two items can be considered to be equal
- +ve, if first item should come after second.

Comparison Functions - A standard C paradigm II

From the man page of `qsort`:

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

From the man page of `strcmp`:

The `strcmp()` function compares the two strings s_1 and s_2 . It returns an integer less than, equal to, or greater than zero if s_1 is found, respectively, to be less than, to match, or be greater than s_2 .

Binary search - Analysis

How many basic operations does binary search require?

Based on the recursive pseudo code above, the time taken is bounded above by $C(n)$, where

$$C(n) = \begin{cases} 1 & \text{if } n \leq 1; \\ 1 + C(\lfloor n/2 \rfloor) & \text{if } n > 1. \end{cases}$$

The exact solution of this one is

$$C(n) = 1 + \lfloor \log_2 n \rfloor \in O(\log n)$$

Intuition: How often do I have to half a range of size n until it is of size 1? $\log_2 n$ times

Binary search - Asymptotic Growth

n	$\log_2 n$
10	3.3
100	6.6
1,000	9.9
10,000	13.3
100,000	16.6
1,000,000	19.9
10,000,000	23.3
100,000,000	26.6
1,000,000,000	29.9
10,000,000,000	33.2
100,000,000,000	36.5
1,000,000,000,000	39.9
10,000,000,000,000	43.2

Binary search - Exercise

Exercise 14

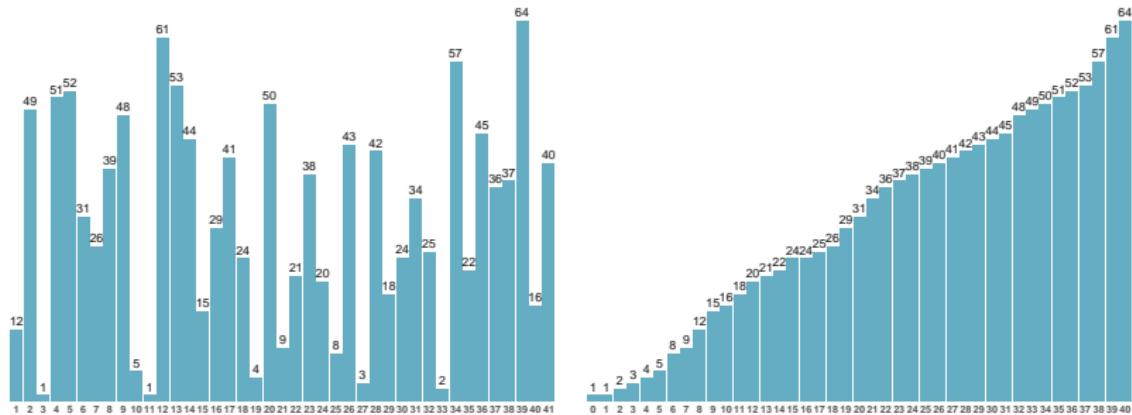
Modify the `binary_search.c` program to measures the runtime and number of basic operations executed by the binary search algorithm for different input sizes.

Exercise 15

Is it possible to apply binary search to a linked list? Discuss!

Sorting

How did the array get sorted? Sorting can increase the efficiency of many applications such as duplicate detection or searching.



Review: Insertion Sort I

One simple algorithm is called **insertion sort**:

- One part of the array is in sorted order (initially $A[0]$)
- Increase the size of the sorted sub array by **inserting** items in the correct position
- Insertion in the array is done by swapping elements into the correct position
- Every iteration, sorted sub array size increases by one

Review: Insertion Sort II

When applied to the array $\{22, 14, 17, 42, 27, 28, 23\}$:

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Initially	22	14	17	42	27	28	23
After i=0	22	14	17	42	27	28	23
After i=1	14	22	17	42	27	28	23
After i=2	14	17	22	42	27	28	23
After i=3	14	17	22	42	27	28	23
After i=4	14	17	22	27	42	28	23
After i=5	14	17	22	27	28	42	23
After i=6	14	17	22	23	27	28	42

Review: Insertion Sort III

```
1 void insertion_sort(int* a, int n)
2 {
3     for (size_t i = 1; i < n; ++i) {
4         int tmp = a[i];
5         size_t j = i;
6         while (j > 0 && tmp < a[j - 1]) {
7             a[j] = a[j - 1];
8             --j;
9         }
10        a[j] = tmp;
11    }
12 }
```

Review: Insertion Sort IV

Insertion sort is a relatively bad algorithm – inserting $A[i]$ might require i swaps (what input causes this?), making the total number of swaps as large as

$$\sum_{i=0}^{n-1} i = \frac{n(n - 1)}{2} \approx n^2/2 \in O(n^2).$$

The **worst case** (and average case) behavior of insertion sort is $O(n^2)$ in the number of items being sorted.

Exercise 16

Where does the summation on the previous slide come from?
What is the worst case input for Insertion Sort? Discuss!

Better Sorting?

Does sorting need to take quadratic time?

Luckily there are much better sorting algorithms. If this was not the case, solving many of the problems we encounter on a daily basis would not be possible!

For the rest of this week, we will look at two popular, efficient sorting algorithms, [Merge Sort](#) and [Quick Sort](#).

Merge Sort - Idea

- Simple **divide-and-conquer** approach
- Split array of size n into two arrays of size $s = n/2$
- Sort arrays of size $n/2$
- Merge sorted arrays back together to get sorted array of size n
- Keep splitting recursively until $s = 1$

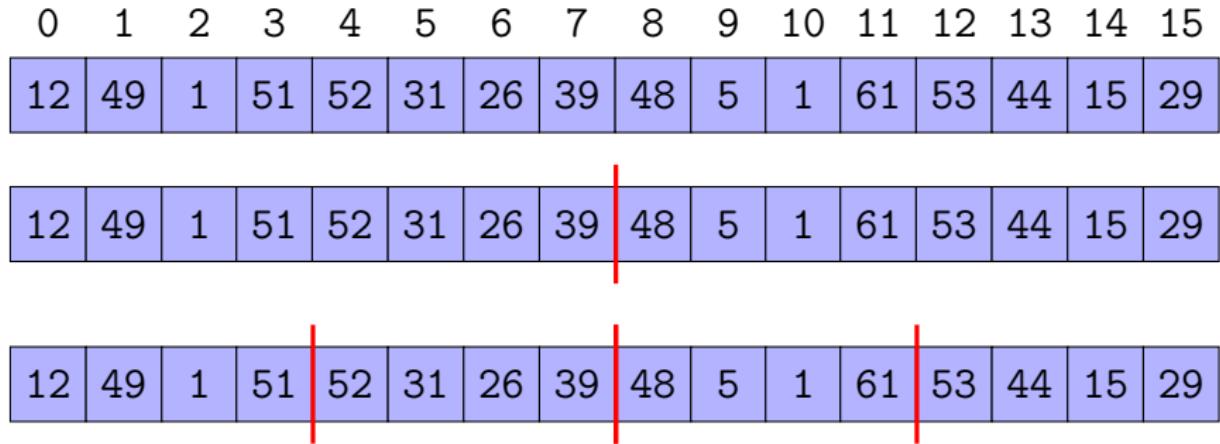
Merge Sort - Split / Divide

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29

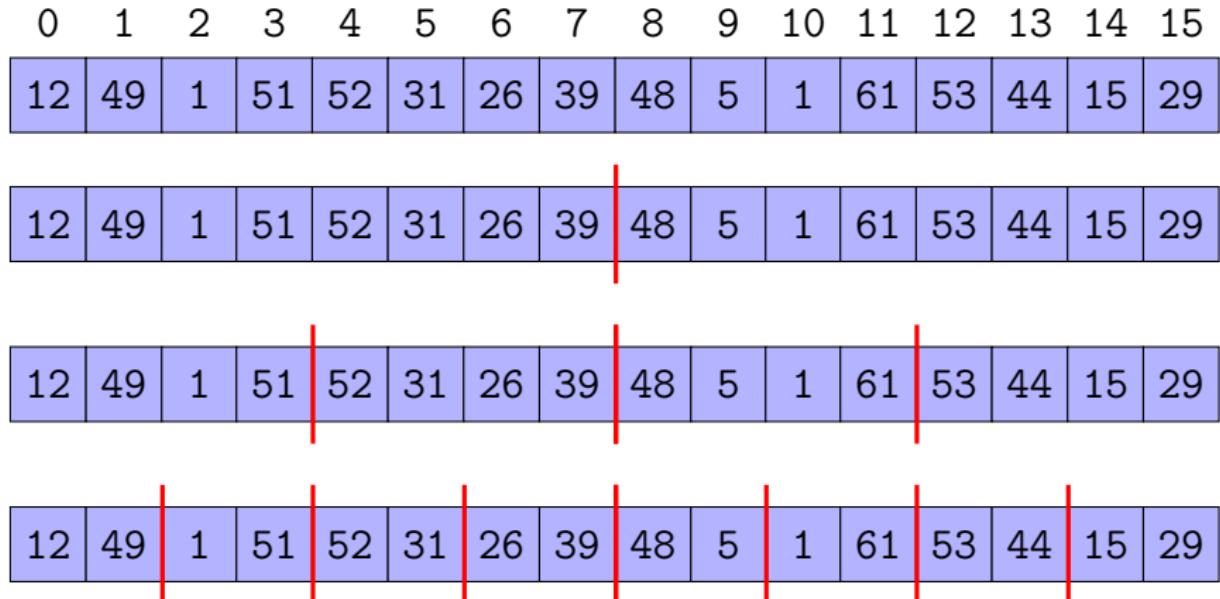
Merge Sort - Split / Divide

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29

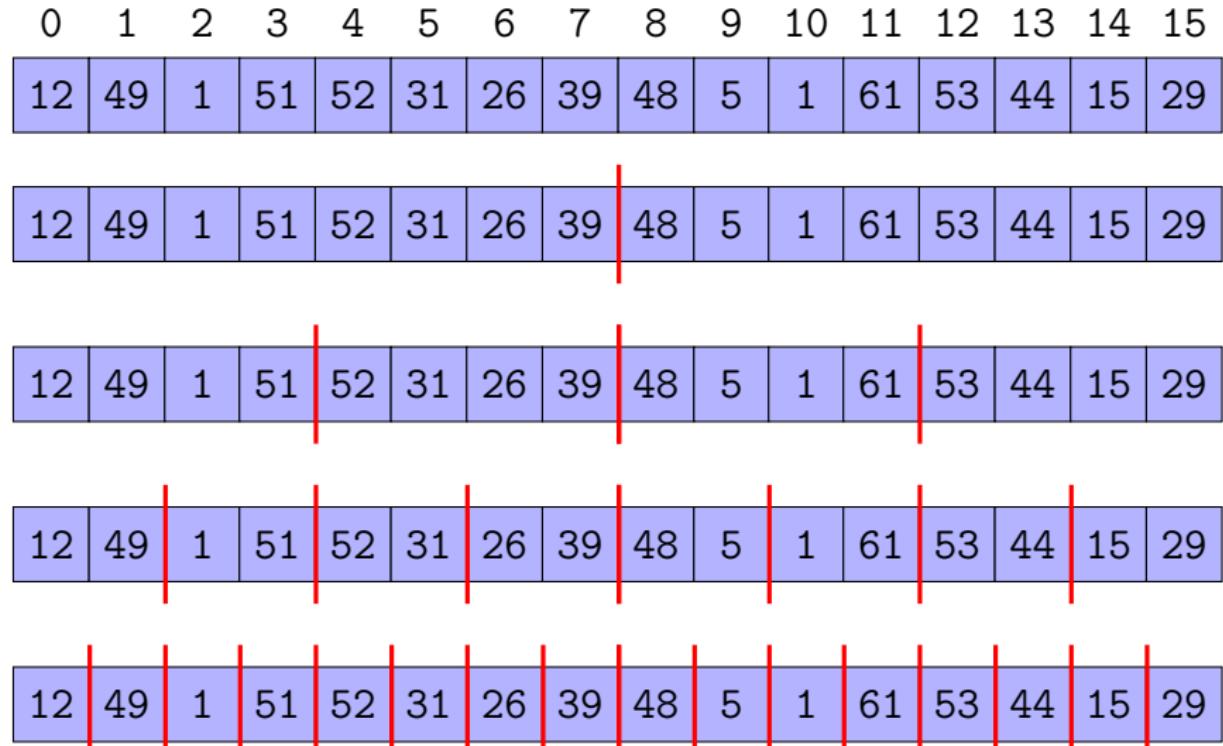
Merge Sort - Split / Divide



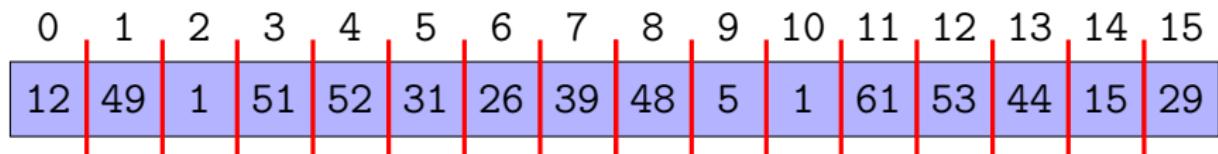
Merge Sort - Split / Divide



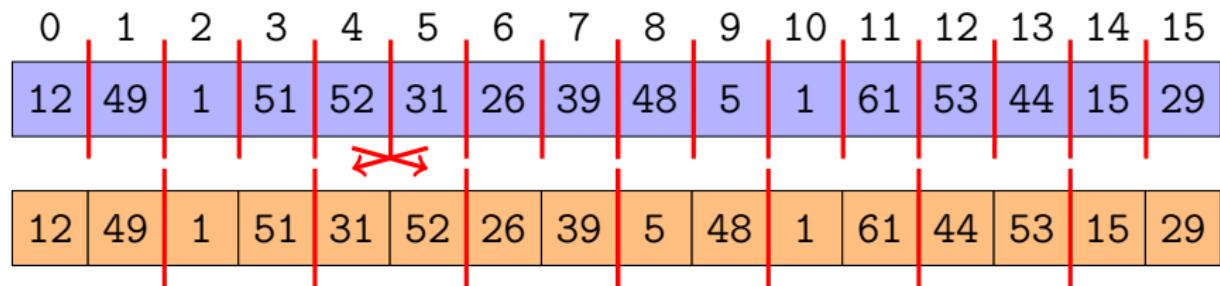
Merge Sort - Split / Divide



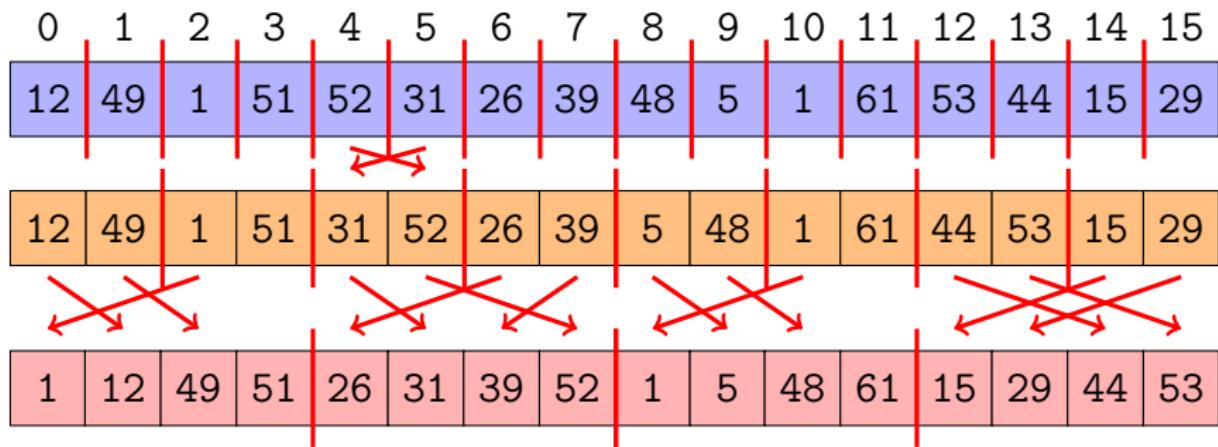
Merge Sort - Merge / Conquer



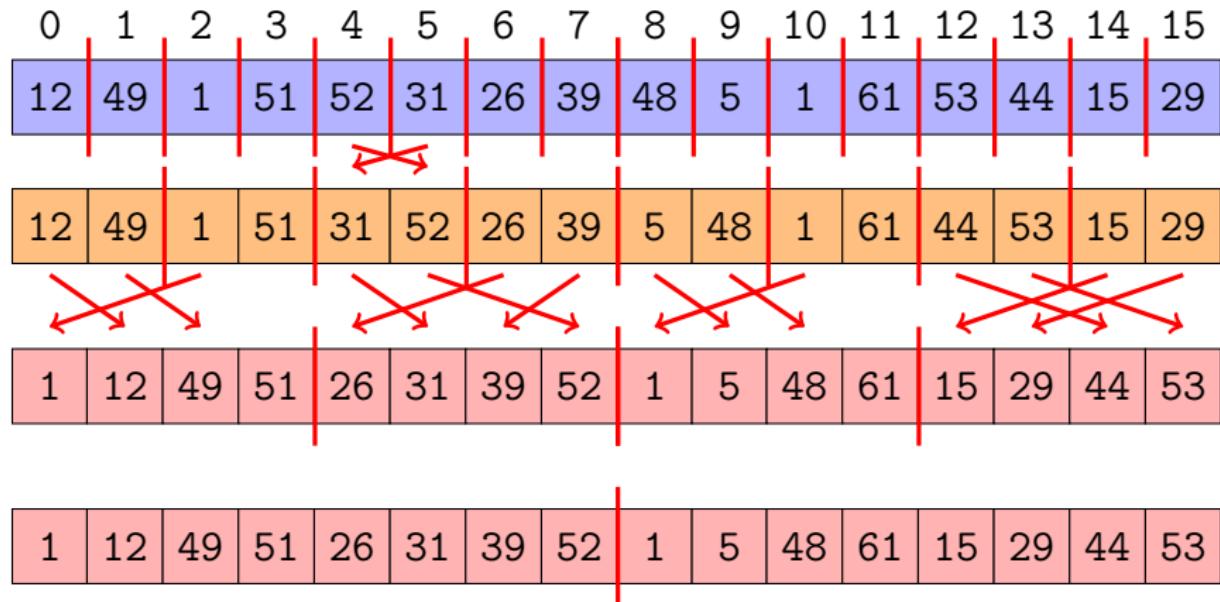
Merge Sort - Merge / Conquer



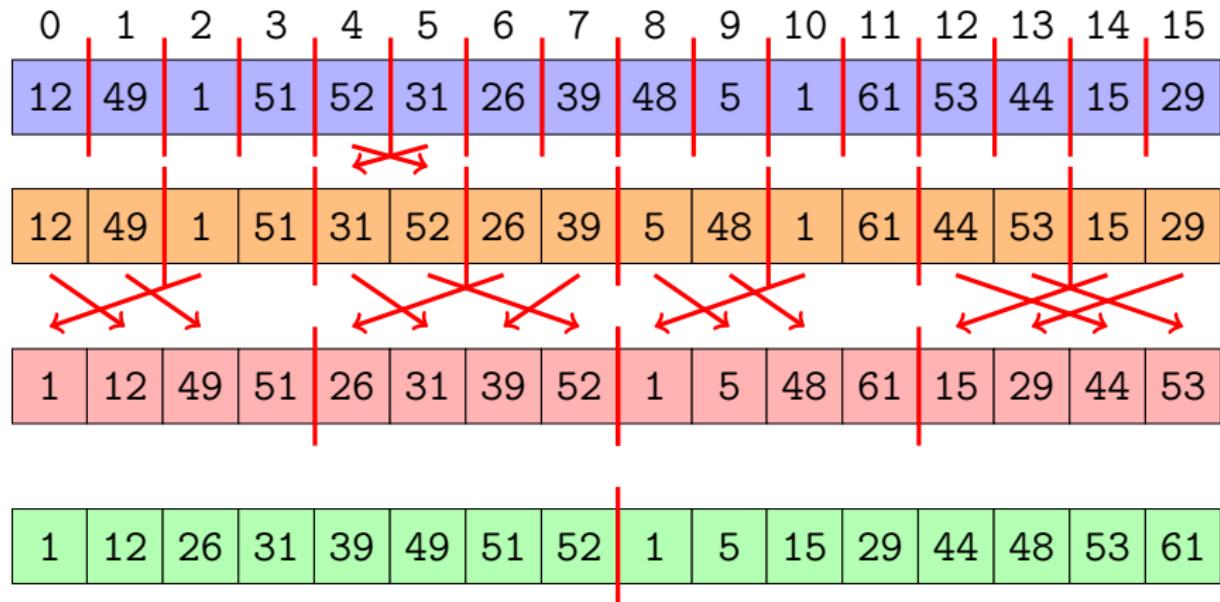
Merge Sort - Merge / Conquer



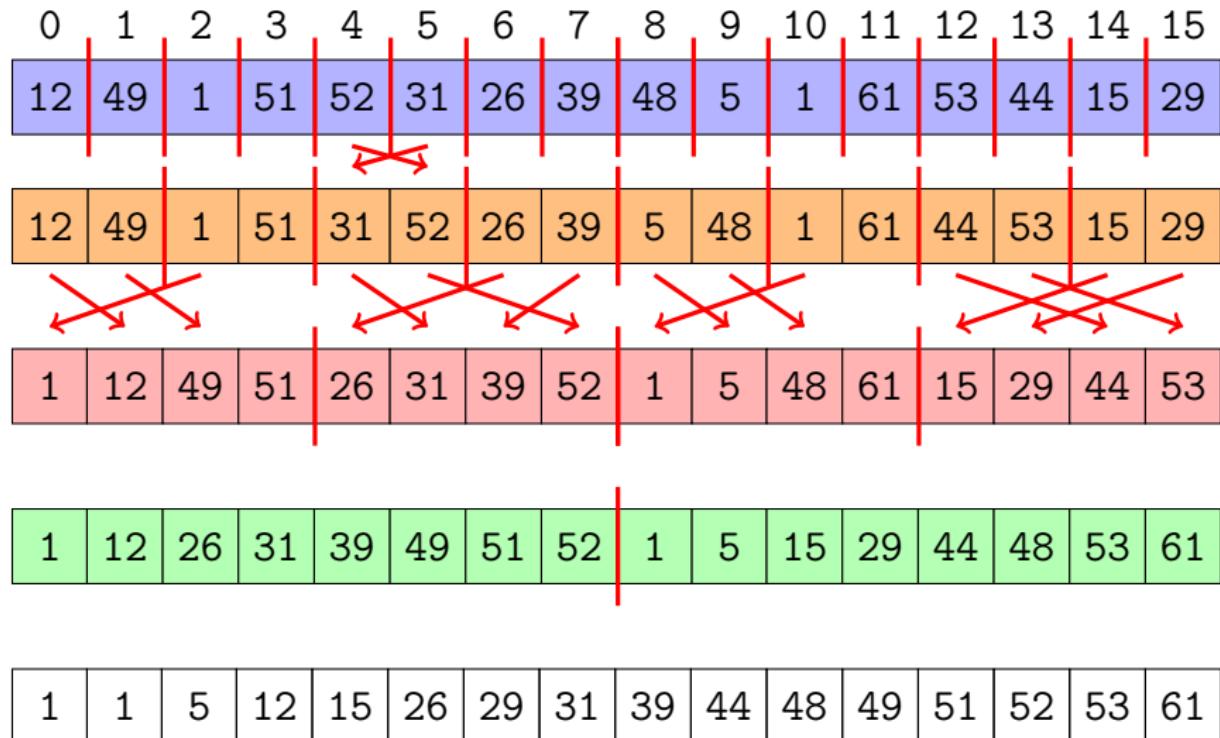
Merge Sort - Merge / Conquer



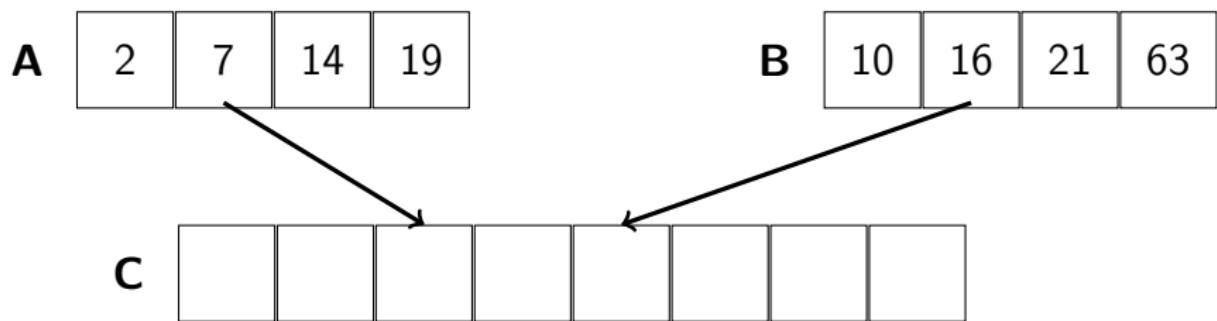
Merge Sort - Merge / Conquer



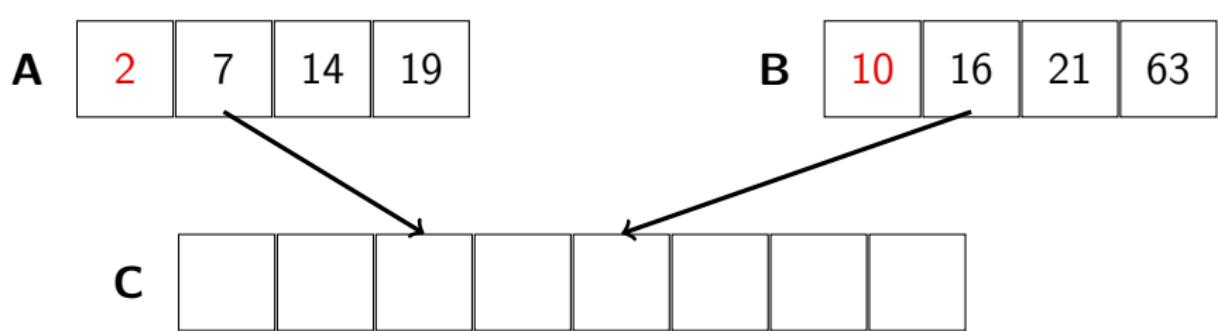
Merge Sort - Merge / Conquer



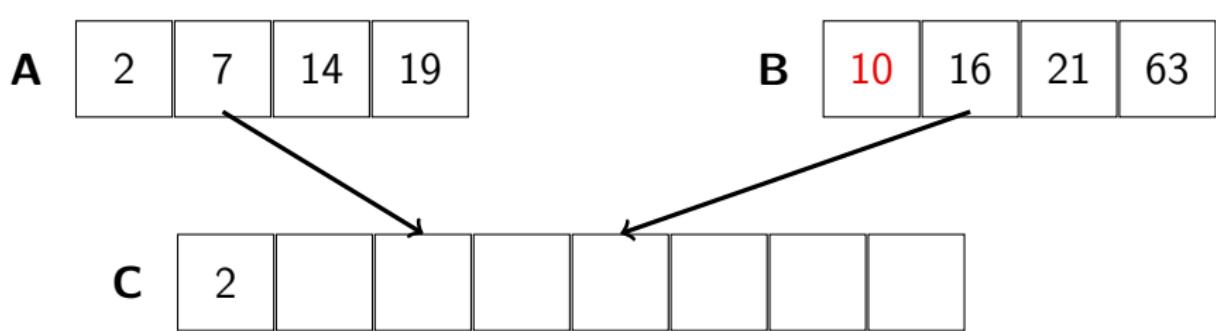
Merge Sort - Merge in detail



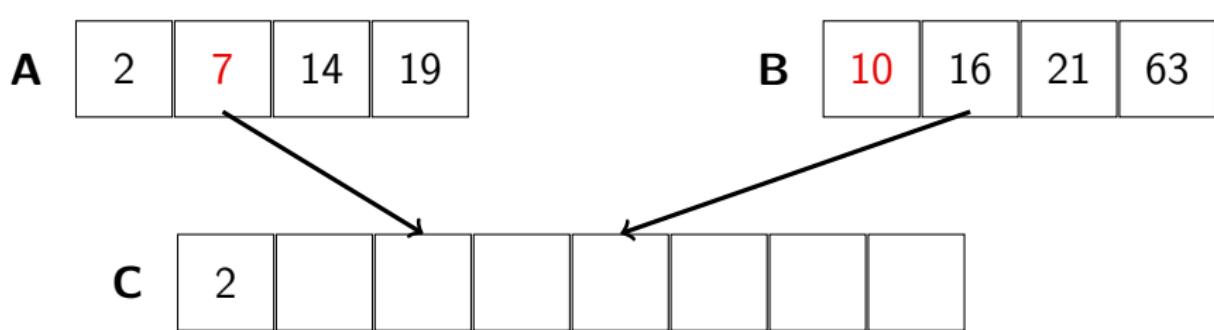
Merge Sort - Merge in detail



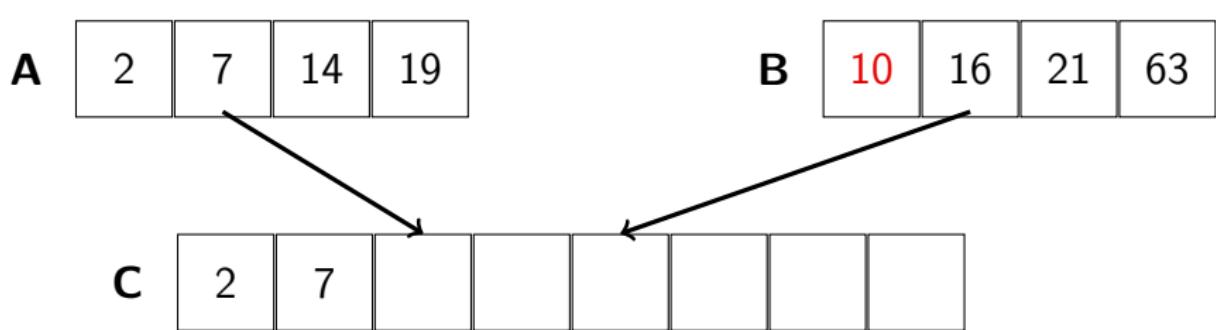
Merge Sort - Merge in detail



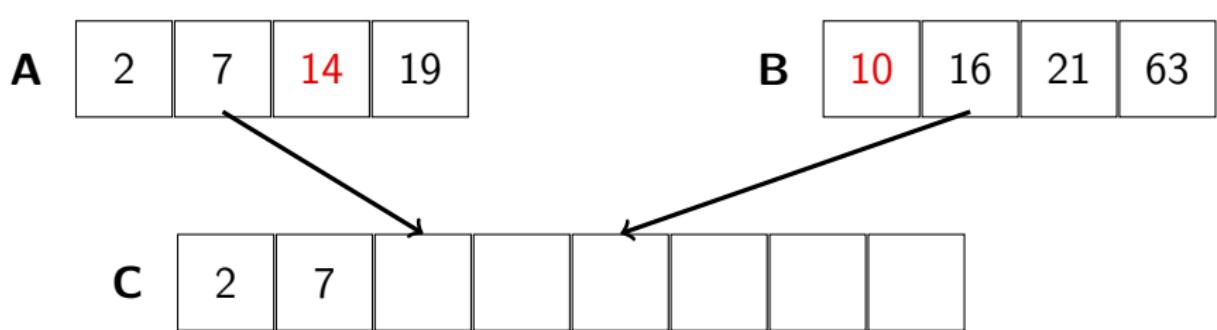
Merge Sort - Merge in detail



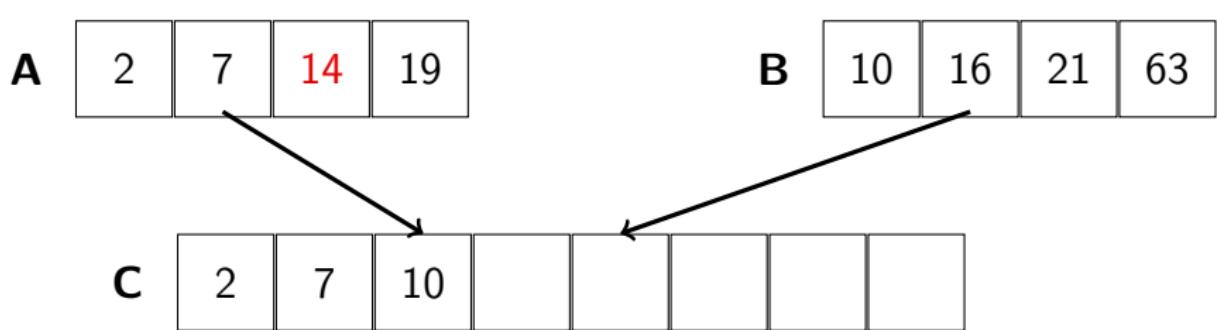
Merge Sort - Merge in detail



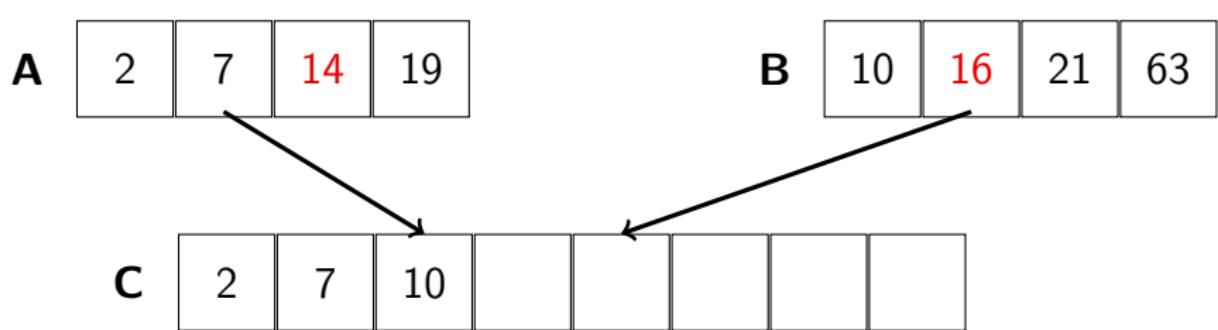
Merge Sort - Merge in detail



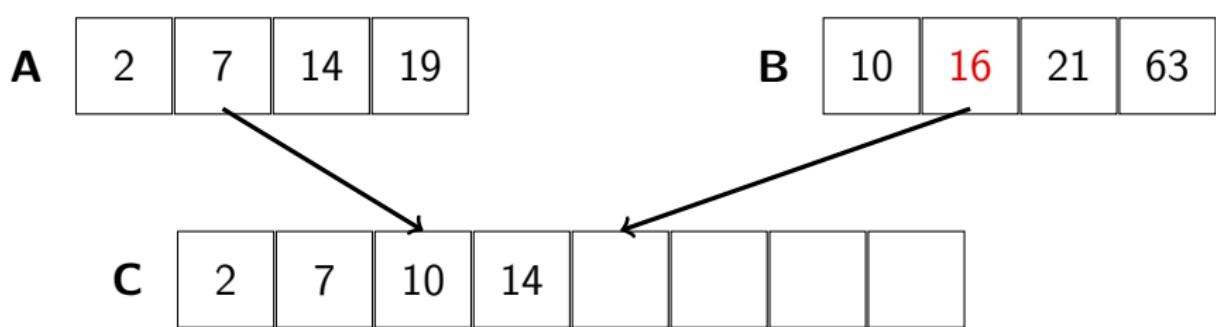
Merge Sort - Merge in detail



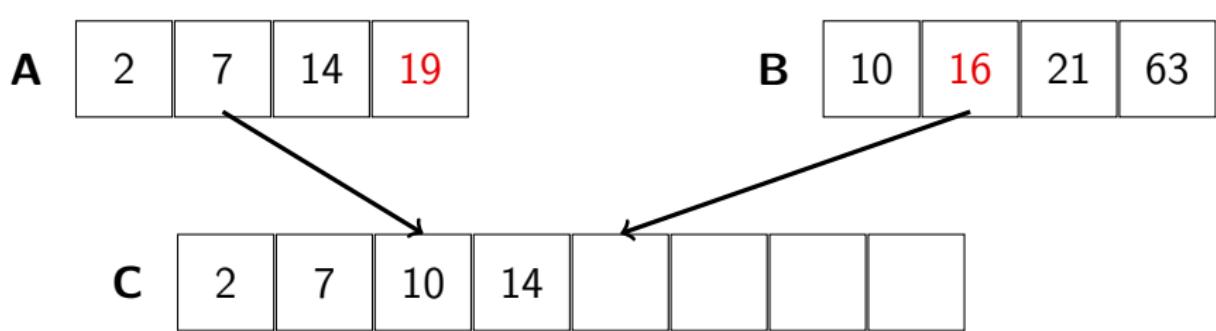
Merge Sort - Merge in detail



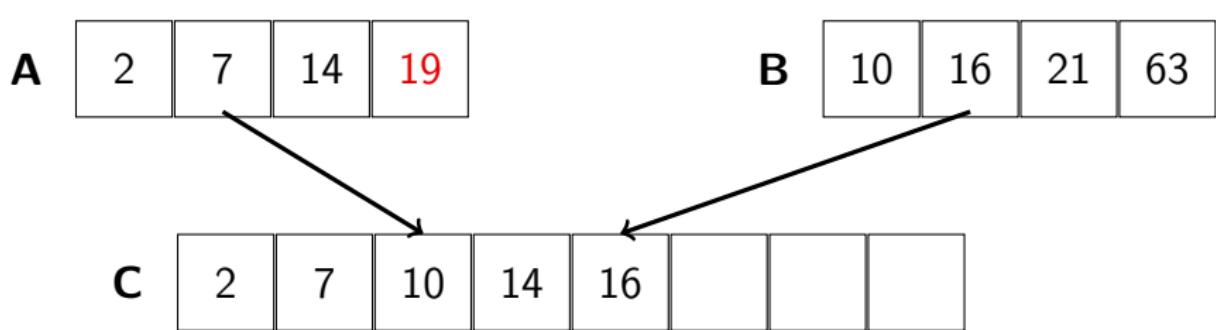
Merge Sort - Merge in detail



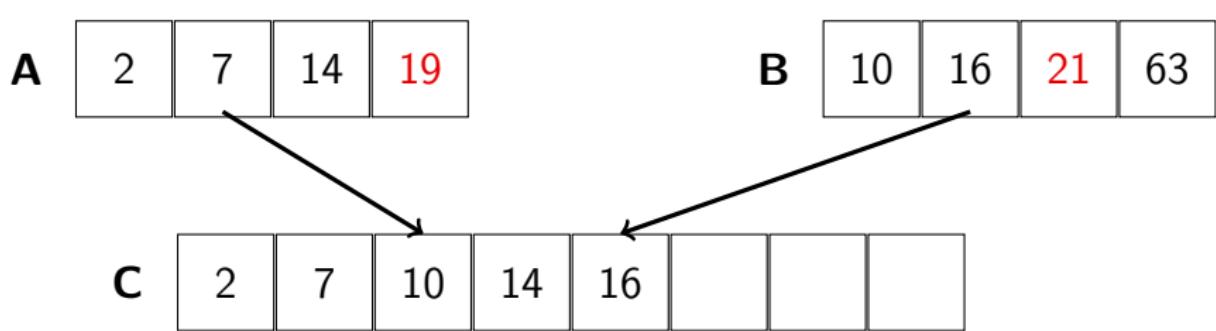
Merge Sort - Merge in detail



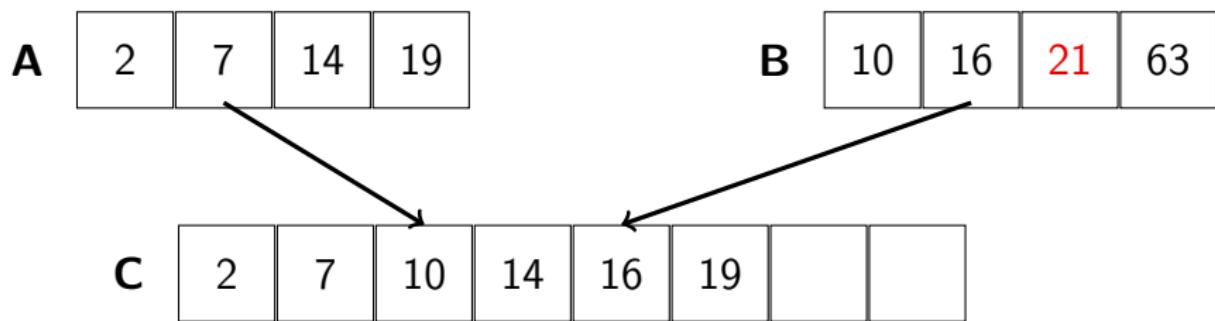
Merge Sort - Merge in detail



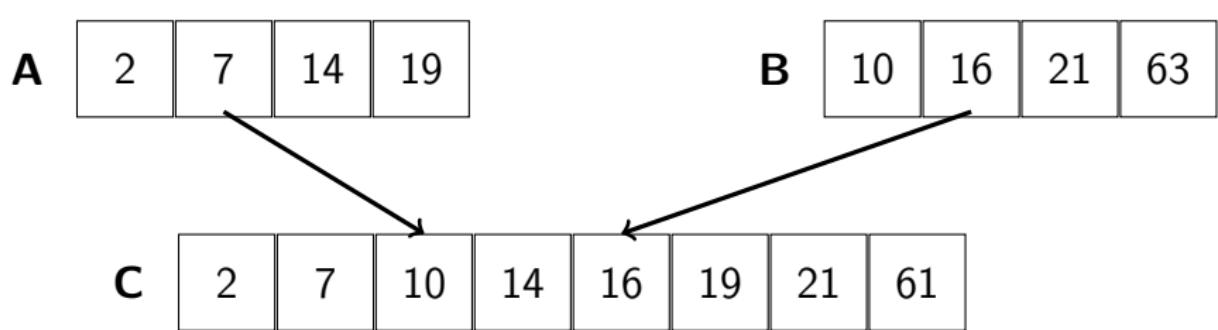
Merge Sort - Merge in detail



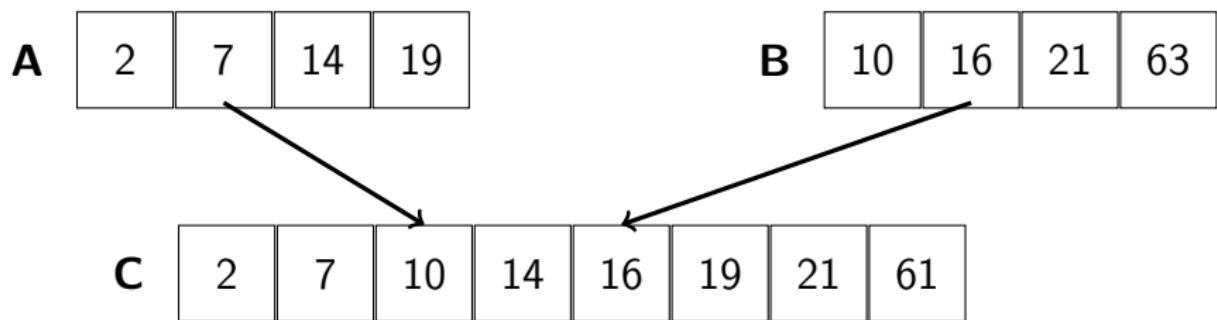
Merge Sort - Merge in detail



Merge Sort - Merge in detail



Merge Sort - Merge in detail

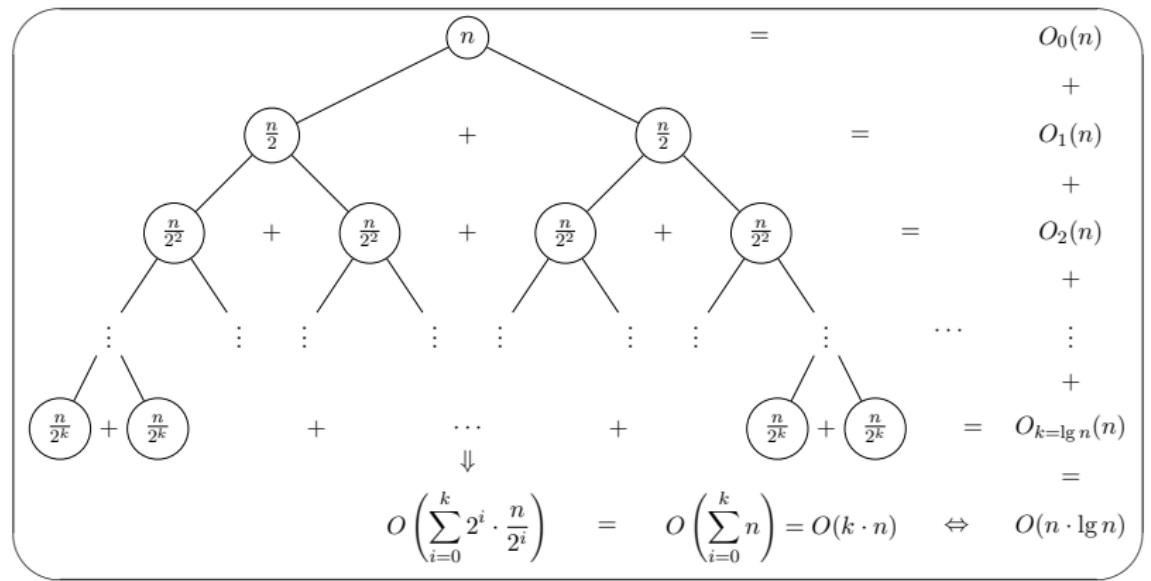


For two sorted arrays of size m and n , it takes $O(m + n)$ time to merge them into a sorted array of size $n + m$.

Merge Sort - Runtime Analysis

- At each level, merging all the sub arrays costs $O(n)$ time
- There are at most $\log_2 n$ levels
- Total runtime complexity is $O(n \log n)$

Merge Sort - Runtime Analysis - Picture



¹taken from <http://www.texexample.net/tikz/examples/merge-sort-recursion-tree/> by Manuel Kirsch
Copyright @ The University of Melbourne

Merge Sort - Other Properties

- Requires extra space
- Is a **stable** sorting algorithm. Order of equal items is preserved

- Quick Sort



THE UNIVERSITY OF

MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

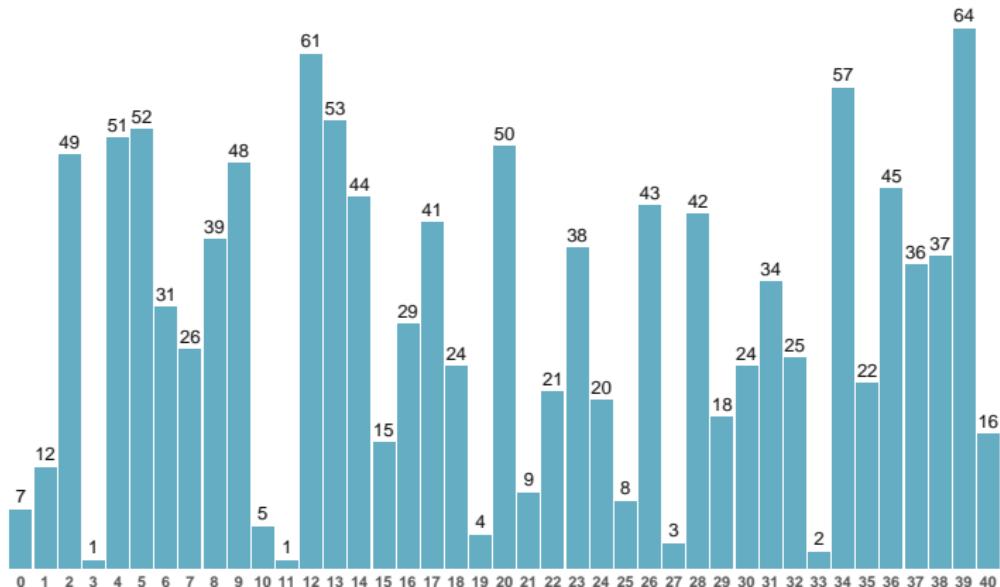
Today

Arrays and Searching/Sorting Algorithms

Searching in Arrays

Task: Determine if item x occurs in array $A[0, n - 1]$.

Search for $x=38$



Copyright © The University of Melbourne
Unsorted: In the worst case, $O(n)$ comparisons

Searching in Arrays

If array is **sorted**, can exit loop early:

```
1 // return item pos if found
2 // or -1 if not found
3 int i = 0;
4 while( i < n && A[ i ] < x )
5     i = i + 1
6 if( i == n || A[ i ] > x )
7     return -1;
8 return i;
```

Worst-case execution time is still linear.

Binary search

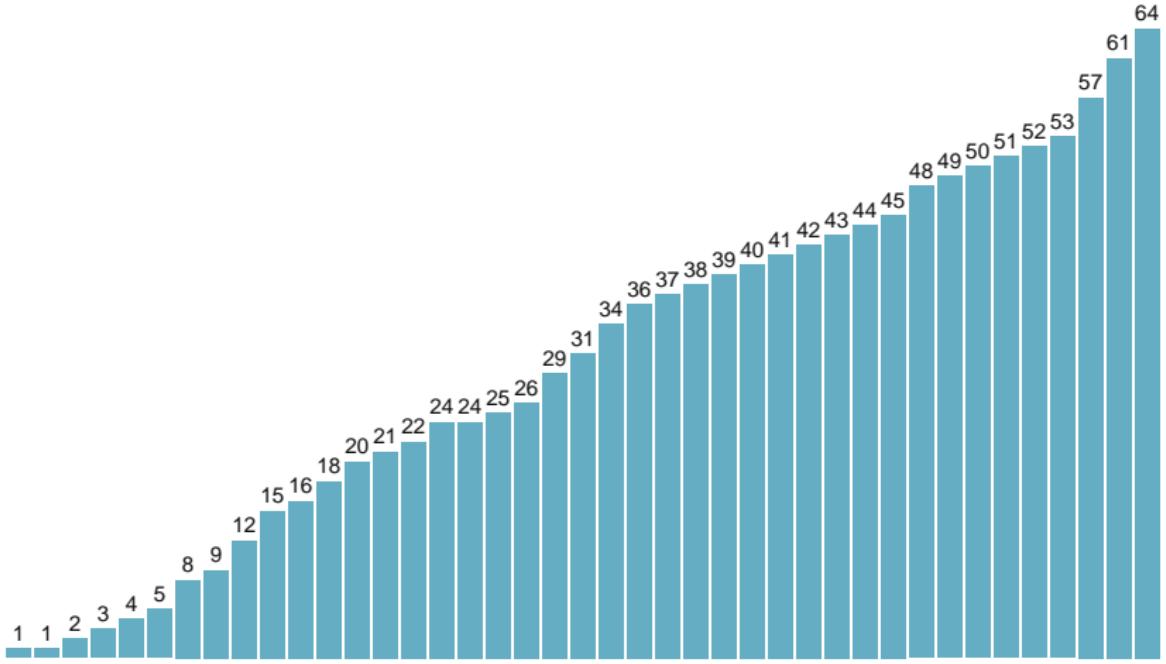
Idea: If array is sorted, search by repeated range halving is better.

Procedure:

- Compare item x with element in the middle element $A[m] = M$ of the current range $A[low, high]$. Here $m = (low + high)/2$.
- If $x < M$ we have to refine the range to $low = low, high = m - 1$.
- If $x > M$ we have to refine the range to $low = m + 1, high = high$.
- If $x == M$ we found the item

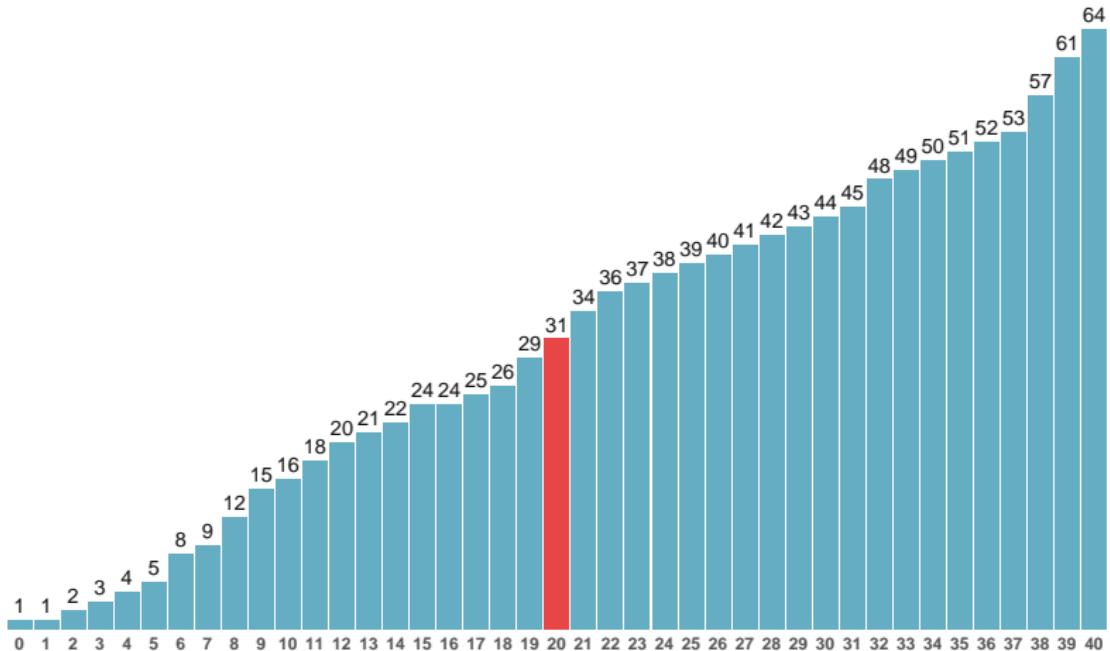
Binary search Example

Search for $x=38$



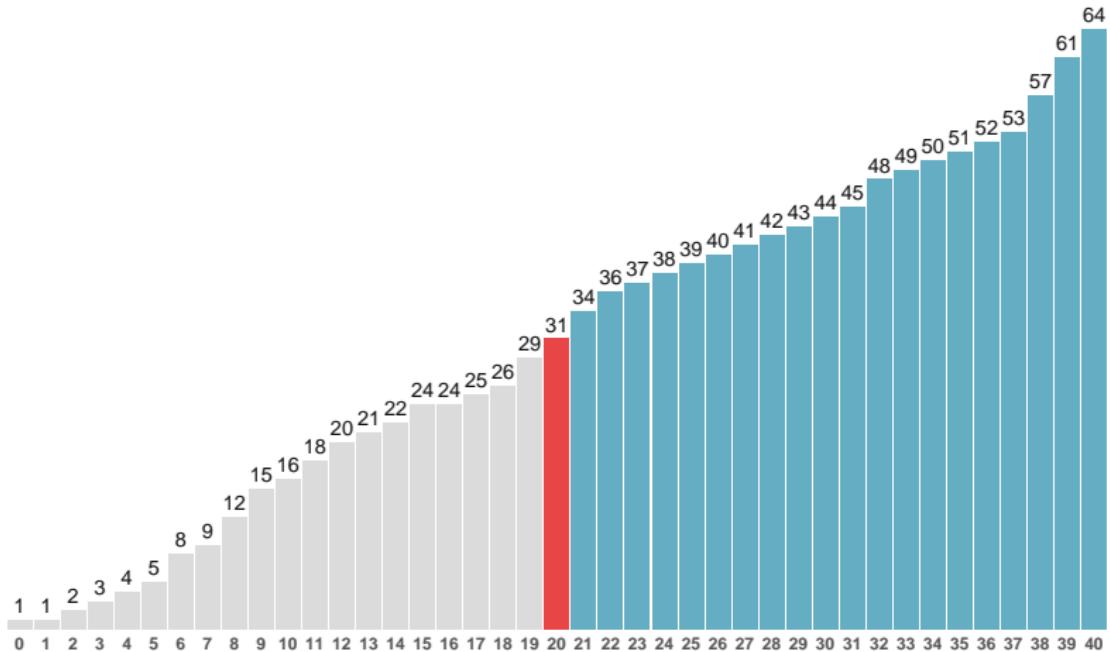
Binary search Example

Search for $x=38$



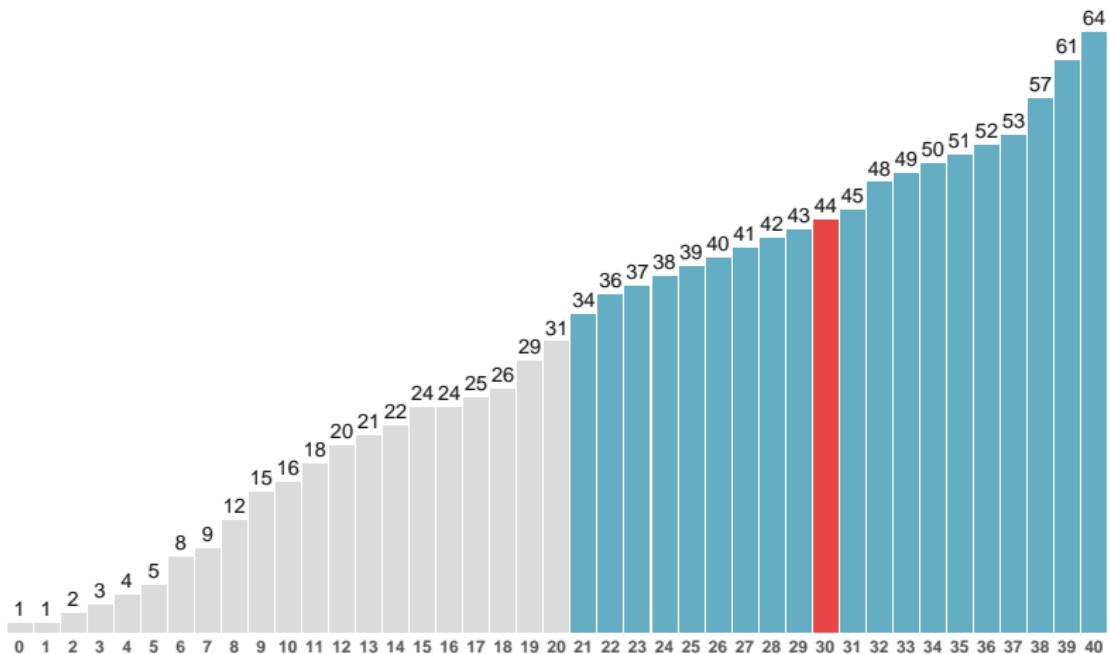
Binary search Example

Search for $x=38$



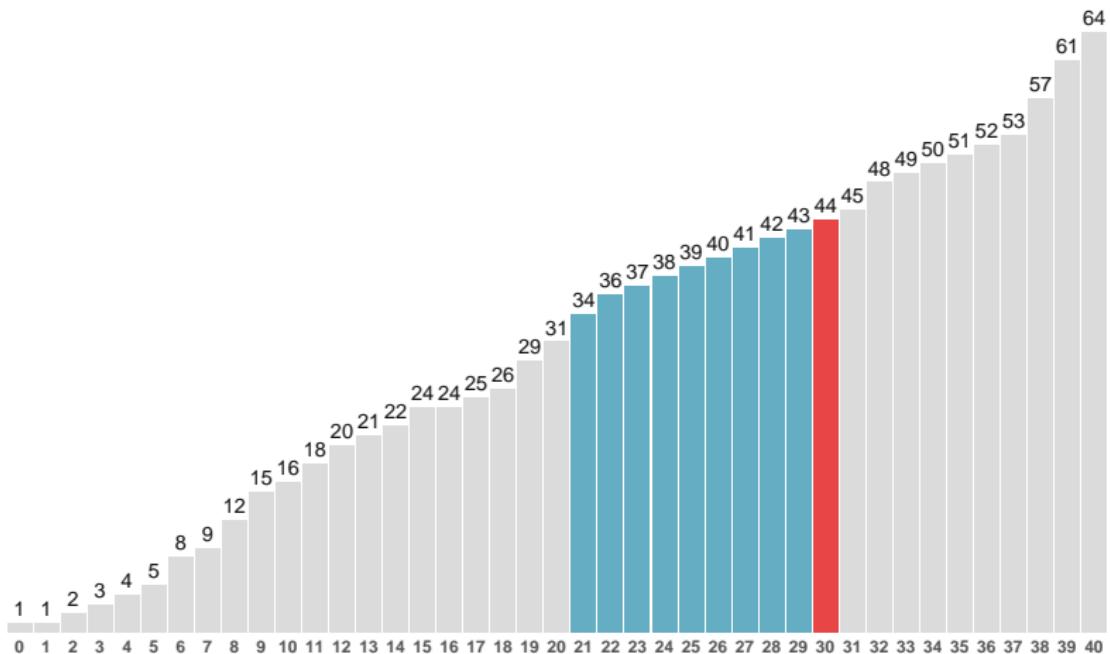
Binary search Example

Search for $x=38$



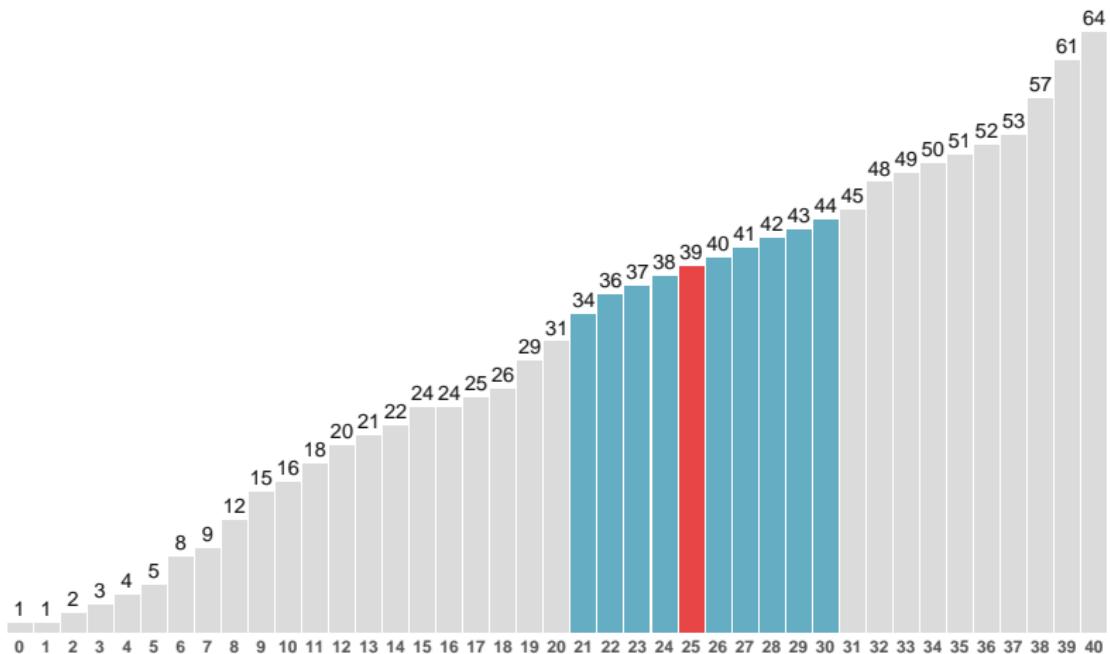
Binary search Example

Search for $x=38$



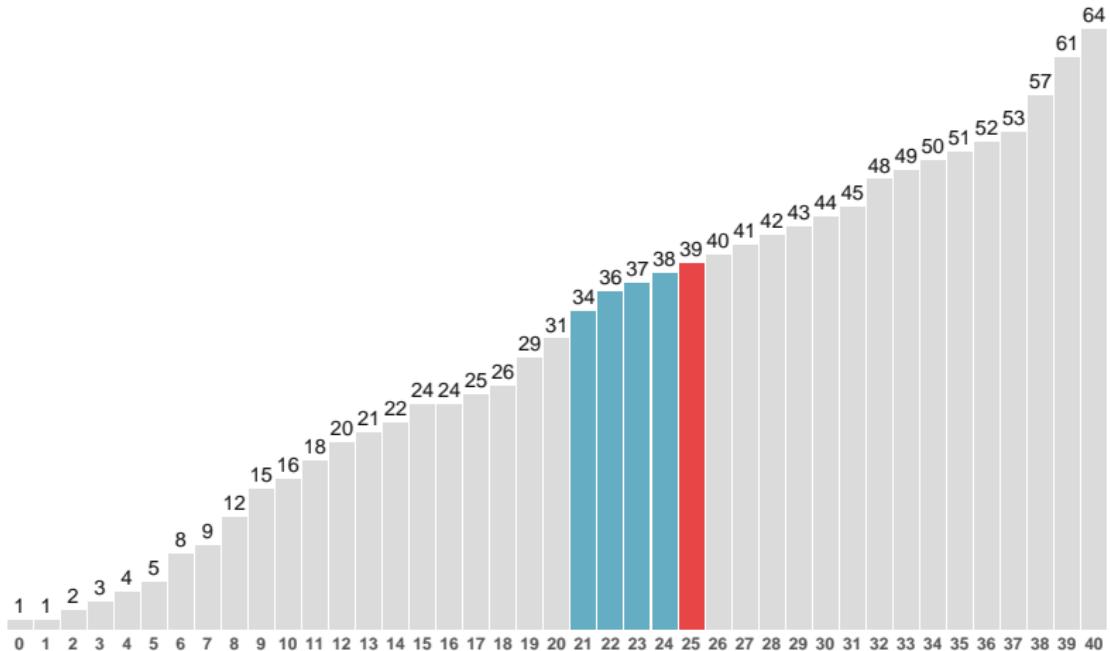
Binary search Example

Search for $x=38$



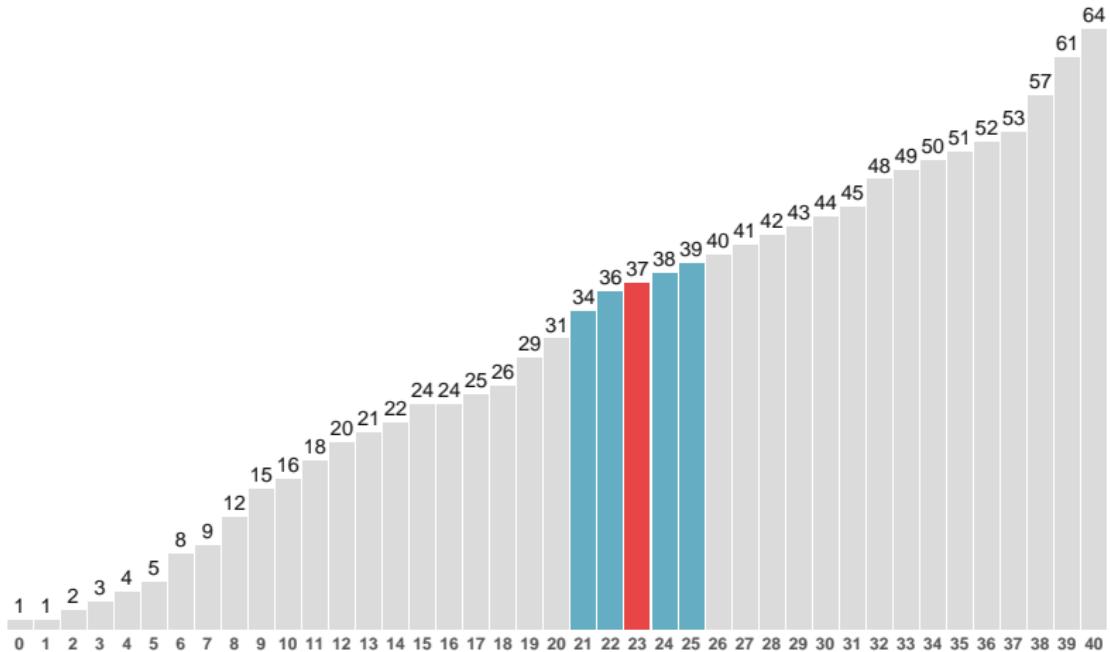
Binary search Example

Search for $x=38$



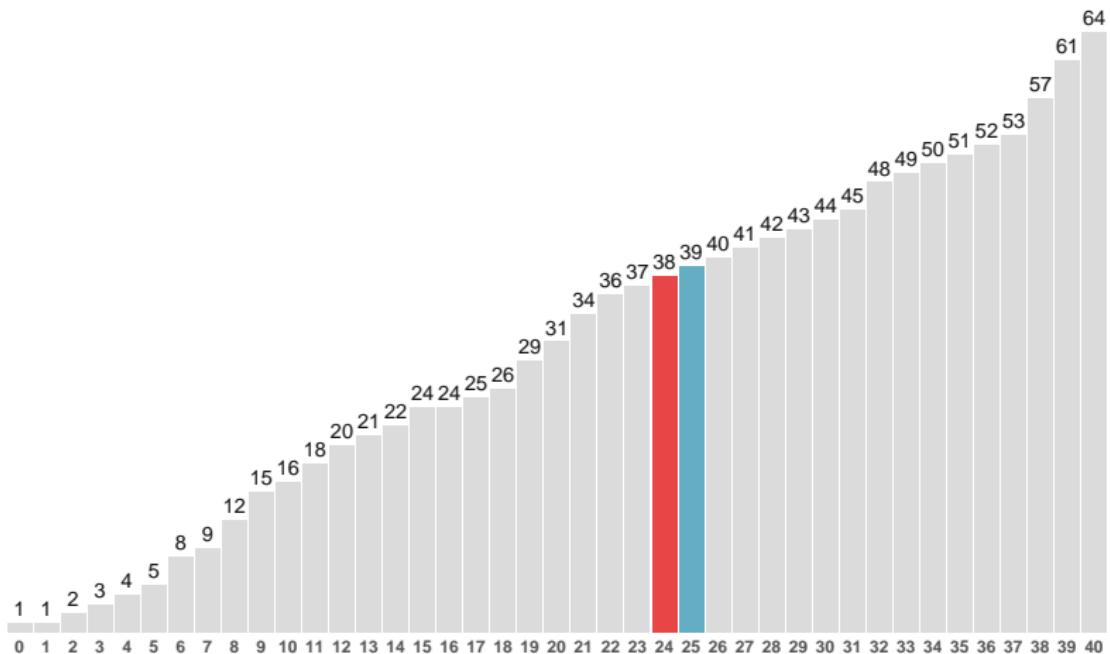
Binary search Example

Search for $x=38$



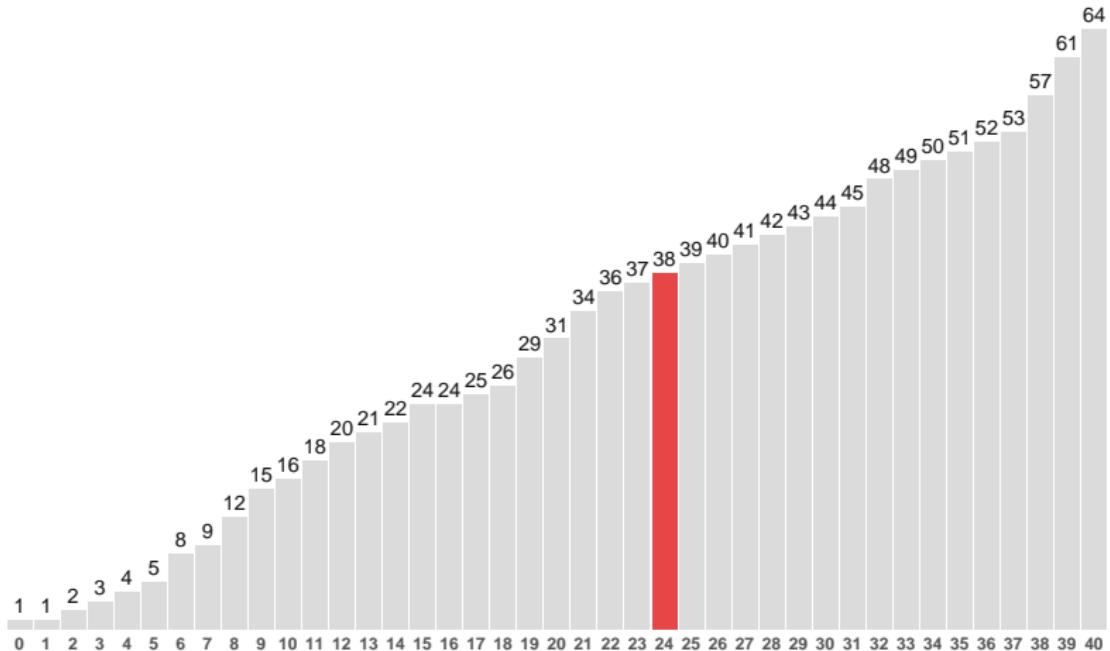
Binary search Example

Search for $x=38$



Binary search Example

Search for $x=38$



Binary search - Pseudocode Recursive

binary_search.c

```
1 int binary_search_r(int* A, int lo, int hi, int x)
2 {
3     if (lo >= hi) {
4         return -1;
5     }
6     int mid = (lo + hi) / 2;
7     int M = A[mid];
8     if (x < M) {
9         return binary_search_r(A, lo, mid, x);
10    } else if (x > M) {
11        return binary_search_r(A, mid + 1, hi, x);
12    } else {
13        return mid;
14    }
15 }
```

Binary search - Pseudocode Non-Recursive

binary_search.c

```
1 int binary_search(int* A, int lo, int hi, int x)
2 {
3     while (lo < hi) {
4         int m = (lo + hi) / 2;
5         if (x < A[m]) {
6             hi = m;
7         } else if (x > A[m]) {
8             lo = m + 1;
9         } else {
10            return m;
11        }
12    }
13    return -1;
14 }
```

Binary search - Function pointer comparison

```
1 int bs_funcmp (void* A, int lo, int hi, void* key,
2                 int (*cmp)(const void* a, const void* b)){
3
4     if (lo >= hi){
5         return -1;
6     }
7     int mid = (lo+hi)/2;
8     void* M = A[mid];
9     int outcome = cmp(key, M);
10
11    if (outcome < 0){
12        return bs_funcmp(A, lo, mid, key, cmp);
13    } else if (outcome > 0){
14        return bs_funcmp(A, mid+1, hi, key, cmp);
15    } else {
16        return mid;
17    }
18 }
```

Comparison Functions - A standard C paradigm I

The function `cmp` is a standard C paradigm.

It is usually passed in to a sorting or searching function as a **function argument**.

It compares (via pointers to underlying objects) two elements, and returns:

- –ve, if first item should come prior to second
- 0, if two items can be considered to be equal
- +ve, if first item should come after second.

Comparison Functions - A standard C paradigm II

From the man page of `qsort`:

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

From the man page of `strcmp`:

The `strcmp()` function compares the two strings s_1 and s_2 . It returns an integer less than, equal to, or greater than zero if s_1 is found, respectively, to be less than, to match, or be greater than s_2 .

Binary search - Analysis

How many basic operations does binary search require?

Based on the recursive pseudo code above, the time taken is bounded above by $C(n)$, where

$$C(n) = \begin{cases} 1 & \text{if } n \leq 1; \\ 1 + C(\lfloor n/2 \rfloor) & \text{if } n > 1. \end{cases}$$

The exact solution of this one is

$$C(n) = 1 + \lfloor \log_2 n \rfloor \in O(\log n)$$

Intuition: How often do I have to half a range of size n until it is of size 1? $\log_2 n$ times

Binary search - Asymptotic Growth

n	$\log_2 n$
10	3.3
100	6.6
1,000	9.9
10,000	13.3
100,000	16.6
1,000,000	19.9
10,000,000	23.3
100,000,000	26.6
1,000,000,000	29.9
10,000,000,000	33.2
100,000,000,000	36.5
1,000,000,000,000	39.9
10,000,000,000,000	43.2

Binary search - Exercise

Exercise 14

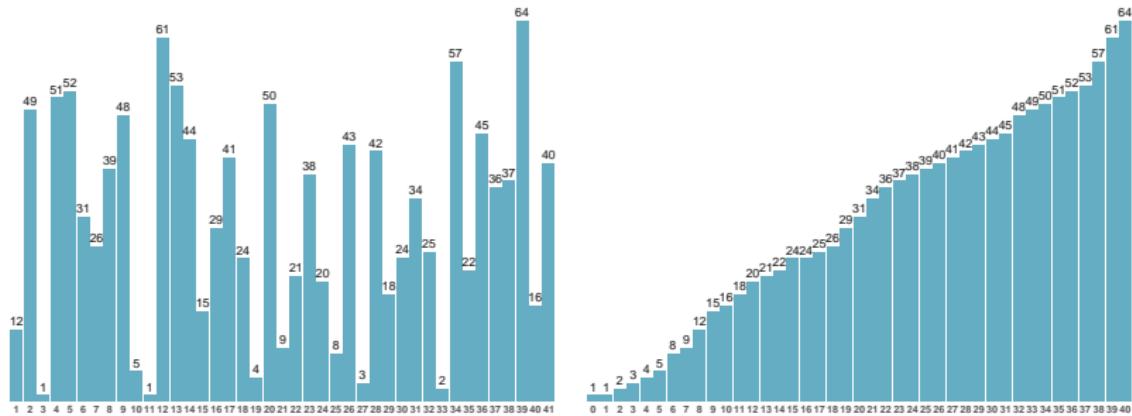
Modify the `binary_search.c` program to measures the runtime and number of basic operations executed by the binary search algorithm for different input sizes.

Exercise 15

Is it possible to apply binary search to a linked list? Discuss!

Sorting

How did the array get sorted? Sorting can increase the efficiency of many applications such as duplicate detection or searching.



Review: Insertion Sort I

One simple algorithm is called **insertion sort**:

- One part of the array is in sorted order (initially $A[0]$)
- Increase the size of the sorted sub array by **inserting** items in the correct position
- Insertion in the array is done by swapping elements into the correct position
- Every iteration, sorted sub array size increases by one

Review: Insertion Sort II

When applied to the array $\{22, 14, 17, 42, 27, 28, 23\}$:

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
Initially	22	14	17	42	27	28	23
After i=0	22	14	17	42	27	28	23
After i=1	14	22	17	42	27	28	23
After i=2	14	17	22	42	27	28	23
After i=3	14	17	22	42	27	28	23
After i=4	14	17	22	27	42	28	23
After i=5	14	17	22	27	28	42	23
After i=6	14	17	22	23	27	28	42

Review: Insertion Sort III

```
1 void insertion_sort(int* a, int n)
2 {
3     for (size_t i = 1; i < n; ++i) {
4         int tmp = a[i];
5         size_t j = i;
6         while (j > 0 && tmp < a[j - 1]) {
7             a[j] = a[j - 1];
8             --j;
9         }
10        a[j] = tmp;
11    }
12 }
```

Review: Insertion Sort IV

Insertion sort is a relatively bad algorithm – inserting $A[i]$ might require i swaps (what input causes this?), making the total number of swaps as large as

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \approx n^2/2 \in O(n^2).$$

The **worst case** (and average case) behavior of insertion sort is $O(n^2)$ in the number of items being sorted.

Exercise 16

Where does the summation on the previous slide come from?
What is the worst case input for Insertion Sort? Discuss!

Better Sorting?

Does sorting need to take quadratic time?

Luckily there are much better sorting algorithms. If this was not the case, solving many of the problems we encounter on a daily basis would not be possible!

For the rest of this week, we will look at two popular, efficient sorting algorithms, [Merge Sort](#) and [Quick Sort](#).

Merge Sort - Idea

- Simple **divide-and-conquer** approach
- Split array of size n into two arrays of size $s = n/2$
- Sort arrays of size $n/2$
- Merge sorted arrays back together to get sorted array of size n
- Keep splitting recursively until $s = 1$

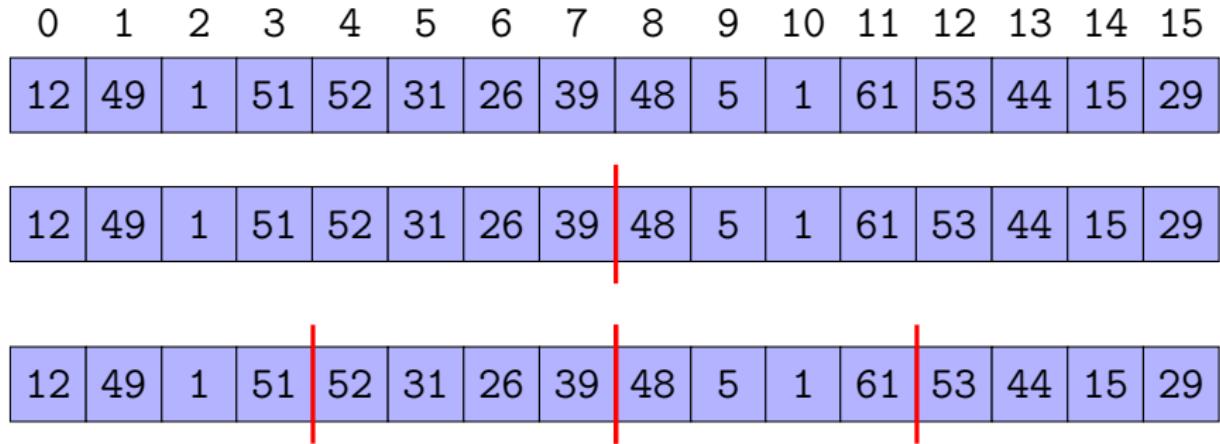
Merge Sort - Split / Divide

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29

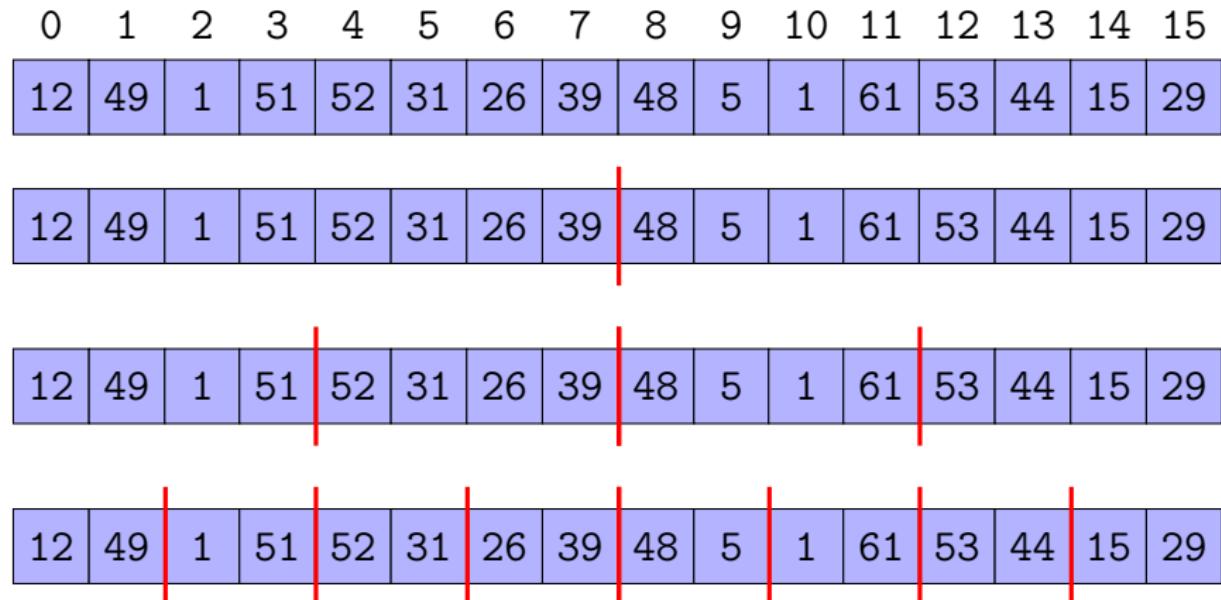
Merge Sort - Split / Divide

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29

Merge Sort - Split / Divide



Merge Sort - Split / Divide



Merge Sort - Split / Divide

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29

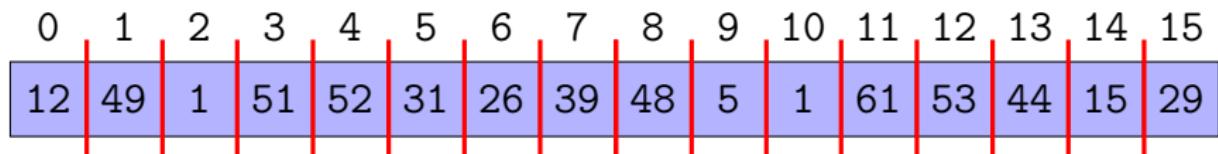
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29
----	----	---	----	----	----	----	----	----	---	---	----	----	----	----	----

12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29
----	----	---	----	----	----	----	----	----	---	---	----	----	----	----	----

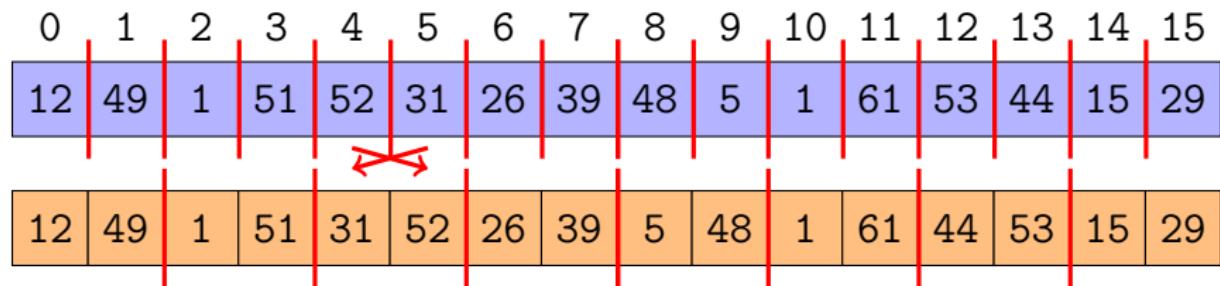
12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29
----	----	---	----	----	----	----	----	----	---	---	----	----	----	----	----

12	49	1	51	52	31	26	39	48	5	1	61	53	44	15	29
----	----	---	----	----	----	----	----	----	---	---	----	----	----	----	----

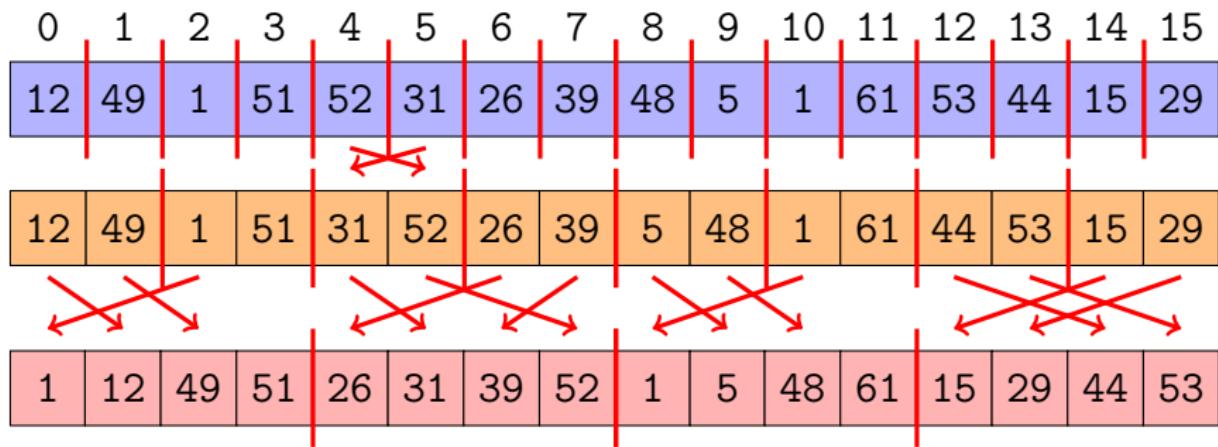
Merge Sort - Merge / Conquer



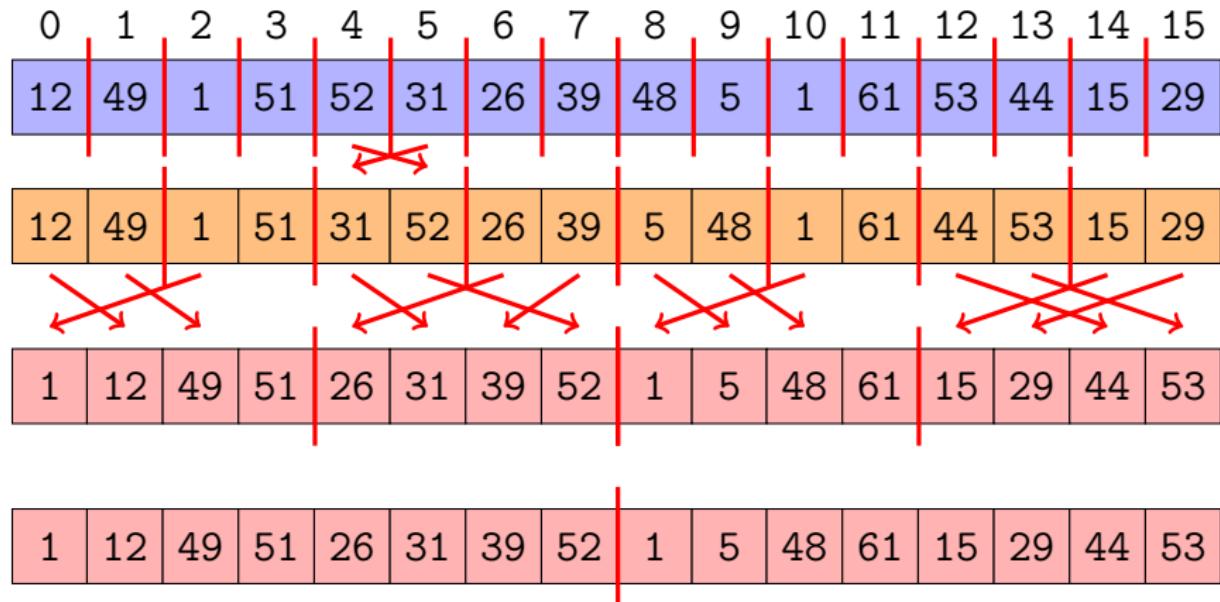
Merge Sort - Merge / Conquer



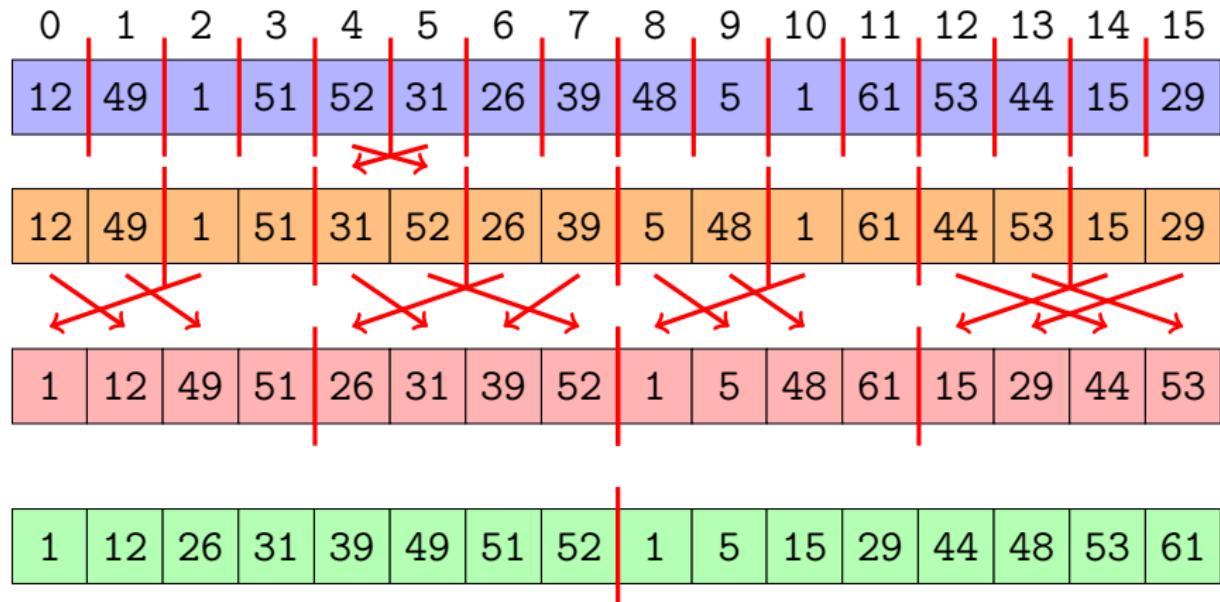
Merge Sort - Merge / Conquer



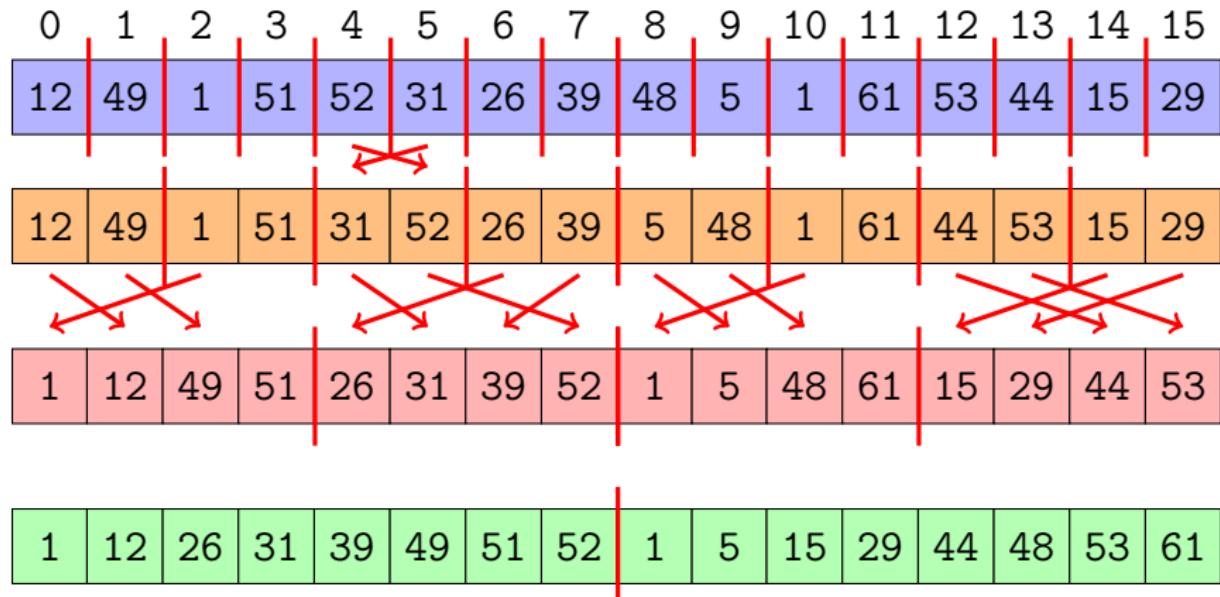
Merge Sort - Merge / Conquer



Merge Sort - Merge / Conquer

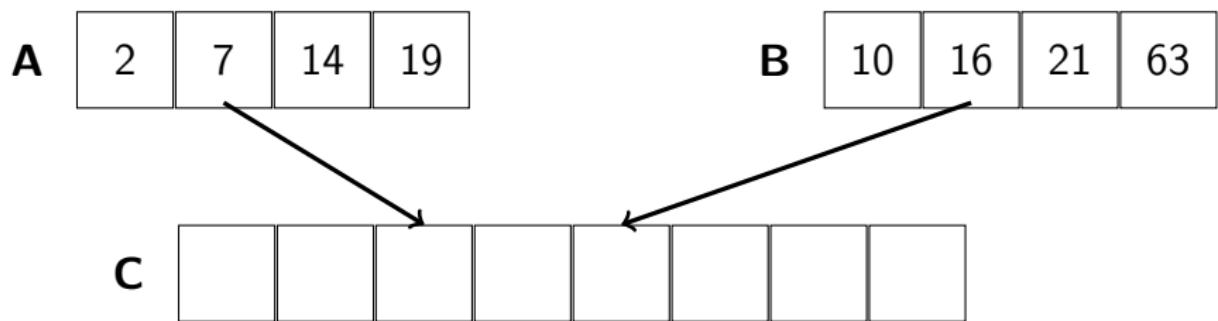


Merge Sort - Merge / Conquer

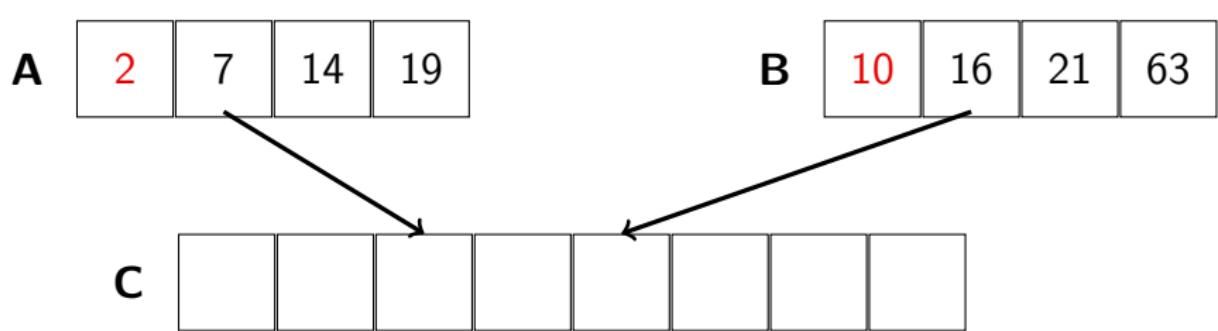


1	1	5	12	15	26	29	31	39	44	48	49	51	52	53	61
---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----

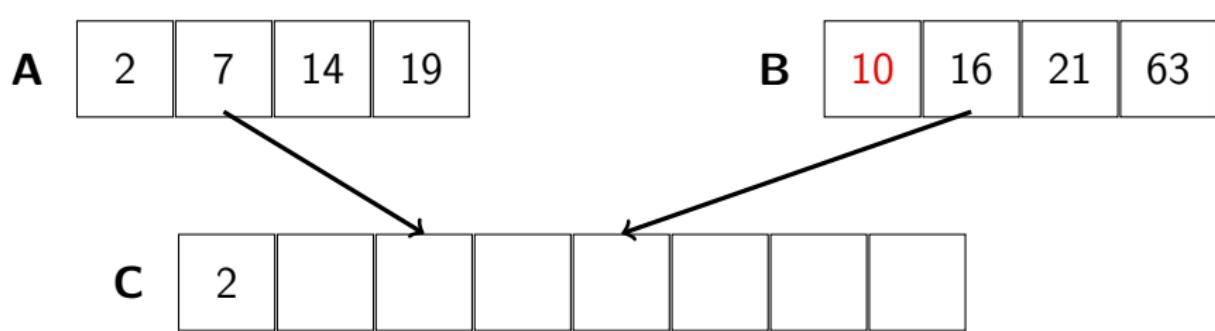
Merge Sort - Merge in detail



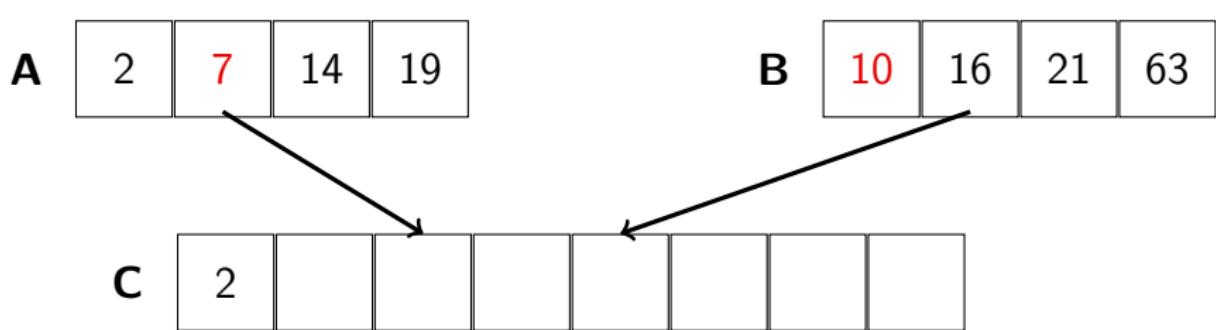
Merge Sort - Merge in detail



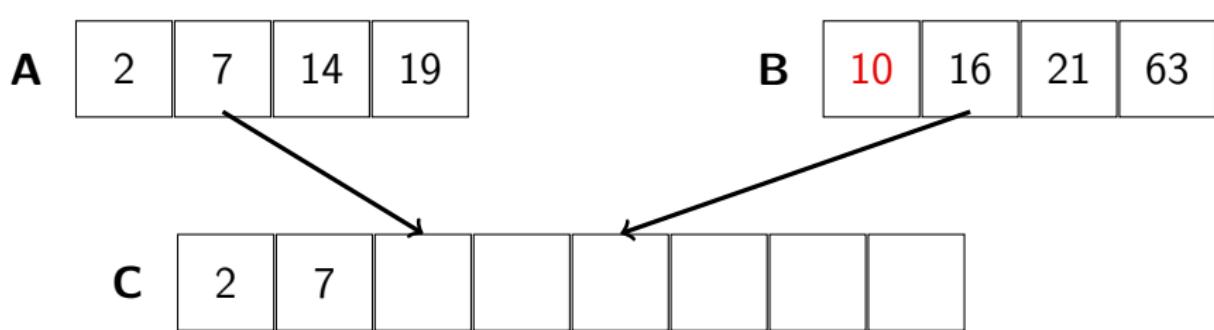
Merge Sort - Merge in detail



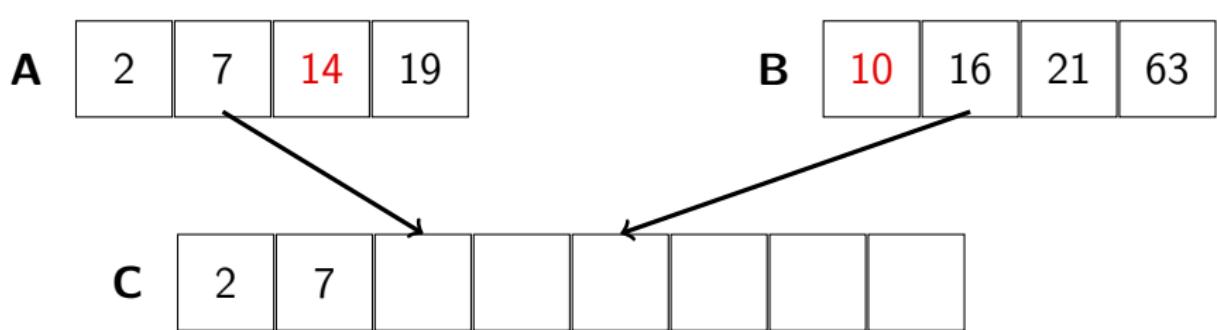
Merge Sort - Merge in detail



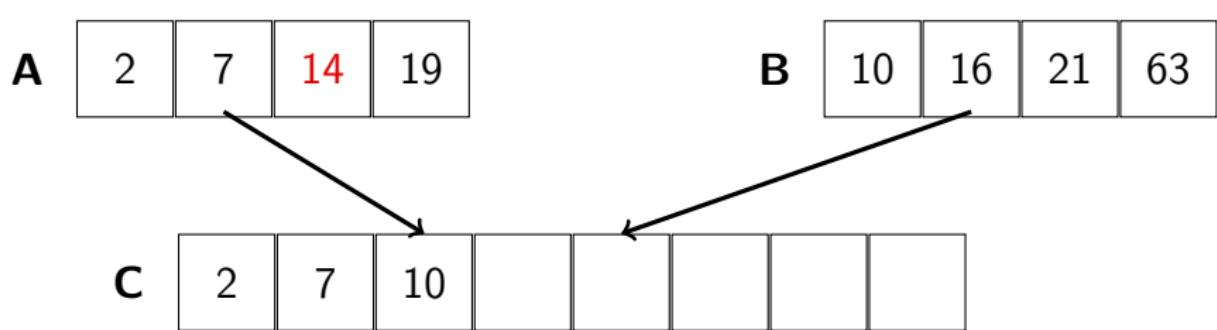
Merge Sort - Merge in detail



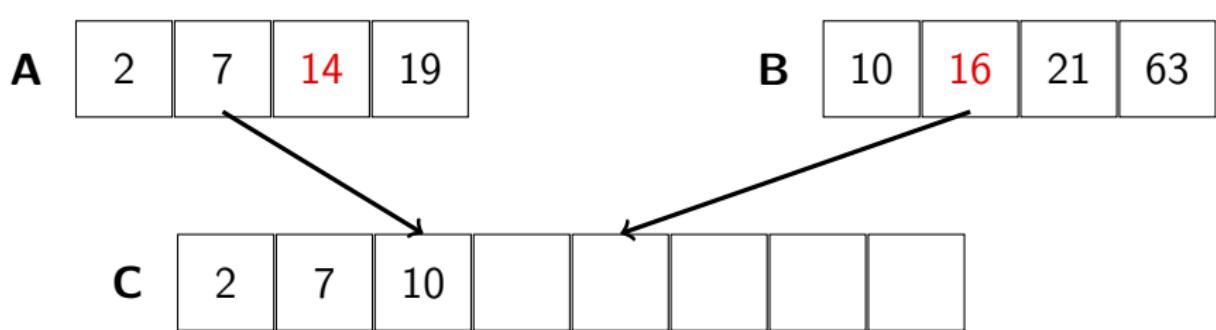
Merge Sort - Merge in detail



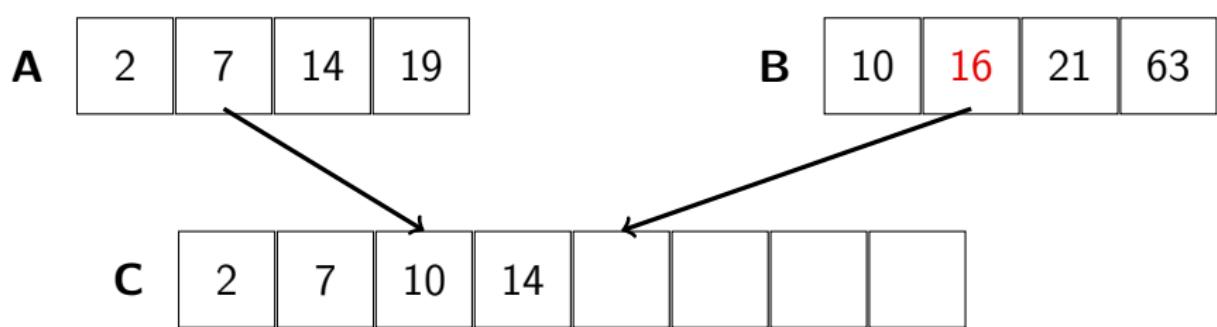
Merge Sort - Merge in detail



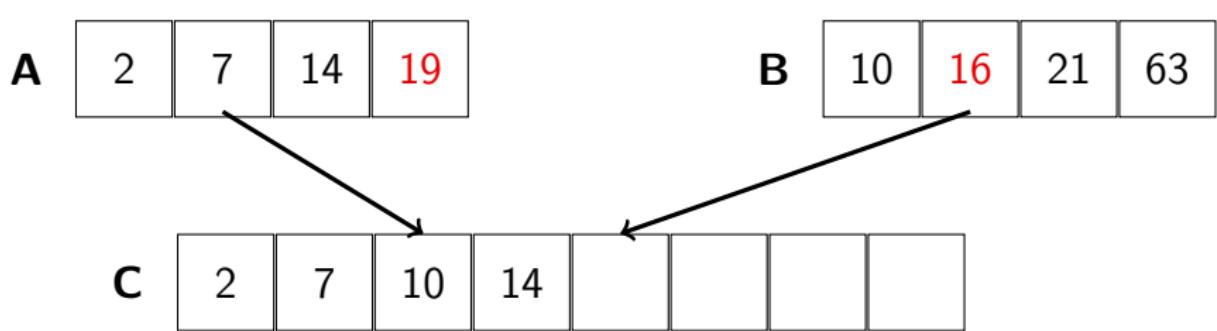
Merge Sort - Merge in detail



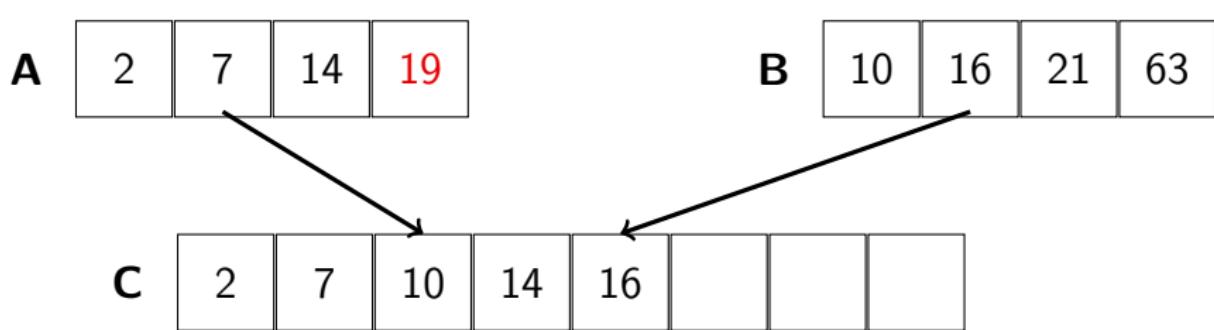
Merge Sort - Merge in detail



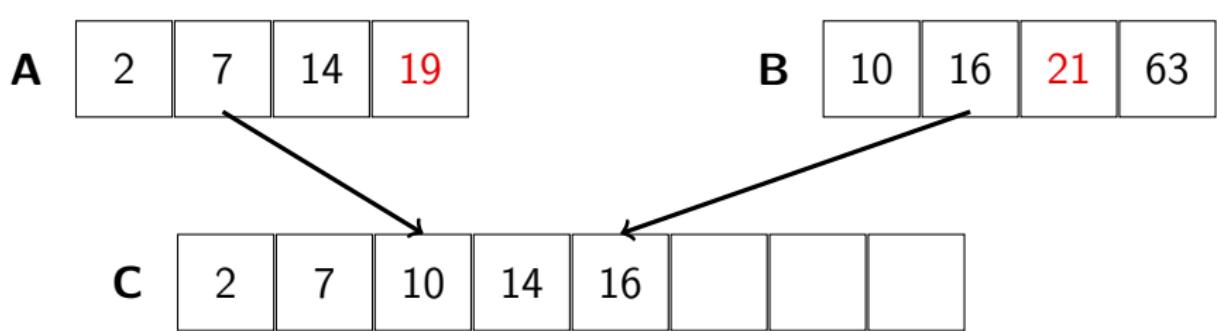
Merge Sort - Merge in detail



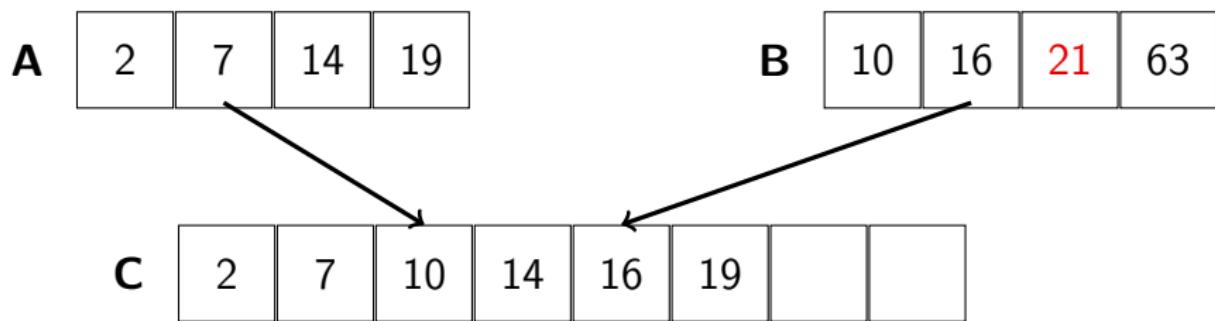
Merge Sort - Merge in detail



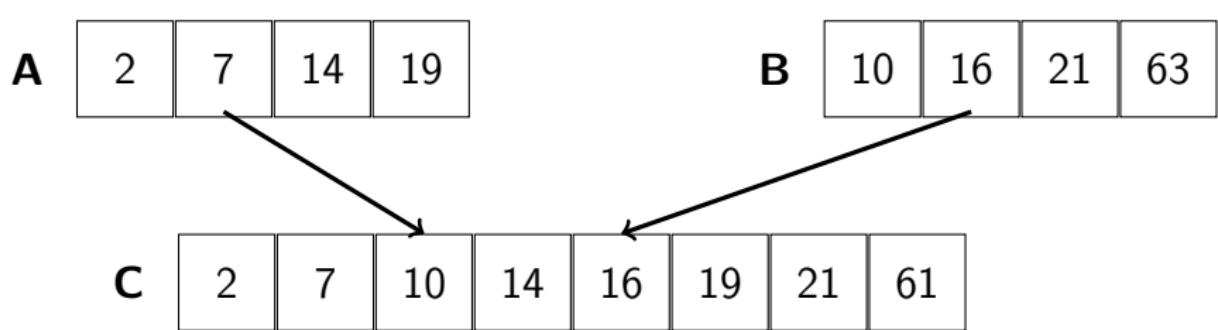
Merge Sort - Merge in detail



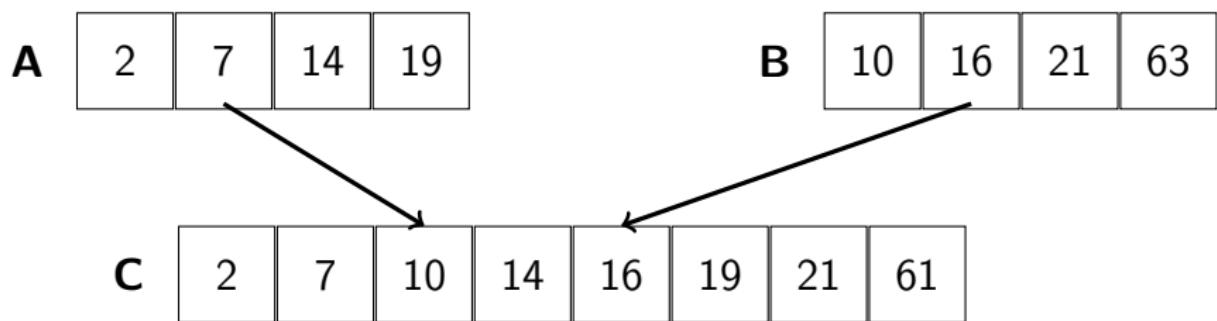
Merge Sort - Merge in detail



Merge Sort - Merge in detail



Merge Sort - Merge in detail

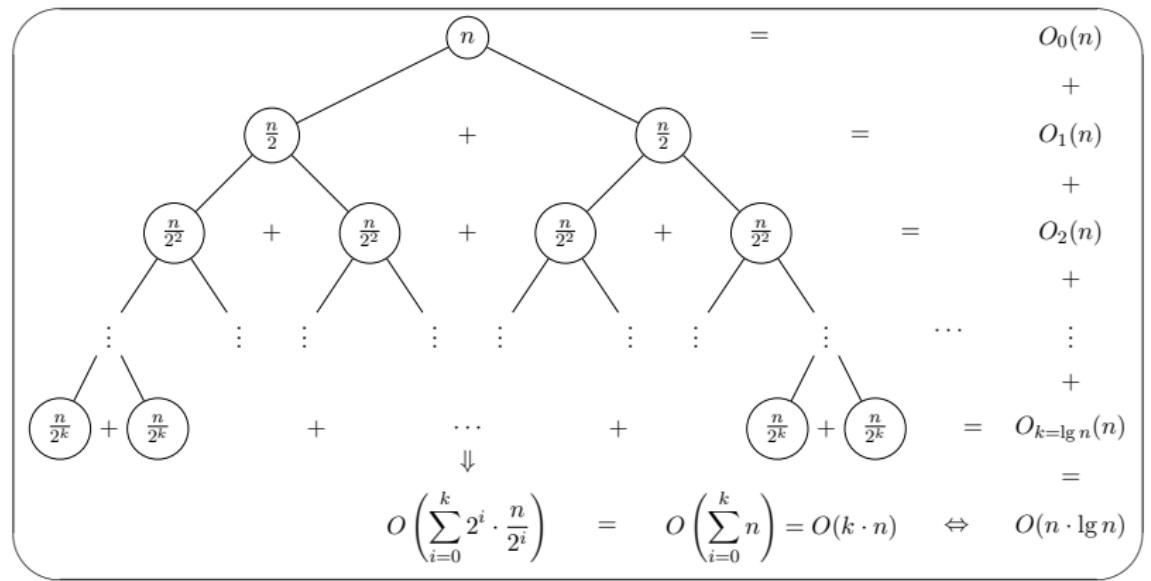


For two sorted arrays of size m and n , it takes $O(m + n)$ time to merge them into a sorted array of size $n + m$.

Merge Sort - Runtime Analysis

- At each level, merging all the sub arrays costs $O(n)$ time
- There are at most $\log_2 n$ levels
- Total runtime complexity is $O(n \log n)$

Merge Sort - Runtime Analysis - Picture



¹taken from <http://www.texexample.net/tikz/examples/merge-sort-recursion-tree/> by Manuel Kirsch

Merge Sort - Other Properties

- Requires extra space
- Is a **stable** sorting algorithm. Order of equal items is preserved

- Quick Sort



THE UNIVERSITY OF

MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

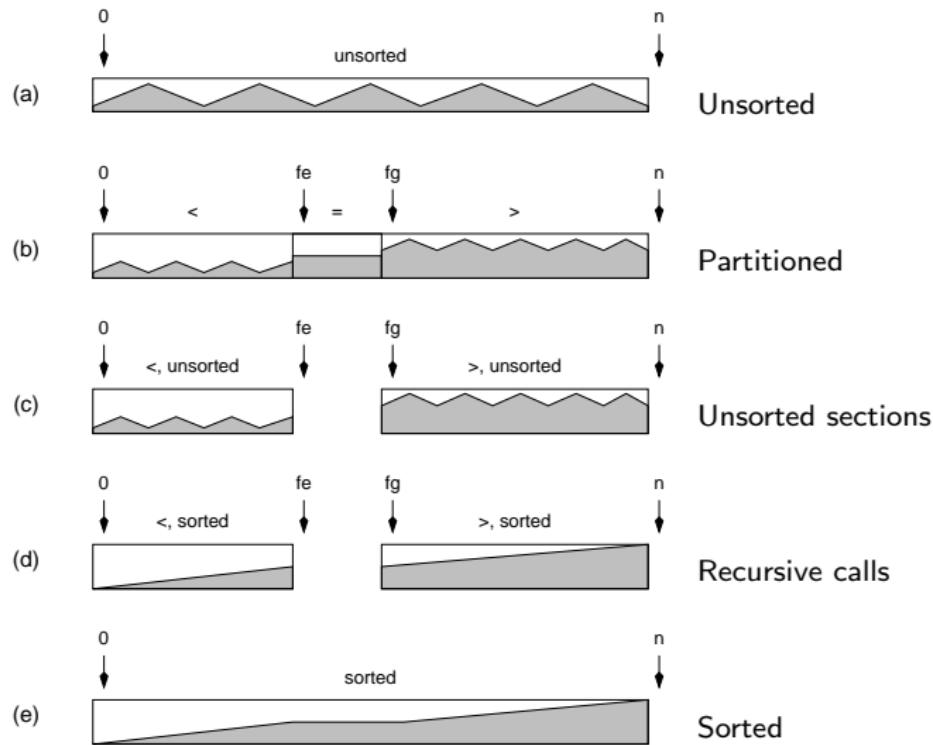
Today

Quick Sort

Quick Sort – Overview

- Quick Sort is a **divide and conquer** algorithm.
- Select an element p from the array A such that half of the elements in A are smaller, and the other half is larger.
- This element p is usually referred to as the **pivot** element.
- Next, we **partition** A such that all elements with **value** less than p are to the left of p , and larger elements are to the right.
- Recursively apply this partitioning to the resulting sub arrays.
- Put the sub arrays back together to get a solution for the initial array A .

Quick Sort – The idea



Quick Sort – Algorithm

```
// Call function with  $A[0, n - 1]$ 
procedure QUICKSORT( $A[l, r]$ ) // Array  $A$  of size  $r - l + 1$ 
    if  $l < r$  then
         $p \leftarrow \text{PARTITION}(A[l, r])$ 
        //  $p$  is the pivot that splits the array
        QUICKSORT( $A[l, p - 1]$ )
        QUICKSORT( $A[p + 1, r]$ )
    end if
end procedure
```

Quick Sort – Partitioning

```
procedure PARTITION( $A[l, r]$ )
     $p \leftarrow A[l]$  //  $p$  is called the pivot element
    while  $i \leq j$  do
        while  $A[i] < p$  do
             $i \leftarrow i + 1$  //  $i$  moves right
        end while
        while  $A[j] > p$  do
             $j \leftarrow j - 1$  //  $j$  moves left
        end while
        SWAP( $A[i], A[j]$ )
    end while
    SWAP( $A[i], A[j]$ ) // undo the last swap
    SWAP( $A[l], A[j]$ ) // pivot in right position
    return  $j$  // return pivot position to QUICKSORT
end procedure
```

Quick Sort – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	49	1	51	52	31	26	12	48	5	1	61	53	44	15	29

Quick Sort – Example

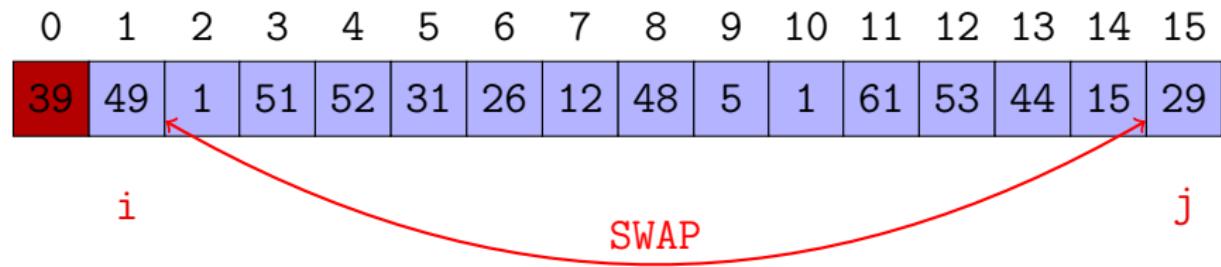
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	49	1	51	52	31	26	12	48	5	1	61	53	44	15	29

Quick Sort – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	49	1	51	52	31	26	12	48	5	1	61	53	44	15	29

i *j*

Quick Sort – Example

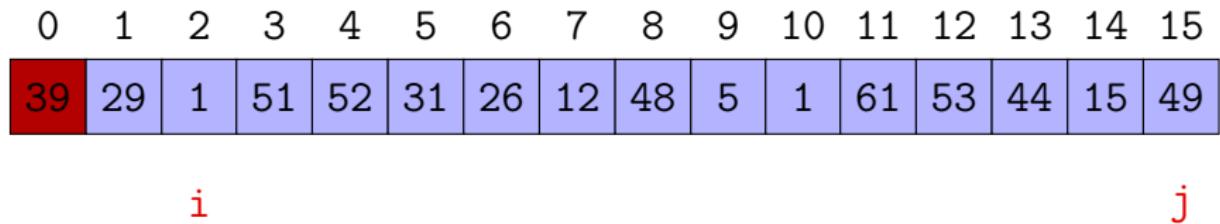


Quick Sort – Example

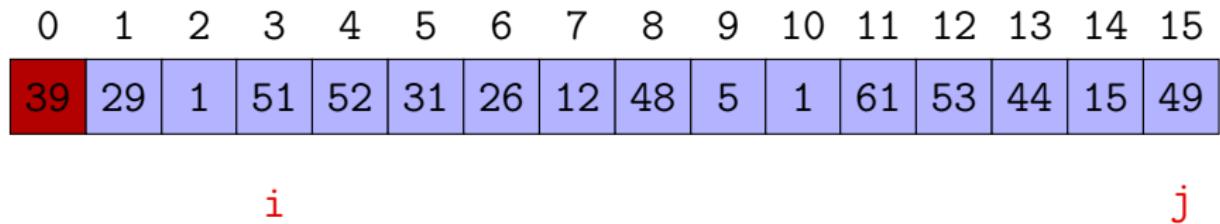
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	29	1	51	52	31	26	12	48	5	1	61	53	44	15	49

i *j*

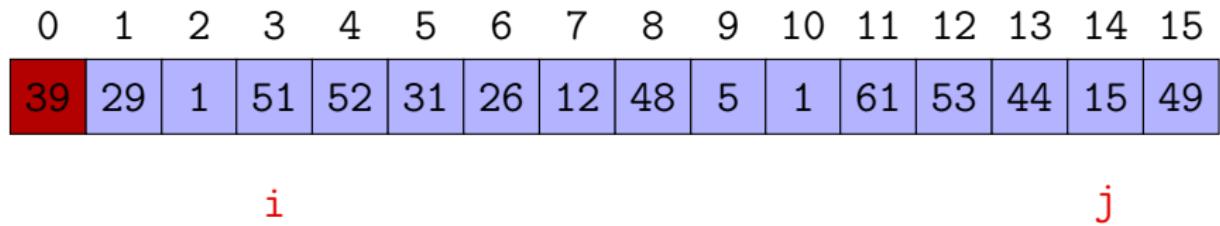
Quick Sort – Example



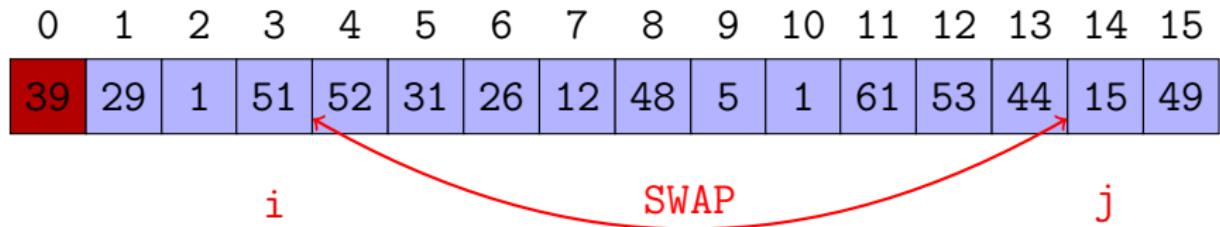
Quick Sort – Example



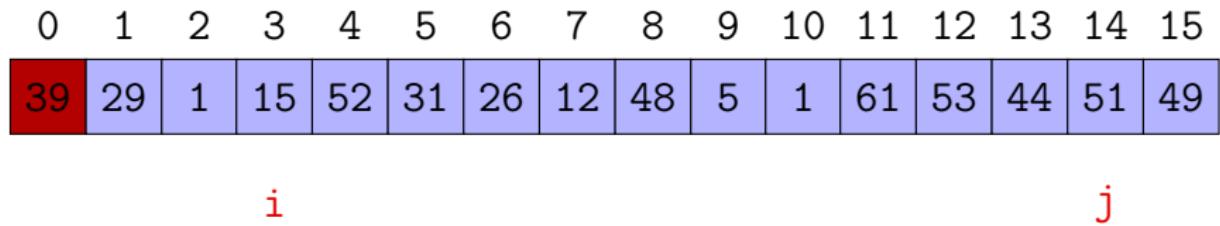
Quick Sort – Example



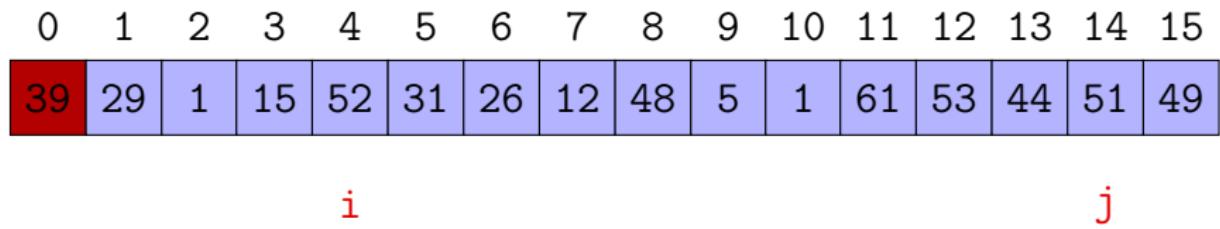
Quick Sort – Example



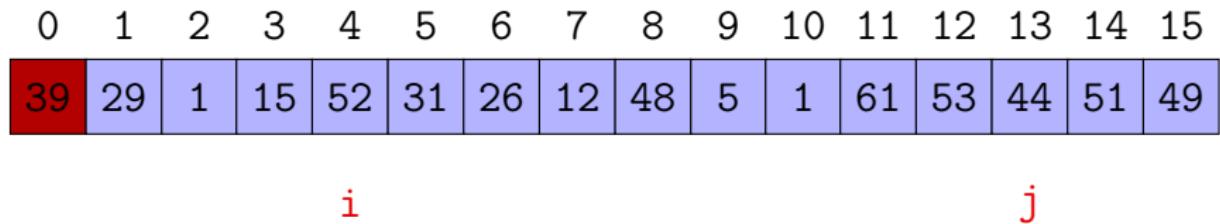
Quick Sort – Example



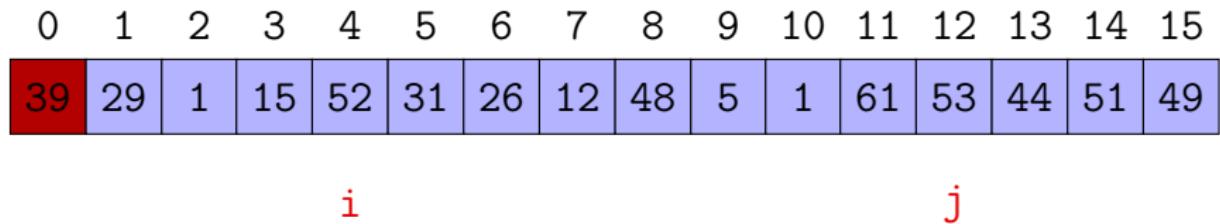
Quick Sort – Example



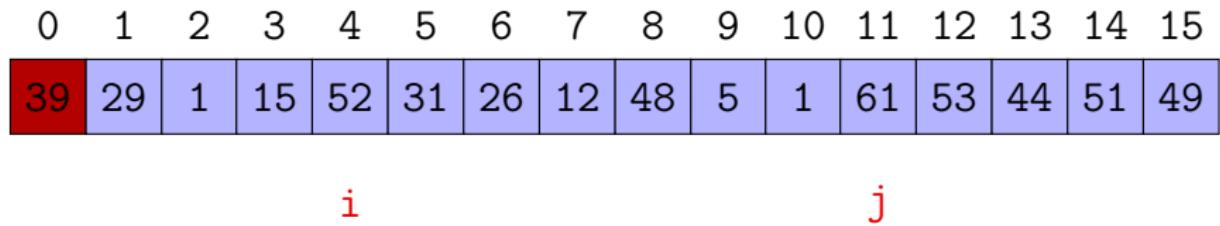
Quick Sort – Example



Quick Sort – Example



Quick Sort – Example

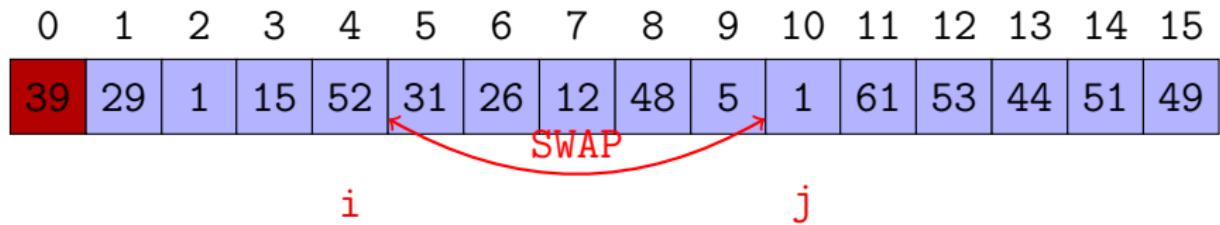


Quick Sort – Example

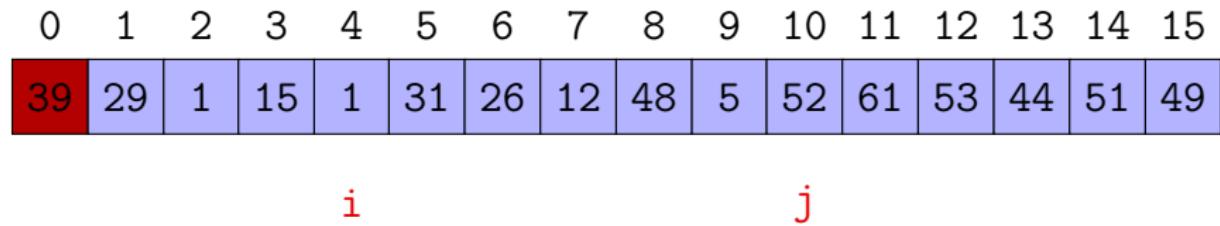
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	29	1	15	52	31	26	12	48	5	1	61	53	44	51	49

i j

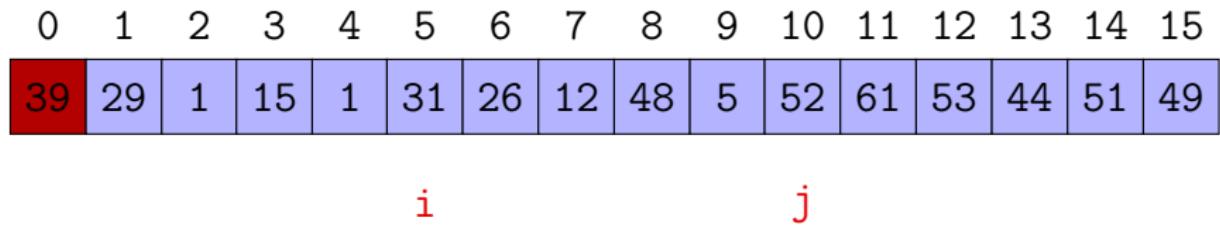
Quick Sort – Example



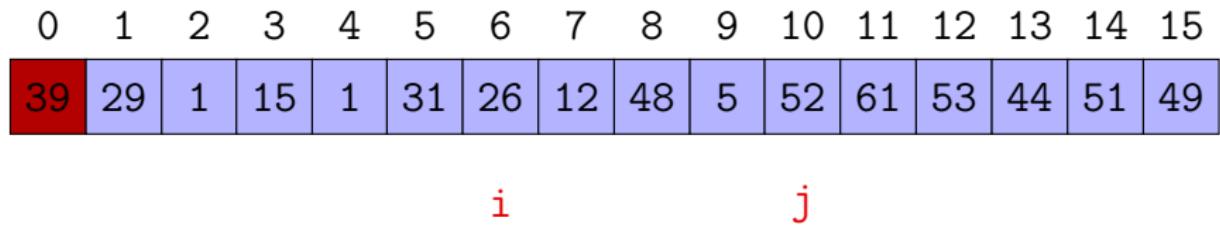
Quick Sort – Example



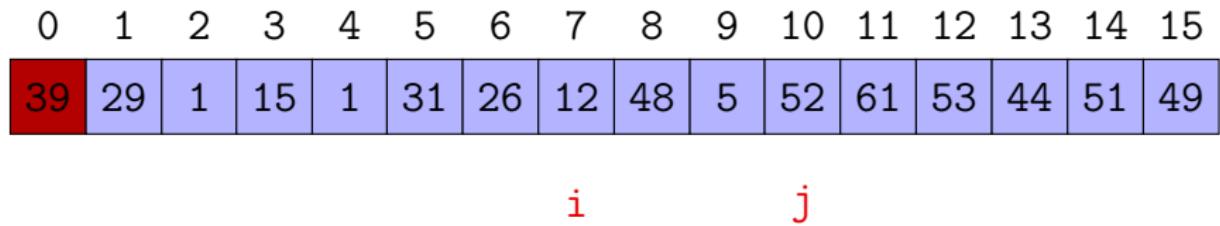
Quick Sort – Example



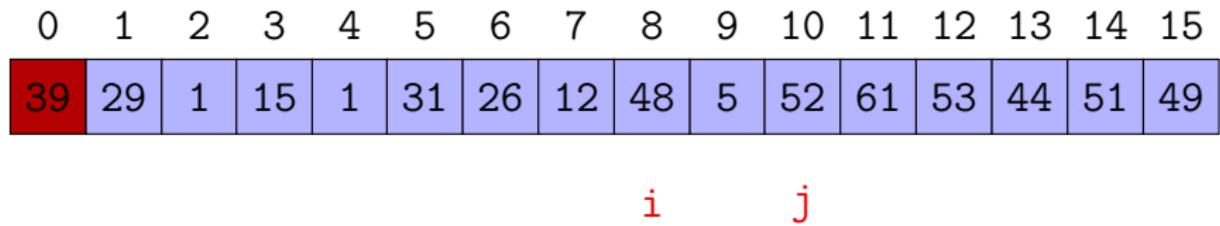
Quick Sort – Example



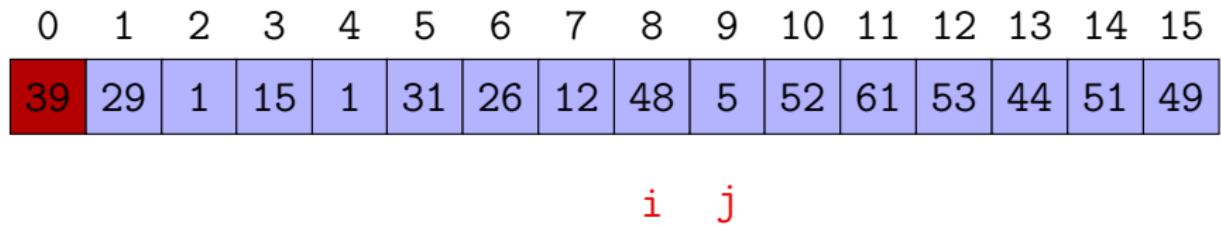
Quick Sort – Example



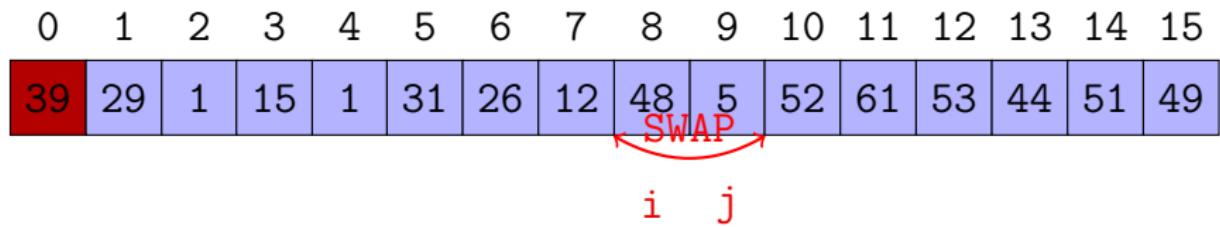
Quick Sort – Example



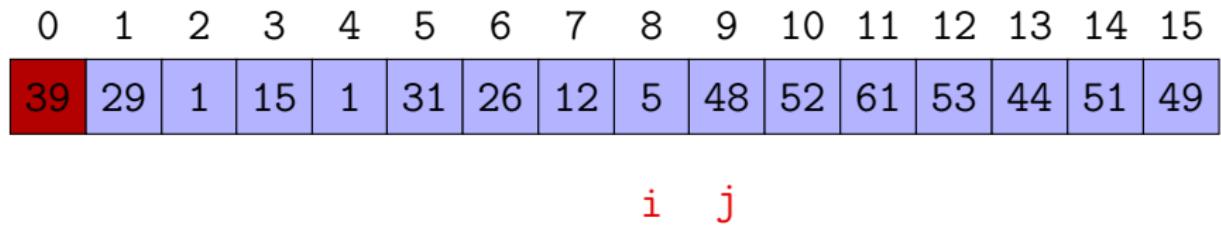
Quick Sort – Example



Quick Sort – Example



Quick Sort – Example

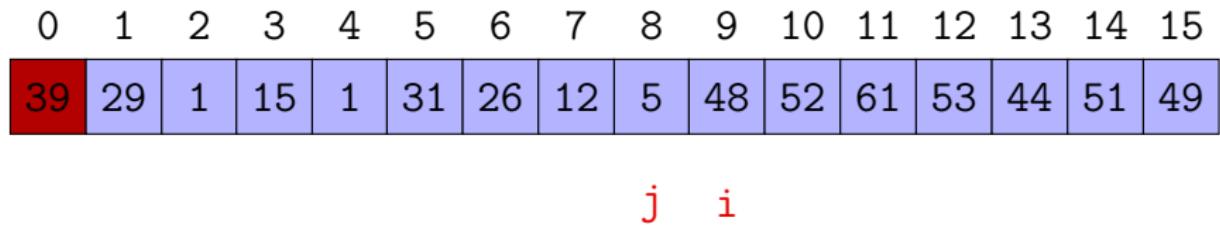


Quick Sort – Example

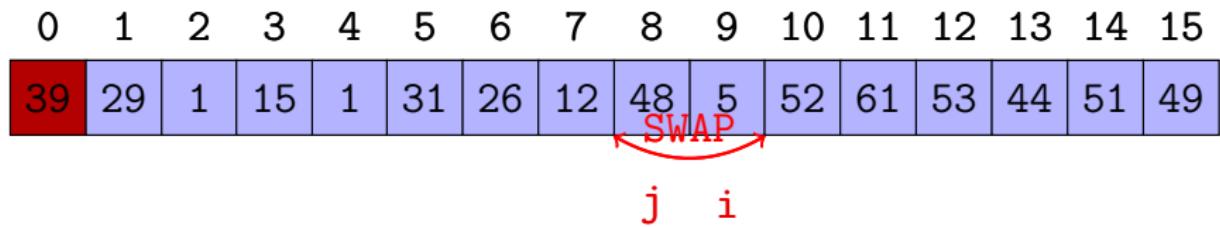
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	29	1	15	1	31	26	12	5	48	52	61	53	44	51	49

j
i

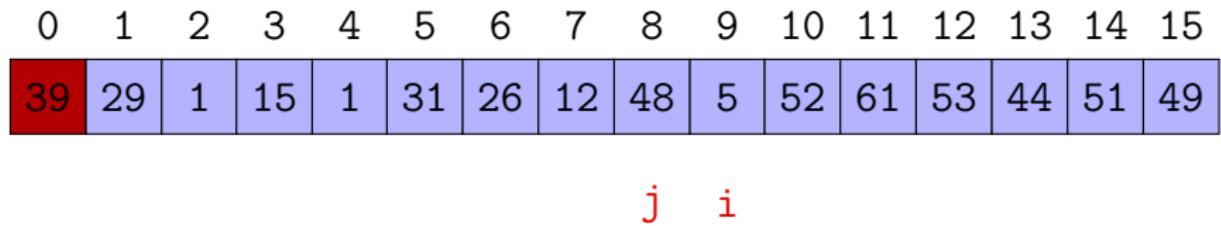
Quick Sort – Example



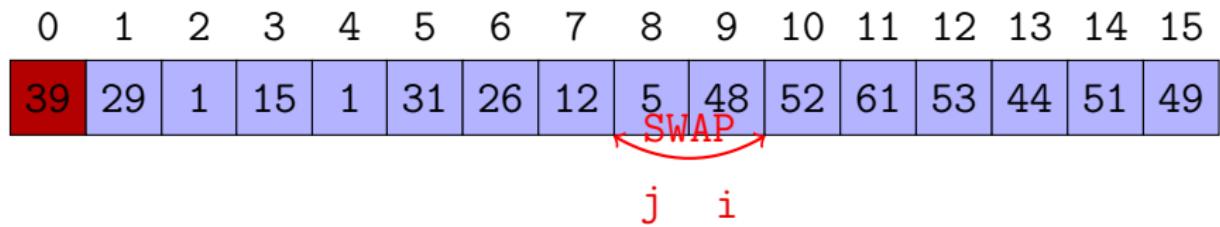
Quick Sort – Example



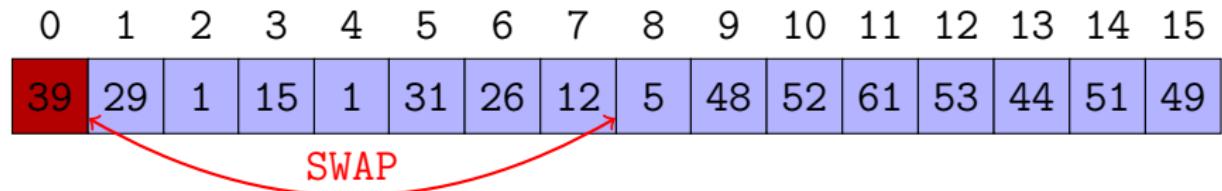
Quick Sort – Example



Quick Sort – Example



Quick Sort – Example



Quick Sort – Example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	29	1	15	1	31	26	12	39	48	52	61	53	44	51	49

$< p$

$> p$

Quick Sort – Analysis

- Most work is done during the partitioning process
- After the partitioning step, the **pivot** element is in the right place of the array
- Partitioning an array of size n is in $O(n)$
- Partitioning is **in place** and does NOT require extra space.

Quick Sort – Analysis

Hope for the best?

$$C(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + 2 \times C(\lfloor n/2 \rfloor) & \text{if } n > 1 \end{cases}$$

In this case, $C(n) \in O(n \log n)$. Wonderful...

Quick Sort – Analysis

Plan for the worst!

$$C(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ n + C(0) + C(n - 1) & \text{if } n > 1 \end{cases}$$

Now $C(n) \in O(n^2)$.

Awful...

qsort() function – An Example

```
1 #include <stdio.h>
2 #include <stdlib.h> /* -- qsort */
3
4 int values [] = {40,10,100,90,20,25};
5
6 int compare(const void *a, const void *b){
7     return (*(int*)a - *(int*)b);
8 }
9
10 int main(){
11     int n;
12     qsort (values, 6, sizeof(int), compare);
13     for (n=0; n<6; n++)
14         printf("%d ", values[n]);
15     printf("\n");
16     return 0;
17 }
```

Next

Balanced Search Trees



THE UNIVERSITY OF

MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

Today

Balanced Search Trees

Binary Search Trees

- Average case insertion and search: $O(\log n)$
- Worst case for both: $O(n)$

So, nice and simple, usually good enough, but not reliable.

A great deal of research effort has been invested in keeping binary search trees balanced.

Binary Search Trees

- How to get a binary search tree to stay balanced...
- or almost balanced...
- ... no matter what order the data are inserted.

Balanced Search Trees

- In a balanced (search) tree of n items:
 - Perfectly balanced tree, height = $\log n$, exactly
 - Balanced tree, height = $O(\log n)$
 - Search and Insert are $O(\log n)$
 - Building a balanced tree of n items is $O(n \log n)$

Balanced Search Trees

- During insertion there are mechanisms for making sure the tree does not grow unbalanced
- At the same time, the BST ordering is preserved
- So, search in a balanced tree is exactly the same as in a BST
- The only difference is that it is $O(\log n)$ worst case search / insertion complexity.

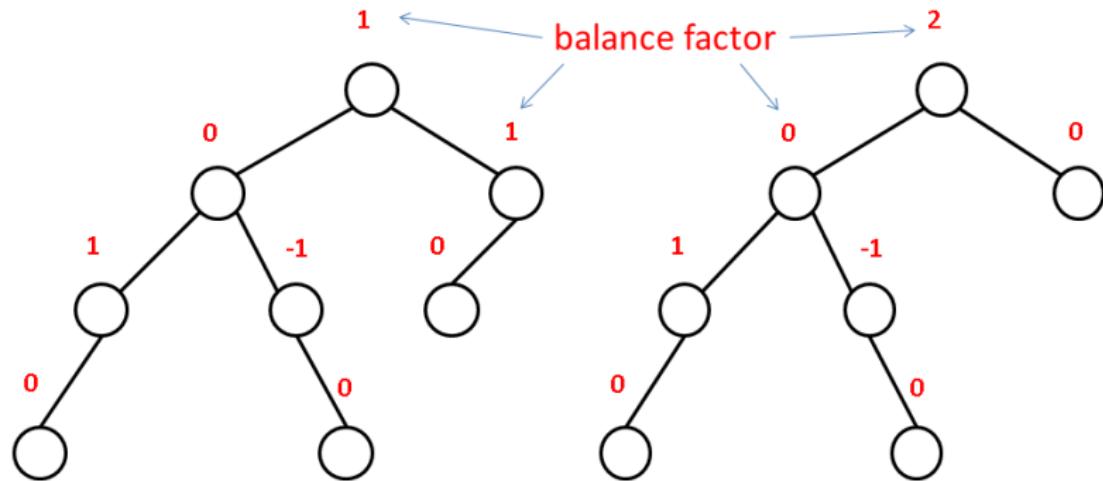
AVL Trees

- The first balanced tree
- Insert node + Keep track of height of subtrees of every node
- Balance node every time difference between subtree heights is > 1

Adelson-**V**elskii, G.; E. M. **L**andis (1962). "An algorithm for the organization of information". Proceedings of the USSR Academy of Sciences 146: 263–266. (Russian) English translation by Myron J. Ricci in Soviet Math. Doklady 3:1259–1263, 1962.

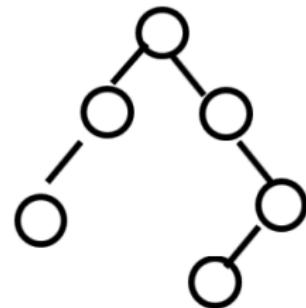
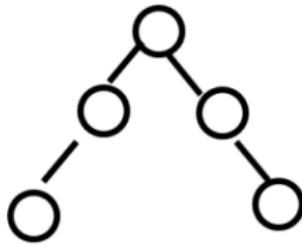
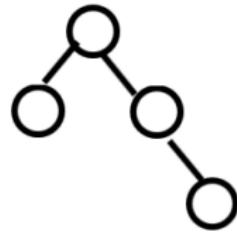
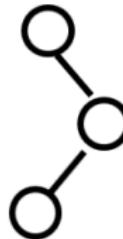
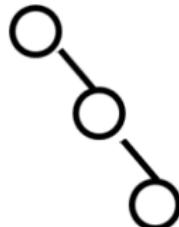
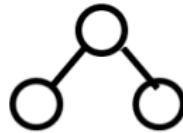
AVL Trees - balance factor

- **Balance factor** = height(left subtree) - height(right subtree)
- An **AVL tree** is a BST in which the balance factor of every node is either 0 or 1 or -1.



AVL Trees

Do these trees satisfy the AVL condition? Why / why not?



AVL Trees - Correcting operation: Rotation

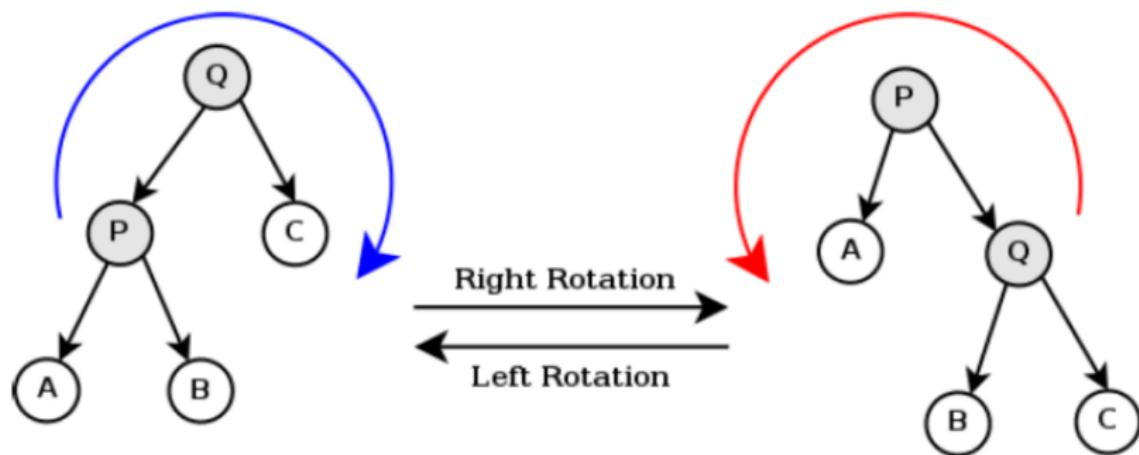
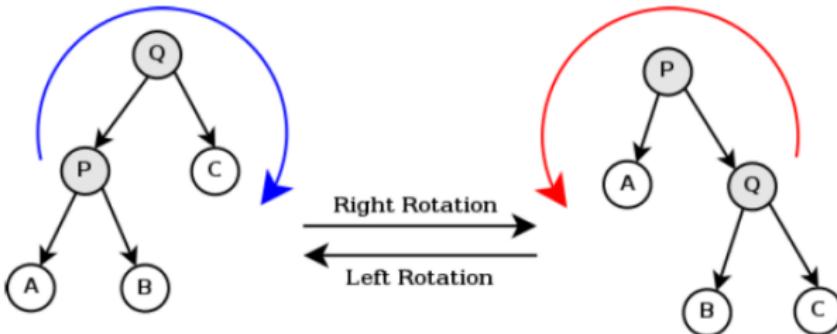


Image from Wikipedia: Tree rotation

AVL Trees - Correcting operation: Rotation



```
RotateR(node)
{
    left = node.Left;
    leftRight = left.Right;
    parent = node.Parent;

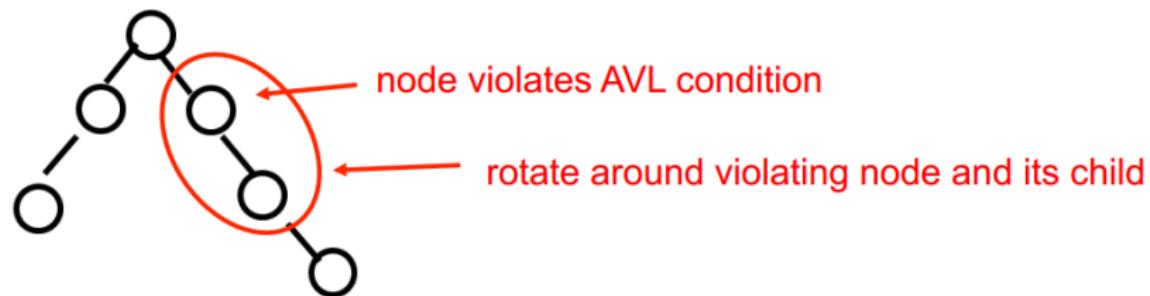
    left.Parent = parent;
    left.Right = node;
    node.Left = leftRight;
    node.Parent = left;
}
```

```
RotateR(Q)
{
    left = P;
    leftRight = B;
    parent = NULL;

    P.Parent = NULL;
    P.Right = Q;
    Q.Left = B;
    Q.Parent = P;
}
```

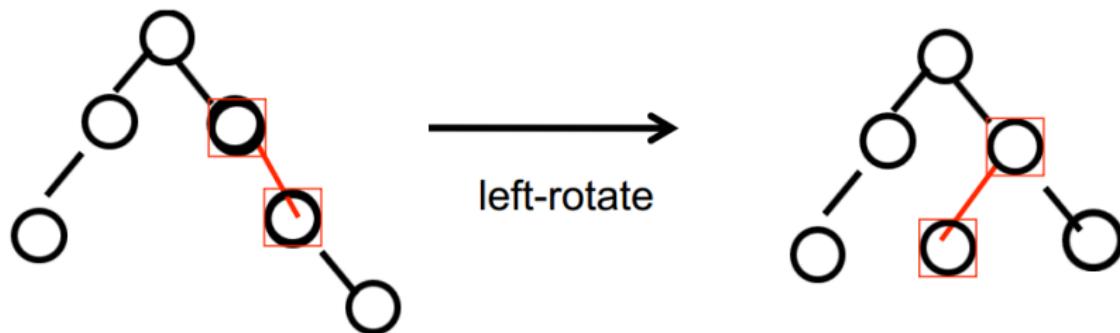
AVL Trees - Rotations

How does rotation help balance a tree?



AVL Trees - Rotations

How does rotation help balance a tree?



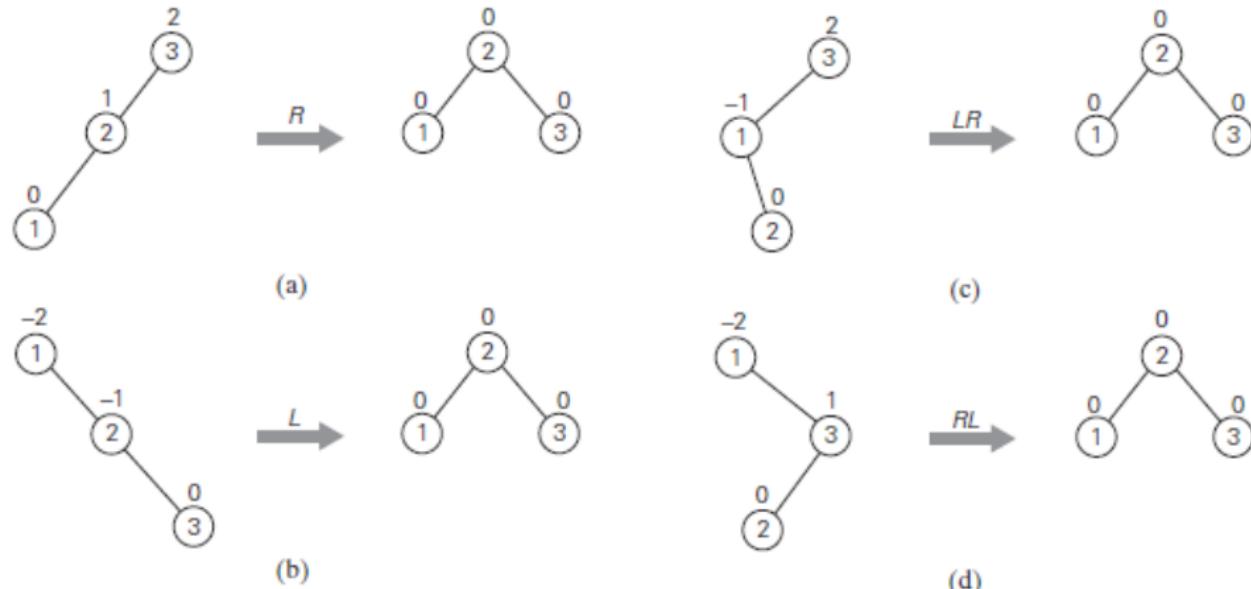
AVL Trees - Rotation Complexity

```
node* insert ( node* tree, node* new_node )
{
    if ( tree == NULL )
        tree = new_node;
    else if ( new_node->key < tree->key ) {
        tree->left = insert ( tree->left, new_node );
        /* Fifty lines of left balancing code */
    }
    else {
        tree->right = insert ( tree->right, new_node );
        /* Fifty lines of right balancing code */
    }
    return tree;
}
```

AVL Trees

- Note that since rotations preserve the bst ordering of the tree, search is the same as for a BST
- In total 4 different types of rotations
- Tree is always reasonably balanced
- Very fiddly to code: must keep track of insertion path and size of all subtrees
- Balancing adds time (but constant time)

AVL Trees - 4 Types of Rotations



Four rotation types: (a) R-rotation (b) L-rotation (c) LR-rotation (d) RL-rotation

Image from the book: Introduction to the Design and Analysis of Algorithms, 3rd Edition, Anany Levitin

AVL Trees - Interactive Demo!

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

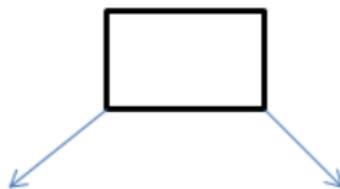
2,3-Trees: Overview

- Trees do not have to be binary!
- Nodes in 2,3-Trees have
 - 1 or 2 keys
 - 2 or 3 pointers, correspondingly
- When a node needs to contain more than 2 keys – split to accommodate new items
- All leaves are at the same level.
- All data is kept in sorted order.

2,3-Trees nodes

- Every node is a 2-node or a 3-node.

2-node

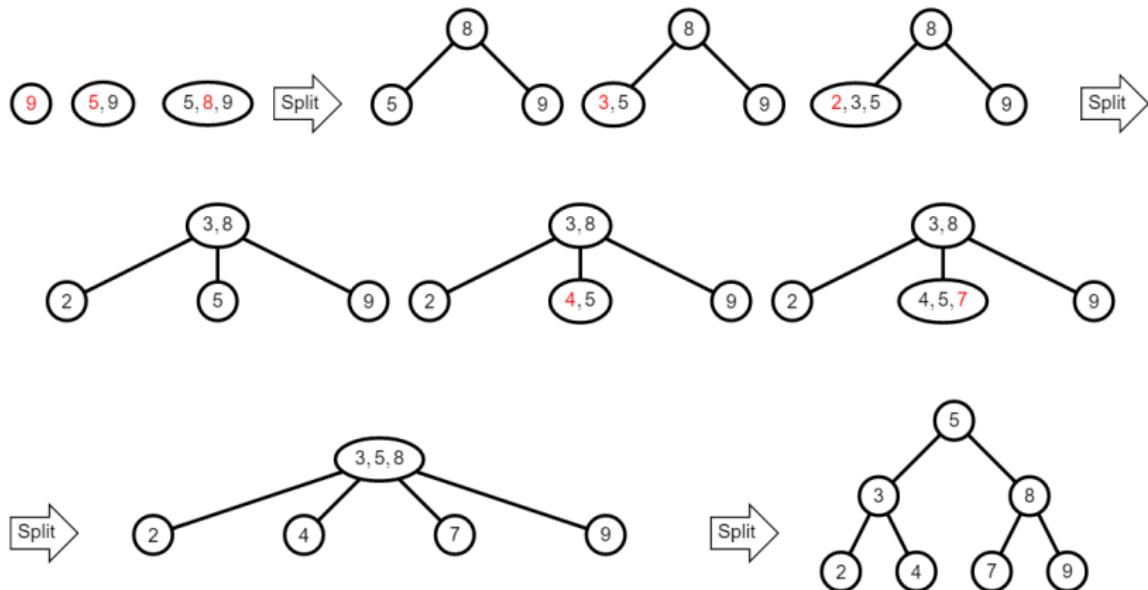


3-node



2,3-Trees construction

Construct a 2-3 tree for the sequence 9, 5, 8, 3, 2, 4, 7.



Many Other Kinds of Balanced Search Trees

- Red-Black trees
- 2,3,4-Trees
- B-Trees
- B⁺-Trees (widely used in practise, especially database systems)

Balanced Search Trees - Summary

- Modify the BST to ensure worst case $O(n)$ search does not occur
- Use rotations or multi-key nodes to ensure depth of $O(\log n)$
- Many different kind of balanced trees exist

Next

Algorithmic and numerical limits



THE UNIVERSITY OF

MELBOURNE



THE UNIVERSITY OF
MELBOURNE

ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne

Slides developed by Prof Rui Zhang and Dr Matthias Petri

Algorithmic and Numerical Limits

Time and Space Tradeoffs

Often it is possible to “trade” the run time performance of an algorithm for space usage.

Allowing an algorithm or data structure to use more space, can make it more efficient to solve certain problems.

Time and Space Tradeoffs - Examples

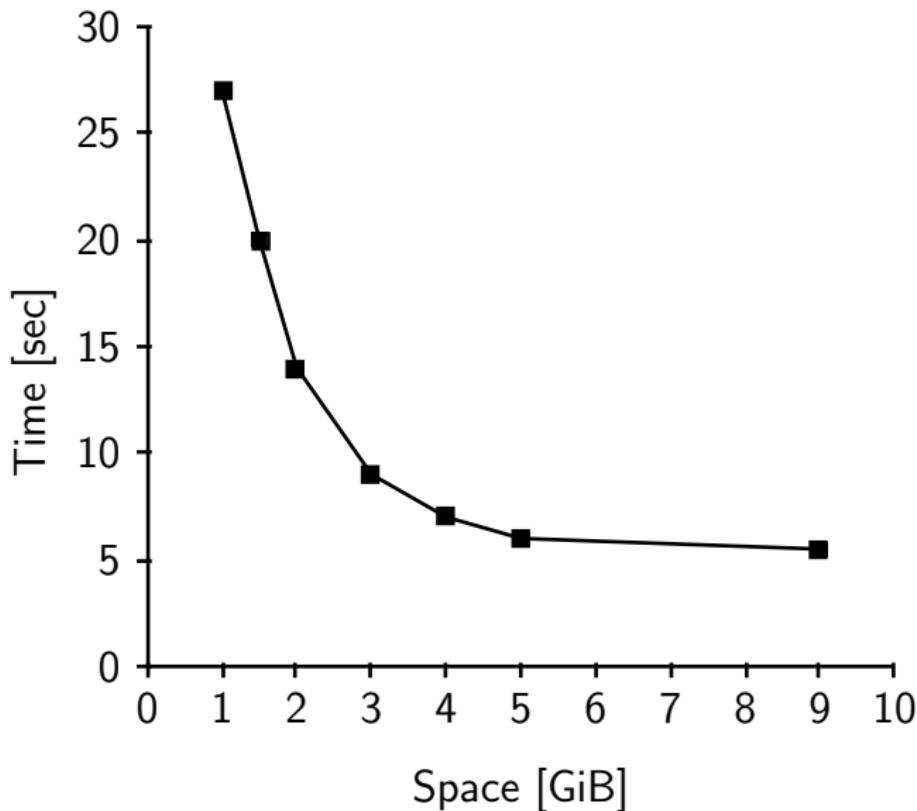
- Adding a “prev” pointer to the `node_t` structure of a linked list
- Precompute and store frequently used `sin()` or `cos()` results
- Build a search index if lots of searches are performed over an input text

Often the storage and construction cost can be **amortized** over many operations. E.g., building a tree may double the cost of adding or deleting every data item, but makes searching much more efficient.

Time and Space Tradeoffs - Pattern Matching

- Search for one pattern often: Precompute a “skip” table for the pattern
<http://whocouldthat.be/visualizing-string-matching/>
- Search for many patterns in one text: build a search index

Time and Space Tradeoffs - Summary



Algorithmic Limits

- Bound the efficiency of **any** solution for a particular problem.
- What are computers capable of?
- How good is our new algorithm?
- Should I invest time coming up with a faster algorithm?

Lower Bounds

An estimate on the minimum amount of work needed to solve a given **problem**.

- Minimum amount of work needed to solve a problem
- We are not talking about specific algorithms anymore

Lower Bounds - Example

- How many comparisons are needed to find the smallest element in a set of n numbers?
- How many comparisons are needed to sort an array of size n
- How many multiplications are needed to multiply a $n \times n$ matrix?

Lower Bounds - Tightness

A lower bound is considered **tight** if there exists an algorithm with the same efficiency as the lower bound.

We use the efficiency class $\Omega()$ to describe lower bounds. $\Omega()$ bounds the growth of a function from below, whereas $O()$ bounds the growth from above.

Examples

Problem	Lower Bound	Tight?
Sorting	$\Omega(n \log n)$	Yes
Find Min	$\Omega(n)$	Yes
Search Sorted Array	$\Omega(\log n)$	Yes
$n \times n$ Matrix Multiplication	$\Omega(n^2)$	No

Hard problems

Is there a subset of the following numbers that adds up to exactly 1,000?

34, 38, 39, 43, 55, 66, 67, 84, 85, 91,
101, 117, 128, 138, 165, 168, 169, 182, 184, 186,
234, 238, 241, 276, 279, 288, 386, 387, 388, 389,
413, 444, 487, 513, 534, 535, 616, 722, 786, 787

In general, given n integer values, and a target value k , determine if there is a subset of the integers that adds up to exactly k . This is often referred to as the **subset sum** problem.

Subset sum algorithm

Evaluate all subsets of the n items, and if any one of them adds to k , return **yes**.

How many subsets are there? A set with n elements has 2^n subsets. Exponentially many!

So the runtime of an algorithm that checks all subsets is $O(2^n)$ - exponential!

Hard problems

- The **subset sum** problem is an example of a very important class of related hard problems.
- All (so far!) take **exponential** time in size of input, which makes them **intractable**. We can solve for small problems, and we can quickly check any proposed solution for large problems. But we can't find solutions for large problems. If any one of these problems can be solved in **polynomial** time, then **they all can!**
- Sometimes it is good that a problem is hard. Think Encryption.

Numerical Limits - Integers

- Most integer number representations consume a fixed amount of storage space
- For example, the `int` type may consume 32 or 64 bits
- More specific types such as `int32_t` or `uint64_t` precisely define the space used by the data type

Unsigned Integers

- Computers store integers numbers in a binary representation
- The number of bits a data type occupies determines the range of numbers it can represent
- Example: `uint32_t x = 1,234,567,890;` is represented as the 32 bit binary sequence
0100 1001 1001 0110 0000 0010 1101 0010

Unsigned Integers - Limits

- What are the limits of a given data type?
- What happens if you try to go beyond this limit?

Unsigned Integers - Limits

```
1 #include <stdio.h>
2 #include <stdint.h>
3
4 int main() {
5     uint16_t x = 0;
6     while( 1 ) {
7         printf("x=%u\n", x);
8         x = x + 1;
9     }
10 }
```

What will this program output?

Unsigned Integers - Limits

- Maximum: $2^b - 1$ where b is the bit length of the integer representation
- Minimum: 0
- Wraps around on overflow

Type	Smallest Value	Largest Value
uint8_t	0	255
uint16_t	0	65,535
uint32_t	0	4,294,967,295
uint64_t	0	18,446,744,073,709,551,615

Signed Integers - Limits

- Maximum: $2^{b-1} - 1$ where b is the bit length of the integer representation
- Minimum: -2^{b-1}
- One bit used as a “sign” bit

Type	Smallest Value	Largest Value
int8_t	-128	127
int16_t	-32,768	32,767
int32_t	-2,147,483,648	2,147,483,647

Numerical Limits - Floating Point

- Computer memory is limited.
- We can not store numbers with infinite precision
- How much accuracy is needed?
- How many integer digits and how many fraction digits?

Numerical Limits - Floating Point

- To an engineer building a highway, it does not matter whether it's 10 meters or 10.0001 meters wide - his measurements are probably not that accurate in the first place.
- To someone designing a microchip, 0.0001 meters (a tenth of a millimeter) is a huge difference - But he'll never have to deal with a distance larger than 0.1 meters.
- A physicist needs to use the speed of light (about 300000000) and Newton's gravitational constant (about 0.0000000000667) together in the same calculation.

Floating Point - Concepts

Four components of a floating point representation:

- The **sign** (positive or negative number)
- The **significand** or **mantissa** containing the actual digits of the number
- The **base** and **exponent** that describe where the decimal (or binary) point is placed relative to the beginning of the significand. For example 10^4 or 2^{-20} .
- Positive exponents represent large numbers, whereas negative exponents are used to represent numbers close to zero.

Examples: 5.23125×10^{15} or 1.43254×2^{-5}

Floating Point - Examples

Sign	Significand	Base	Exponent	Scientific notation	Fixed point notation
+	5.23123	10	8	5.23123×10^8	523123000
-	1.234	10	-6	-1.234×10^{-6}	-0.000001234
-	6941.5	2	-12	-6941.5×2^{-12}	-0.2118378
+	11011101 ₂	2	-12	221×2^{-12}	0.006744385

Floating Point - Decimal Fractions

Decimal number: 53127.4592

10^4	10^3	10^2	10^1	10^0	10^{-1}	10^{-2}	10^{-3}	10^{-4}
5	3	1	2	7	4	5	9	2

$$\begin{aligned} 53127.4592 &= 5 \times 10^4 \\ &\quad + 3 \times 10^3 \\ &\quad + 1 \times 10^2 \\ &\quad + 2 \times 10^1 \\ &\quad + 7 \times 10^0 \\ &\quad + 4 \times 10^{-1} \\ &\quad + 5 \times 10^{-2} \\ &\quad + 9 \times 10^{-3} \\ &\quad + 2 \times 10^{-4} \end{aligned}$$

Floating Point - Binary Fractions

Binary number: 11101.1001

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	1	1	0	1	1	0	0	1

$$\begin{aligned} 11101.1001 &= 1 \times 2^4 \\ &\quad + 1 \times 2^3 \\ &\quad + 1 \times 2^2 \\ &\quad + 0 \times 2^1 \\ &\quad + 1 \times 2^0 \\ &\quad + 1 \times 2^{-1} \\ &\quad + 0 \times 2^{-2} \\ &\quad + 0 \times 2^{-3} \\ &\quad + 1 \times 2^{-4} \end{aligned}$$

$$= 29.5625_{10}$$

Floating Point - The IEEE 754 Standard

- Standardizes the way computers represent floating point numbers
- Standardizes basic operations such as addition, multiplication so every machine and compiler generate the same results during computation
- Widely used on most machines and programming languages out there (graphic cards, phones, CPUs, C++ etc)

Floating Point - The IEEE 754 Standard

Single Precision (float) type:

- 32 bits in total storage cost
- 1 bit to store the sign
- 23 bits to store the significand
- 8 bits to store the exponent
- all base 2

Floating Point - The IEEE 754 Standard

Single Precision (double) type:

- 64 bits in total storage cost
- 1 bit to store the sign
- 52 bits to store the significand
- 11 bits to store the exponent
- all base 2

Floating Point - Limits float

Largest positive floating point number (full significand, largest exponent): $1.99999988 \times 2^{127} \approx 3.4 \times 10^{38}$

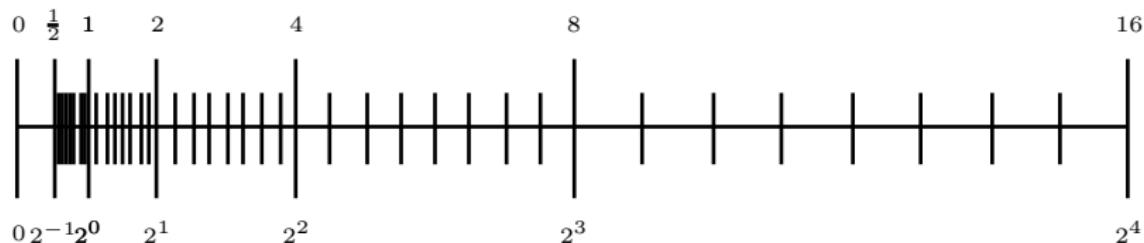
Smallest positive floating point number (empty significand, smallest exponent): $1.0 \times 2^{-126} \approx 1.2 \times 10^{-38}$

Largest negative floating point number (full significand, largest exponent): $-1.99999988 \times 2^{127} \approx -3.4 \times 10^{38}$

Smallest negative floating point number (empty significand, smallest exponent): $-1.0 \times 2^{-126} \approx -1.2 \times 10^{-38}$

Floating Point - Limitations

For example, using a floating point type that has only 3 bits of precision (the size of the significand). Then, between any two powers of 2, there are $2^3 = 8$ representable numbers:



However, the difference between two number changes as the exponent increases.

Floating Point - Limitations

While we can use floating point numbers to represent very large and very small numbers, there are limits.

For example, given a number $z = 2147483042$, the next larger number that can be represented using a ~~double~~ is
2147483008.00 float

The gaps between the numbers represented by a double depend on the magnitude of the number itself. Large number, large gaps; Small number, small gaps.

Summary

- Often we can trade runtime performance for space usage to change the performance of algorithms
- Certain problems have a lower bound, on the amount of work that needs to be performed to solve the problem
- There exist many problems which we do not know (and some think there exist none) any efficient algorithms
- Integer types can represent a limited range of values
- Performing operations on floating point numbers of different magnitude can lead to unexpected outcomes

Next

PART 2



THE UNIVERSITY OF

MELBOURNE



ENGR30003 Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne
Slides developed by Prof Richard Sandberg

This week

LECTURE 11/12

Root finding

Why Programming?

Many engineering problems can be cast into a set of equations.
Take example of [fluid flows](#):



[Figure](#): Examples of high-speed vehicles for which accurate prediction of flow crucial.

Why Programming?

Many engineering problems can be cast into a set of equations.

Fluid flows are represented by the [Navier–Stokes equations](#):

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_k} (\rho u_k) = 0 , \quad (1)$$

$$\frac{\partial}{\partial t} (\rho u_i) + \frac{\partial}{\partial x_k} [\rho u_i u_k + p \delta_{ik} - \tau_{ik}] = 0 , \quad (2)$$

$$\frac{\partial}{\partial t} (\rho E) + \frac{\partial}{\partial x_k} [\rho u_k H + q_k - u_i (\tau_{ik} - \rho \sigma_{ik})] = 0 , \quad (3)$$

$$p = \frac{\rho T}{\gamma M^2} , \quad \tau_{ik} = \frac{2\mu}{Re} \left(S_{ik} - \frac{1}{3} S_{jj} \delta_{ik} \right) , \quad S_{ik} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_k} + \frac{\partial u_k}{\partial x_i} \right)$$

Total enthalpy H and total energy E are

$$H = E + p/\rho, \quad E = T/[\gamma(\gamma - 1)M^2] + 1/2u_i u_i, \quad \gamma = 1.4.$$

$$q_k = -\frac{\kappa}{PrEcRe} \frac{\partial T}{\partial x_k} .$$

Why Programming?

Navier–Stokes equations are a system of five non-linear and coupled equations.

No general analytical solution exists (some exact solutions exist for very simple special cases, defined by B.C.s).

⇒ Solve set of equations numerically, abiding to requirements:

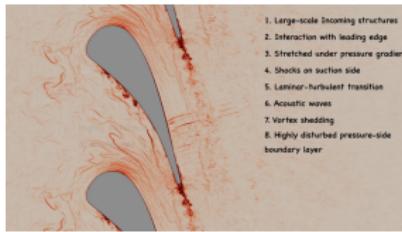
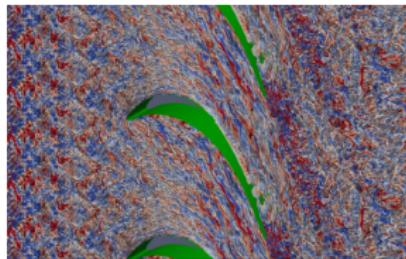
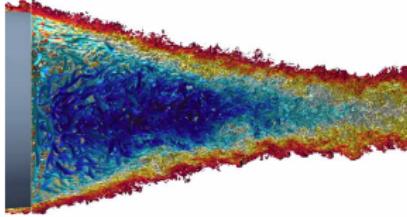
1. accuracy and consistency of method
2. minimize amplitude and phase errors
3. scheme must be stable
4. scheme must be efficient (ratio accuracy/comp cost)

Why Programming?

If you can write code to solve these equations
(and have access to massive HPC 😊):



Examples of simulations using Melbourne research code:



Back to basics

Over the next 7 weeks, we will cover the following topics:

1. Root finding
2. Systems of Linear Equations (Direct/Iterative methods)
3. Least Squares
4. Interpolation
5. Differentiation/Integration
6. ODEs
7. Regression

Root finding

Why is root finding important?

Consider following example:

A skydiver jumps out of an airplane. Newton's law:

$$\text{net force on an object} = m \frac{dv}{dt}. \quad (4)$$

m is mass of skydiver, v their velocity and t time.

For mass in free fall,

$$\text{net force} = \text{weight of mass} + \text{drag on mass} \quad (5)$$

$$= mg - Dv \quad (6)$$

Therefore

$$\frac{dv}{dt} = g - \frac{D}{m}v \quad (7)$$

Root finding

The solution to this ODE is:

$$v(t) = \frac{gm}{D} \left(1 - e^{-\frac{Dt}{m}}\right) \quad (8)$$

for $v = 0$ at $t = 0$.

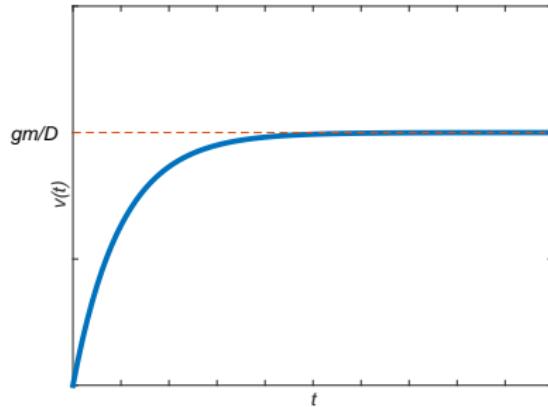


Figure: Velocity as function of time.

Root finding

$$v(t) = \frac{gm}{D} \left(1 - e^{-\frac{Dt}{m}}\right)$$

Suppose we want to find the value of drag, D , so that mass m will reach a velocity v at time t .

We cannot simply rearrange the equation to solve for D as it shows up multiple times.

$$f(D) = \frac{gm}{D} \left(1 - e^{-\frac{Dt}{m}}\right) - v(t).$$

Then we can find a solution for D by setting $f(D) = 0$.

In numerical analysis this is called **finding the roots of an equation**. Different methods to find the roots of equations will be discussed in the following.

Root finding

The methods of solving roots of equations can be broadly classified as

1. Graphical methods
2. Bracketing methods
3. Open methods

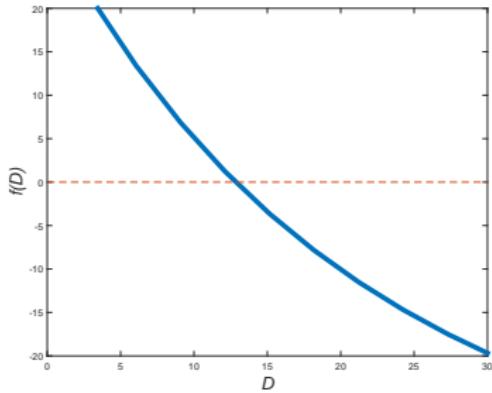
All these methods have their advantages and disadvantages.

Root finding - Graphical Method

One crude way of finding roots for an equation: plot the graph and visually inspect where function evaluates to zero.

E.g., plot $f(D)$ and find value of D where $f(D) = 0$.

Using: $m = 100\text{kg}$, $t = 8\text{s}$, $v = 50\text{m/s}$, $g = 10\text{m/s}^2$ we get



⇒ the root of $f(D)$ occurs for D close to 13.

The exact solution is $D = 12.8396261$.

Root finding - Graphical Method

Graphical method is **not very practical** because

- accuracy of solution is very poor
(depends on how accurately you can read off the graph !)
- inefficient if need to solve $f(D) = 0$ for different m, g and t

BUT: the graphical method has **advantages**

- you can see exactly how function behaves
- can quickly see how many roots function has
- can be used as initial guess for other methods

Root finding - Bracketing Method

There are two different Bracketing Methods. They are

- Bisection Method
- Method of false position

These methods are known as bracketing methods because they rely on having **two initial guesses**:

x_l : lower bound

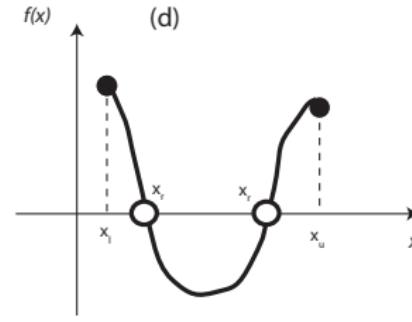
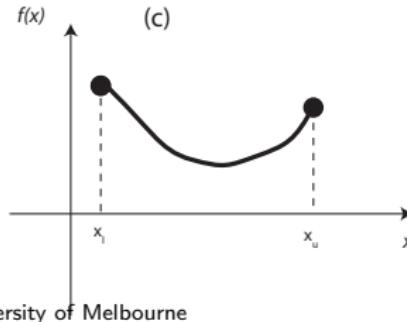
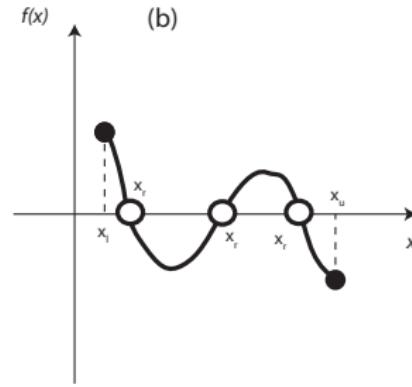
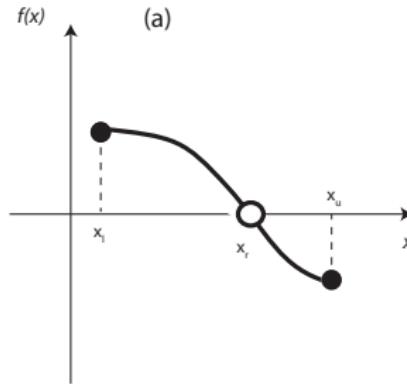
x_u : upper bound.

x_l and x_u **must bracket** (be either side of) the root.

This requirement in many cases can be hard to meet, as illustrated in the next figure.

Root finding - Bracketing Method

Illustration of various situations that can occur when bracketing the roots



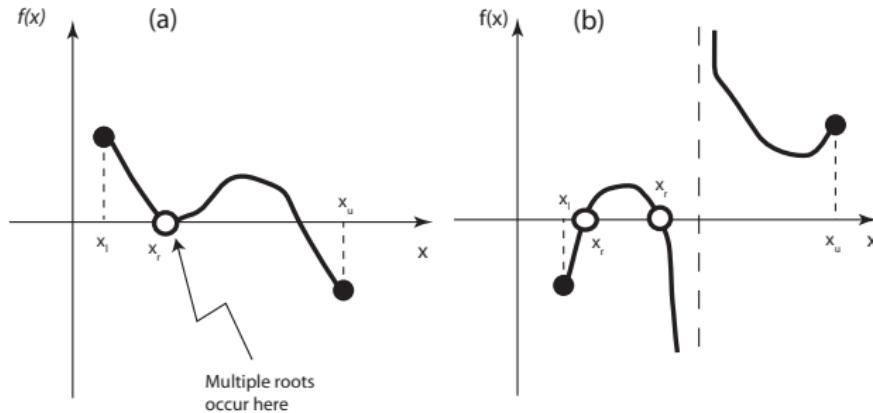
Root finding - Bracketing Method

- (a) If $f(x_u)$ and $f(x_l)$ are of different sign
 \Rightarrow there must be at least one root, x_r , between x_u and x_l ,
 i.e. $x_l < x_r < x_u$.
- (b) if $f(x_u)$ and $f(x_l)$ are of different sign
 \Rightarrow there might be an odd number of roots.
- (c) if $f(x_u)$ and $f(x_l)$ are of the same sign
 \Rightarrow there might not be any root between x_u and x_l .
- (d) if $f(x_u)$ and $f(x_l)$ are of the same sign
 \Rightarrow there might be multiple roots (even number) between
 x_u and x_l .

Root finding - Bracketing Method

There are, of course, **exceptions** to the rules above:

- When the function is tangential to the x -axis, multiple roots can occur
- In regions close to discontinuities, there might also be exceptions to the above rules.



Bracketing Methods - Bisection Method

One popular method for finding roots is the **Bisection Method**:

The Bisection Method uses the following steps:

1. Obtain two guesses for the root, x_l and x_u , that “brackets” the root, x_r . These two guesses can be found by plotting a graph of $f(x)$ and see where approximately it crosses the x axis. In general, if

$$f(x_u)f(x_l) < 0$$

then you can be reasonably sure that you have at least one root between x_l and x_u .

2. Estimate the root of $f(x)$ to be

$$x_r' = \frac{x_l + x_u}{2}$$

Bracketing Methods - Bisection Method

3. Make a choice:

- If $f(x_r') = 0$, then x_r' is the root, i.e. $x_r' = x_r$.
Stop, you found your root!
- If $f(x_l)f(x_r') < 0$, then root is in lower subinterval.
Set $x_u = x_r$ and go back to STEP 2.
- If $f(x_u)f(x_r') < 0$, then root is in upper subinterval. Set
 $x_l = x_r$ and go back to STEP 2.

You usually cannot get exactly $f(x_r) = 0$. In practice, if $|f(x_l)f(x_r')| < \epsilon$, where ϵ is a small value (the stopping criterion), you would take x_r' to be the root.

Exercise

Exercise 1

Use Bisection method to obtain root of following equation:

$$f(D) = \frac{1000}{D} \left(1 - \exp \left(-\frac{8D}{100} \right) \right) - 50$$

Exercise

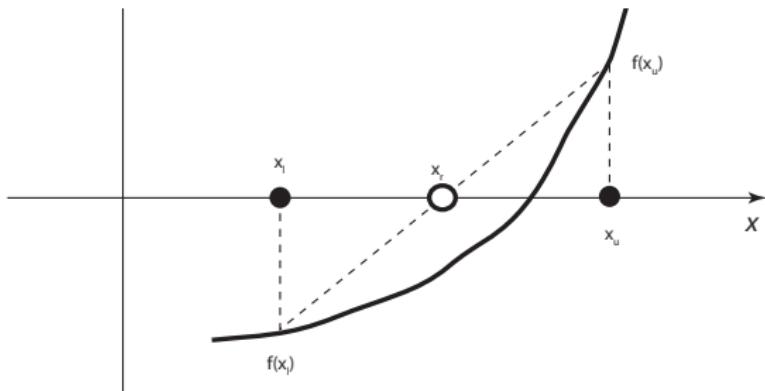
```
1  eps      = 1.0E-6 /* accuracy of your answer */
2  maxiter= 50;      /* maximum number of iterations*/
3  Dl = 10.0;        /* Initial lower guess of root*/
4  Du = 20.0;        /* Initial upper guess of root*/
5
6  for i = 1:maxiter,
7      Dr = (Dl + Du)/2.; /* Guess that root is between Dl and Du */
8      F1 = (1000/Dl)*(1-exp(-(8*Dl/100)))-50;
9      Fu = (1000/Du)*(1-exp(-(8*Du/100)))-50;
10     Fr = (1000/Dr)*(1-exp(-(8*Dr/100)))-50;
11
12    /* testing to see if we have found the root. If the root is found then exit the for loop */
13    if abs(Fr) < eps
14        break;
15    end
16
17    /* Algorithm to find out whether the upper or lower guess is closer to the actual root */
18    if (Fr*F1)<0
19        Du=Dr;
20    elseif (Fr*Fu)<0
21        Dl=Dr;
22    else
23        break;
24    end;
25 end
```

Write the above pseudo-code in C and obtain the result:

$$Dr=12.8396261.$$

Bracketing Methods - Method of False Position

Method of false position based on the following observation



If $f(x_l)$ is closer to zero than $f(x_u)$, then root is likely to be closer to x_l than to x_u . Using similarity of triangles, we can obtain the following expression

$$\frac{-f(x_l)}{x_r - x_l} = \frac{f(x_u)}{x_u - x_r}$$

Bracketing Methods - Method of False Position

which can be rearranged to

$$x_r = \frac{x_u f(x_l) - x_l f(x_u)}{f(x_l) - f(x_u)}$$

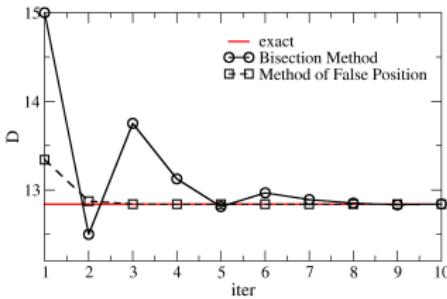
or

$$x_r = x_u - \frac{f(x_u)(x_l - x_u)}{f(x_l) - f(x_u)}$$

To use the Method of False Position to find roots, use Steps 1-3 of the Bisection method but replace Step 2 with the above formula

Bisection Method vs Method of False Position

Illustration of the differences between the two methods



The figure shows solutions using initial lower and upper guesses of $D_l = 10$ and $D_u = 20$.

Method of False position: **monotonic convergence** to solution

Bisection method: **oscillates** towards the solution

⇒ Method of False position converges (much) faster

Exercise

Exercise 2

Use Method of False Position to obtain the root of

$$f(D) = \frac{1000}{D} \left(1 - \exp \left(-\frac{8D}{100} \right) \right) - 50$$

Root finding - Open Methods

Open methods require either only one initial guess or two initial guesses which do not have to bracket the actual root.

⇒ easier to obtain initial guess for open methods (you can choose almost anything!).

In general, if open methods converge, they do so much more rapidly than bracketing methods.

Unlike bracketing methods, which always converge, open methods *MAY NOT* converge. We will discuss the following open methods:

- Fixed-point iteration
- Newton-Raphson method
- Secant method

Open Methods - Fixed-point iteration

To find the root of

$$f(x) = 0, \quad (9)$$

arrange function so that x is on left hand side of equation, i.e.

$$x = g(x).$$

Rewrite the above equation as

$$x_{i+1} = g(x_i). \quad (10)$$

Eq. (10) provides us with an iterative formula to find roots of Eq. (9). Note that functional form of Eq. (9) is not changed. Eq. (10) is just a rearrangement of Eq. (9).

Suppose we want to find the root to

$$f(x) = e^{-x} - x = 0. \quad (11)$$

Open Methods - Fixed-point iteration

Rearrange to get

$$x = e^{-x}$$

Thus, $g(x) = e^{-x}$. Hence, the formula for one point iteration scheme for solving this example is

$$x_{i+1} = e^{-x_i}$$

The iterative equation for fixed-point iteration is

$$x_{i+1} = g(x_i) \quad (12)$$

Iterate until:

$$|g(x_i) - x_i| < \varepsilon$$

where ε is your convergence criterium.

Convergence of Fixed-point iteration

Since $f(x_r) = 0$ and Eq. (12) is obtained by just rearranging $f(x)$
⇒ x_r must also satisfy

$$x_r = g(x_r) \quad (13)$$

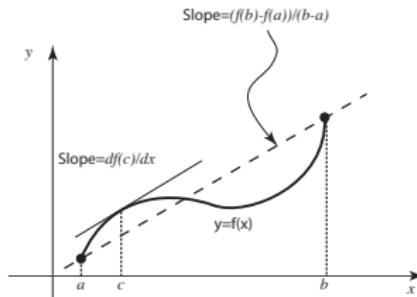
Subtracting Eq. (13) from (12) gives

$$x_{i+1} - x_r = g(x_i) - g(x_r).$$

The above equation can be re-written as [by multiplying by
 $1 = (x_i - x_r)/(x_i - x_r)$]

$$x_{i+1} - x_r = \left(\frac{g(x_i) - g(x_r)}{(x_i - x_r)} \right) (x_i - x_r) \quad (14)$$

Derivative mean value theorem



If f continuous function and differentiable between (a, b) , then a number c between a and b exists such that

$$\frac{dg}{dx}(x = c) = \frac{g(b) - g(a)}{b - a}$$

Or, there is a point $(x = c)$ of a function (g) in between a and b such that the derivative of that function is given by $g(b) - g(a)/(b - a)$.

Convergence of Fixed-point iteration

Here: $\frac{dg}{dx}(c) = \frac{g(x_r) - g(x_i)}{x_r - x_i}$ (15)

$$x_{i+1} - x_r = \left(\frac{dg}{dx}(c) \right) (x_i - x_r)$$

This says that newer guess will be closer to actual root than older guess, provided that $|dg(c)/dx| < 1$.

⇒ One condition for convergence.

Exercise - Fixed-point iteration

Exercise 3

Find the root for the following equation

$$f(x) = e^x - 10x$$

by using the fixed point iteration. Try to find the solution with $g(x) = e^x/10$ and $x_0 = 1.0$ and $x_0 = 4.0$ as your initial guesses. Which one of your solutions blows up ? Why ? Can you get a solution with $x_0 = 3.5$?

Open Methods - Newton–Raphson

The Newton–Raphson method is most widely used root finding method.

Converges quickly and only needs one initial guess.

Fig. 3 shows it is easy to express x_i in terms of $f(x_i)$, $f'(x_i)$ and x_{i+1} .

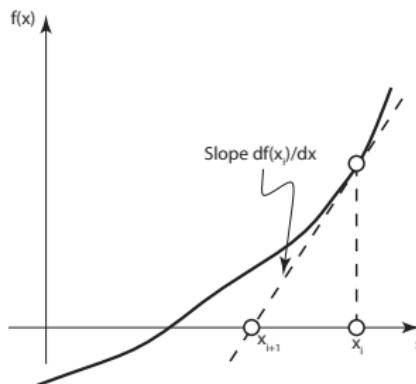


Figure: Newton–Raphson method

Open Methods - Newton–Raphson

$$f'(x_i) = \frac{f(x_i)}{x_i - x_{i+1}} \Rightarrow x_i - x_{i+1} = \frac{f(x_i)}{f'(x_i)}$$

gives following iterative formula for Newton–Raphson method

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (16)$$

To find root using Newton-Raphson method, do the following:

1. Let the initial guess be x_i
2. Find x_{i+1} by using Eq. (16)
3. Let $x_i = x_{i+1}$; repeat steps 2 and 3 until you feel your answer is accurate enough ($< \varepsilon$).

Exercise - Newton–Raphson

Exercise 4

Newton–Raphson method to find the root of

$$f(x) = \cos x - x$$

Exercise - Newton-Raphson

```
1  eps = 1.0E-8; /* value that determines accuracy of your answer */
2  maxiter = 50; /* maximum number of iterations*/
3  x_i = 0.2;    /* Initial lower guess of root*/
4
5  f_i = cos(x_i) - x_i /* Initial function value */
6
7  for i=1:maxiter,
8      f_i = cos(x_i) - x_i; /* Compute function value */
9      df_i = -sin(x_i) - 1; /* Derivative of f */
10     x_i = x_i-f_i/df_i;   /* New value of x_i */
11
12    /* testing to see if we have found the root */
13    /* If the root is found then exit the for loop */
14    if abs(f_i) < eps
15        break;
16    end
17 end;
```

Write the above psuedo-code in C and obtain the result of
 $x_i=0.739085133$.

Open Methods - Newton–Raphson

One of the reasons why Newton–Raphson method is so popular is that it converges very fast.

Proof: The Taylor series expansion of a function $f(x)$ can be written as

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{f''(x_i)}{2!}(x_{i+1} - x_i)^2$$

If we let $x_{i+1} = x_r$, then we will have $f(x_{i+1}) = f(x_r) = 0$. Put this information into the above equation to get

$$0 = f(x_i) + f'(x_i)(x_r - x_i) + \frac{f''(x_i)}{2!}(x_r - x_i)^2 \quad (17)$$

Open Methods - Newton–Raphson

We can also rearrange Eq. (16) to get

$$0 = f(x_i) + f'(x_i)(x_{i+1} - x_i). \quad (18)$$

Subtracting Eq. (18) from Eq. (17) will give

$$0 = f'(x_i)(x_r - x_{i+1}) + \frac{f''(x_i)}{2!}(x_r - x_i)^2.$$

Rearrange above equation to get

$$x_r - x_{i+1} = -\frac{f''(x_i)}{2f'(x_i)}(x_r - x_i)^2 \quad (19)$$

thus

$$|x_r - x_{i+1}| = \left| \frac{f''(x_i)}{2f'(x_i)} \right| (x_r - x_i)^2 \quad (20)$$

Open Methods - Newton–Raphson

$$|x_r - x_{i+1}| = \left| \frac{f''(x_i)}{2f'(x_i)} \right| (x_r - x_i)^2 \quad (21)$$

Eq. (21) states that magnitude of error in Newton–Raphson method roughly proportional to square of previous error.

⇒ quadratic convergence (only if $f'(x_r) \neq 0$).

If $f'(x_r) = 0$, then another method has to be introduced in order to achieve quadratic convergence.

Newton–Raphson - Pitfalls

- Requires evaluation of derivative of a function - can be difficult or inconvenient.
- Problems in vicinity of local maxima/minima because $f'(x_i) = 0$
This is illustrated in Fig. 4.

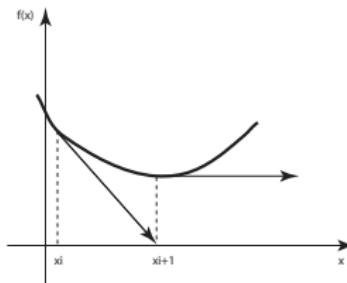


Figure: Pitfalls of Newton–Raphson method.

Open methods - Secant Method

One of main reasons for introducing secant method is that in Newton–Raphson method, (analytical) derivatives $f'(x_i)$ required.

In Secant method, derivative of function is approximated by backward finite difference

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

Insert into Eq. (16) gives formula for Secant method

$$x_{i+1} = x_i - \frac{(x_i - x_{i-1}) f(x_i)}{(f(x_i) - f(x_{i-1}))} \quad (22)$$

Eq. (22) requires two initial estimates of x , i.e. x_i and x_{i-1} .

However, because function $f(x)$ not required to change signs between estimates, not classified as bracketing method.

Open methods - Secant Method

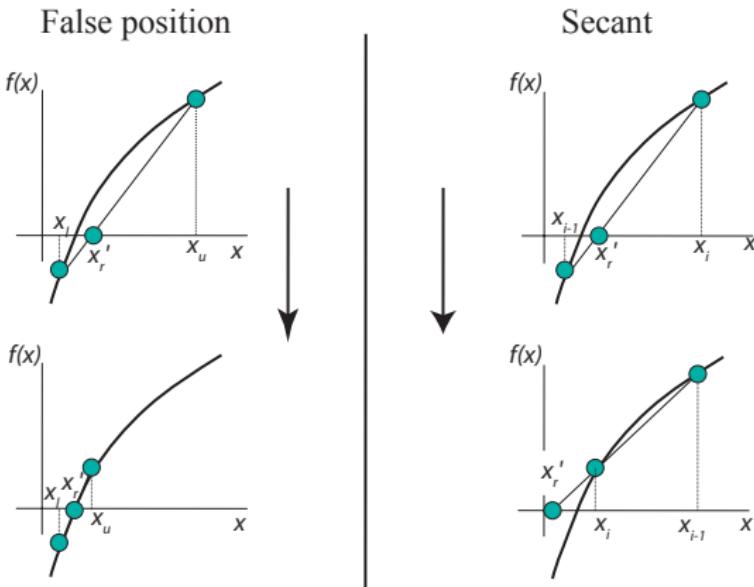


Figure: Difference between Secant method (classified as an open method) and method of false position (classified as bracketing method).

Root finding - System of nonlinear equations

So far, only talked about methods of finding roots for one equation.
What if we need to simultaneously find roots for two or more equations ?

For example, what if you want to find a set of x and y values so that they satisfy the following two equations.

$$y = x^2 + 1 \quad (23)$$

$$y = 3 \cos(x) \quad (24)$$

Root finding - System of nonlinear equations

These two equations are illustrated in Fig. 6.

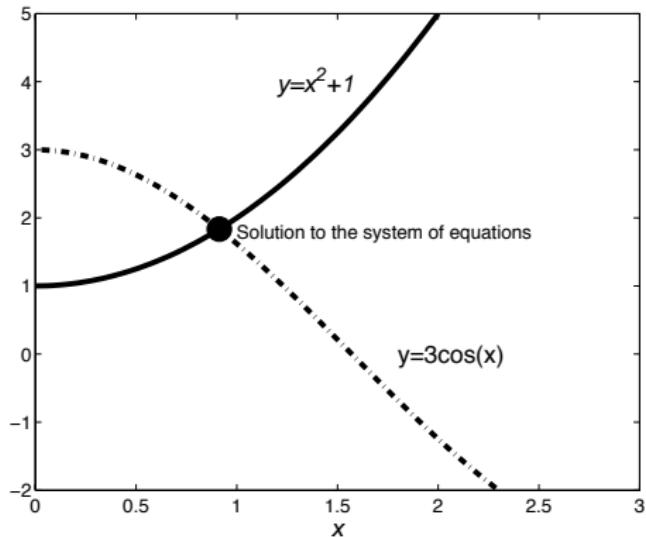


Figure: Eqs. (23) and (24). Find point of intersection.

Root finding - System of nonlinear equations

In general, find all x_i 's so that we satisfy the following set of equations

$$f_1(x_1, x_2, \dots, x_n) = 0$$

$$f_2(x_1, x_2, \dots, x_n) = 0$$

$$f_3(x_1, x_2, \dots, x_n) = 0$$

.

.

.

$$f_n(x_1, x_2, \dots, x_n) = 0$$

Here, let's just confine ourselves to $n = 2$.

So we wish to find $x_1 = x$ and $x_2 = y$ so that the following two equations are satisfied

$$u(x, y) = 0 \qquad v(x, y) = 0.$$

Root finding - System of nonlinear equations

The multivariable Taylor series for u and v can be written as

$$u_{i+1} = u_i + (x_{i+1} - x_i) \frac{\partial u}{\partial x}(x_i, y_i) + (y_{i+1} - y_i) \frac{\partial u}{\partial y}(x_i, y_i)$$

$$v_{i+1} = v_i + (x_{i+1} - x_i) \frac{\partial v}{\partial x}(x_i, y_i) + (y_{i+1} - y_i) \frac{\partial v}{\partial y}(x_i, y_i)$$

For simplicity, the following notation will be used

$$\frac{\partial u}{\partial x}(x_i, y_i) \equiv \left(\frac{\partial u}{\partial x} \right)_i$$

Root finding - System of nonlinear equations

The above two equations can be re-written as

$$u_{i+1} = u_i + (x_{i+1} - x_i) \left(\frac{\partial u}{\partial x} \right)_i + (y_{i+1} - y_i) \left(\frac{\partial u}{\partial y} \right)_i$$

$$v_{i+1} = v_i + (x_{i+1} - x_i) \left(\frac{\partial v}{\partial x} \right)_i + (y_{i+1} - y_i) \left(\frac{\partial v}{\partial y} \right)_i$$

which by setting $u_{i+1} = v_{i+1} = 0$ can be simplified to

$$\left(\frac{\partial u}{\partial x} \right)_i x_{i+1} + \left(\frac{\partial u}{\partial y} \right)_i y_{i+1} = -u_i + x_i \left(\frac{\partial u}{\partial x} \right)_i + y_i \left(\frac{\partial u}{\partial y} \right)_i \quad (25)$$

$$\left(\frac{\partial v}{\partial x} \right)_i x_{i+1} + \left(\frac{\partial v}{\partial y} \right)_i y_{i+1} = -v_i + x_i \left(\frac{\partial v}{\partial x} \right)_i + y_i \left(\frac{\partial v}{\partial y} \right)_i \quad (26)$$

Root finding - System of nonlinear equations

The above two equations are a set of two linear equations. They can easily be solved.

By solving Eqs. (25) and (26) for x_{i+1} and y_{i+1} , it can be found that

$$x_{i+1} = x_i + \frac{v_i \left(\frac{\partial u}{\partial y} \right)_i - u_i \left(\frac{\partial v}{\partial y} \right)_i}{\left(\frac{\partial u}{\partial x} \right)_i \left(\frac{\partial v}{\partial y} \right)_i - \left(\frac{\partial u}{\partial y} \right)_i \left(\frac{\partial v}{\partial x} \right)_i}$$

$$y_{i+1} = y_i + \frac{u_i \left(\frac{\partial v}{\partial x} \right)_i - v_i \left(\frac{\partial u}{\partial x} \right)_i}{\left(\frac{\partial u}{\partial x} \right)_i \left(\frac{\partial v}{\partial y} \right)_i - \left(\frac{\partial u}{\partial y} \right)_i \left(\frac{\partial v}{\partial x} \right)_i}$$

Next week

Systems of Linear Equations



THE UNIVERSITY OF

MELBOURNE



ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne
Slides developed by Prof Richard Sandberg

This week

LECTURE 13/14

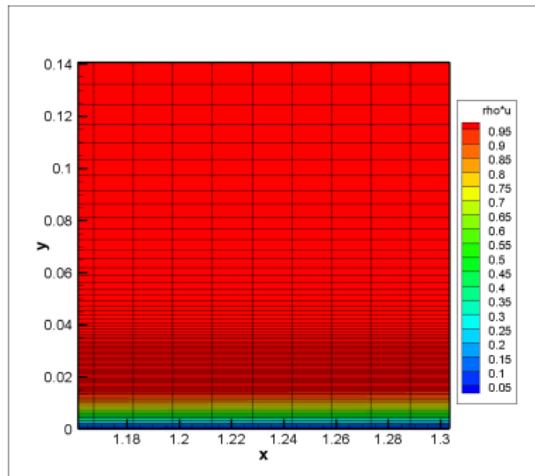
Systems of Linear Algebraic Equations

Systems of Linear Algebraic Equations

In many problems of engineering interest, need to solve a set of linear algebraic equations.

Consider following example:

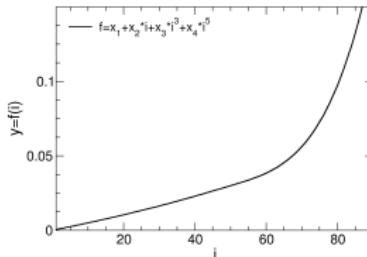
We want to design a grid for a simulation that has stretching, i.e. is finer in one place than in another.



Systems of Linear Algebraic Equations

A good choice is a polynomial function of 5th order,

$$f(i) = x_1 + x_2 i + x_3 i^3 + x_4 i^5. \quad (1)$$



We then specify several constraints, saying that

1. $f(i = 1) = c_1 = 0$
2. $f'(i = 1) = c_2 = \Delta_{wall}$
3. $f(i = n) = c_3 = f_{top}$
4. $f'(i = n) = c_4 = \Delta_{top}$

Systems of Linear Algebraic Equations

The derivative of the function is

$$f'(i) = x_2 + 3x_3i^2 + 5a_4i^4. \quad (2)$$

Thus we can write a system of equations as:

$$x_1 + x_2 + x_3 + x_4 = 0$$

$$0 + x_2 + 3x_3 + 5x_4 = \Delta_{wall}$$

$$x_1 + nx_2 + n^3x_3 + n^5x_4 = f_{top}$$

$$0 + x_2 + 3n^2x_3 + 5n^4x_4 = \Delta_{top}$$

Systems of Linear Algebraic Equations

We can write this system as:

$$[A] \{X\} = \{C\} \quad (3)$$

where

$$[A] = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 3 & 5 \\ 1 & n & n^3 & n^5 \\ 0 & 1 & 3n^2 & 5n^4 \end{bmatrix},$$

$$\{X\} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix}, \quad \{C\} = \begin{Bmatrix} 0 \\ \Delta_{wall} \\ f_{top} \\ \Delta_{top} \end{Bmatrix}.$$

This is a system of algebraic equations, for which we would like to find x_i .

Systems of Linear Algebraic Equations

The available methods to solve systems of linear algebraic equations can be broadly classified as follows:

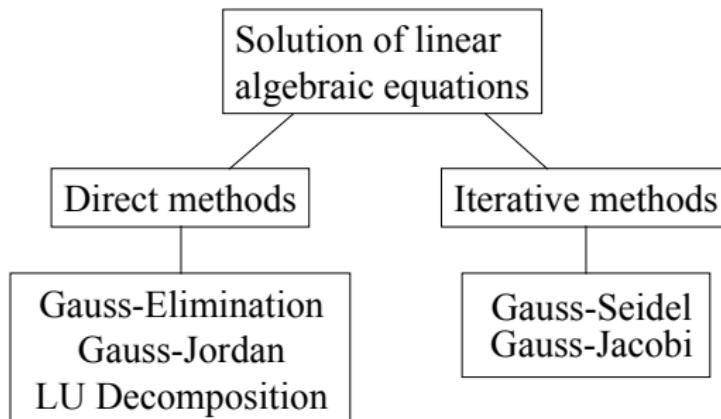


Figure: Classification of different methods for solving a system of algebraic equations.

Systems of Linear Algebraic Equations

Consider following set of equations:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n = c_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n = c_2$$

.

.

.

.

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nn}x_n = c_n$$

where all the a 's and c 's are constants.

We need to find all the x_i 's such that all the above equations are satisfied.

Systems of Linear Algebraic Equations

Sometimes, the matrix notation will be used

$$[A] \{X\} = \{C\} \quad (4)$$

where

$$[A] = \begin{bmatrix} a_{11} & a_{12} & \cdot & \cdot & a_{1n} \\ a_{21} & a_{22} & \cdot & \cdot & a_{2n} \\ a_{31} & a_{32} & \cdot & \cdot & a_{3n} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ a_{n1} & a_{n2} & \cdot & \cdot & a_{nn} \end{bmatrix},$$

$$\{X\} = \left\{ \begin{array}{c} x_1 \\ x_2 \\ x_3 \\ \cdot \\ x_n \end{array} \right\}, \quad \{C\} = \left\{ \begin{array}{c} c_1 \\ c_2 \\ c_3 \\ \cdot \\ c_n \end{array} \right\}.$$

Systems of Linear Algebraic Equations

Equation (4) can be solved by inverting the matrix $[A]$ and then multiplying $[C]$ by $[A]^{-1}$.

$$\{X\} = [A]^{-1} \{C\}$$

But this might be difficult and time consuming, sometimes impossible to do if $[A]$ is large.

Can we numerically solve Equation (4) ?

Direct Methods - Gauss Elimination I

In *Direct Methods* methods, no iterations are involved.

One method is **Gauss elimination**, a procedure to solve the following system of linear equations

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1n}x_n = c_1 = a_{1,n+1}$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2n}x_n = c_2 = a_{2,n+1}$$

.

.

.

.

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nn}x_n = c_n = a_{n,n+1}$$

The following steps need to be taken:

Direct Methods - Gauss Elimination II

- First augment $[A]$ with $\{C\}$.

$$\left[\begin{array}{cccc|cc} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdot & \cdot & \cdot & a_{1n}^{(1)} & | & a_{1,n+1}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & \cdot & \cdot & \cdot & a_{2n}^{(1)} & | & a_{2,n+1}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & \cdot & \cdot & \cdot & a_{3n}^{(1)} & | & a_{3,n+1}^{(1)} \\ \cdot & | & \cdot \\ \cdot & | & \cdot \\ \cdot & | & \cdot \\ \cdot & | & \cdot \\ a_{n1}^{(1)} & a_{n2}^{(1)} & a_{n3}^{(1)} & \cdot & \cdot & \cdot & a_{nn}^{(1)} & | & a_{n,n+1}^{(1)} \end{array} \right]$$

- Forward elimination to get matrix into solvable form
Eliminate x_1 from second and subsequent equations.
 \Rightarrow to eliminate x_1 from i th row, multiply row 1 by $a_{i1}^{(1)}/a_{11}^{(1)}$ and subtract from row i .

Direct Methods - Gauss Elimination III

$$\left[\begin{array}{cccccc|c} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdot & \cdot & \cdot & a_{1n}^{(1)} & | & a_{1,n+1}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & a_{23}^{(1)} & \cdot & \cdot & \cdot & a_{2n}^{(1)} & | & a_{2,n+1}^{(1)} \\ a_{31}^{(1)} & a_{32}^{(1)} & a_{33}^{(1)} & \cdot & \cdot & \cdot & a_{3n}^{(1)} & | & a_{3,n+1}^{(1)} \\ \cdot & | & \cdot \\ \cdot & | & \cdot \\ \cdot & | & \cdot \\ a_{n1}^{(1)} & a_{n2}^{(1)} & a_{n3}^{(1)} & \cdot & \cdot & \cdot & a_{nn}^{(1)} & | & a_{n,n+1}^{(1)} \end{array} \right]$$

↓

$$\left[\begin{array}{cccccc|c} a_{11}^{(2)} & a_{12}^{(2)} & a_{13}^{(2)} & \cdot & \cdot & \cdot & a_{1n}^{(2)} & | & a_{1,n+1}^{(2)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdot & \cdot & \cdot & a_{2n}^{(2)} & | & a_{2,n+1}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} & \cdot & \cdot & \cdot & a_{3n}^{(2)} & | & a_{3,n+1}^{(2)} \\ \cdot & | & \cdot \\ \cdot & | & \cdot \\ \cdot & | & \cdot \\ 0 & a_{n2}^{(2)} & a_{n3}^{(2)} & \cdot & \cdot & \cdot & a_{nn}^{(2)} & | & a_{n,n+1}^{(2)} \end{array} \right]$$

$$a_{ij}^{(2)} = a_{ij}^{(1)} \quad \text{for} \quad i=1 \quad \text{and} \quad j=1, 2, 3, \dots, n+1$$

$$a_{ij}^{(2)} = 0 \quad \text{for} \quad i=2, 3, \dots, n \quad \text{and} \quad j=1$$

$$a_{ij}^{(2)} = a_{ij}^{(1)} - \left(\frac{a_{i1}^{(1)}}{a_{11}^{(1)}} \right) a_{1j}^{(1)} \quad \text{for} \quad i=2, 3, \dots, n \quad \text{and} \quad j=2, 3, \dots, n+1$$

Direct Methods - Gauss Elimination IV

- Next, eliminate x_2 from third and subsequent equations.
⇒ to eliminate x_2 from the i th row, multiply row 2 by $a_{i2}^{(2)}/a_{22}^{(2)}$ and subtract from row i .

$$\left[\begin{array}{cccccc|cc} a_{11}^{(2)} & a_{12}^{(2)} & a_{13}^{(2)} & \cdot & \cdot & \cdot & a_{1n}^{(2)} & a_{1,n+1}^{(2)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdot & \cdot & \cdot & a_{2n}^{(2)} & a_{2,n+1}^{(2)} \\ 0 & a_{32}^{(2)} & a_{33}^{(2)} & \cdot & \cdot & \cdot & a_{3n}^{(2)} & a_{3,n+1}^{(2)} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 0 & a_{n2}^{(2)} & a_{n3}^{(2)} & \cdot & \cdot & \cdot & a_{nn}^{(2)} & a_{n,n+1}^{(2)} \end{array} \right]$$



$$\left[\begin{array}{cccccc|cc} a_{11}^{(3)} & a_{12}^{(3)} & a_{13}^{(3)} & \cdot & \cdot & \cdot & a_{1n}^{(3)} & a_{1,n+1}^{(3)} \\ 0 & a_{22}^{(3)} & a_{23}^{(3)} & \cdot & \cdot & \cdot & a_{2n}^{(3)} & a_{2,n+1}^{(3)} \\ 0 & 0 & a_{33}^{(3)} & \cdot & \cdot & \cdot & a_{3n}^{(3)} & a_{3,n+1}^{(3)} \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \\ 0 & 0 & a_{n3}^{(2)} & \cdot & \cdot & \cdot & a_{nn}^{(2)} & a_{n,n+1}^{(2)} \end{array} \right]$$

Direct Methods - Gauss Elimination V

$$a_{ij}^{(3)} = a_{ij}^{(2)}, \quad a_{ij}^{(3)} = 0, \quad a_{ij}^{(3)} = a_{ij}^{(2)} - \left(\frac{a_{i2}^{(2)}}{a_{22}^{(2)}} \right) a_{2j}^{(2)}$$

Repeating steps above, we get matrix

$$\left[\begin{array}{cccccc|cc} a_{11}^{(n)} & a_{12}^{(n)} & a_{13}^{(n)} & a_{14}^{(n)} & \dots & a_{1n}^{(n)} & | & a_{1,n+1}^{(n)} \\ 0 & a_{22}^{(n)} & a_{23}^{(n)} & a_{24}^{(n)} & \dots & a_{2n}^{(n)} & | & a_{2,n+1}^{(n)} \\ 0 & 0 & a_{33}^{(n)} & a_{34}^{(n)} & \dots & a_{3n}^{(n)} & | & a_{3,n+1}^{(n)} \\ 0 & 0 & 0 & a_{44}^{(n)} & \dots & a_{4n}^{(n)} & | & a_{4,n+1}^{(n)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & | & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & | & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & | & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & | & \vdots \\ 0 & 0 & 0 & \dots & \dots & a_{nn}^{(n)} & | & a_{n,n+1}^{(n)} \end{array} \right] \quad (5)$$

$$a_{ij}^{(k)} = a_{ij}^{(k-1)}, \quad a_{ij}^{(k)} = 0, \quad a_{ij}^{(k)} = a_{ij}^{(k-1)} - \left(\frac{a_{i,k-1}^{(k-1)}}{a_{k-1,k-1}^{(k-1)}} \right) a_{k-1,j}^{(k-1)}$$

Direct Methods - Gauss Elimination VI

We can solve system of equations given by Eq. (5) by using back substitution.

First look at the last row of Eq. (5)

$$a_{nn}^{(n)} x_n = a_{n,n+1}^{(n)}$$

$$x_n = \frac{a_{n,n+1}^{(n)}}{a_{nn}^{(n)}} \quad (6)$$

So we can get a numerical value for x_n .

Next, look at the 2nd last row of Eq. (5)

$$a_{n-1,n-1}^{(n)} x_{n-1} + a_{n-1,n}^{(n)} x_n = a_{n-1,n+1}^{(n)}$$

$$x_{n-1} = \frac{a_{n-1,n+1}^{(n)} - a_{n-1,n-1}^{(n)} x_n}{a_{n-1,n-1}^{(n)}} \quad (7)$$

Direct Methods - Gauss Elimination VII

Since we already know x_n from Eq. (10), we can also get a numerical value for x_{n-1} from Eq. (7).

Similarly, looking at the 3rd last row of Eq. (5)

$$a_{n-2,n-2}^{(n)}x_{n-2} + a_{n-2,n-1}^{(n)}x_{n-1} + a_{n-2,n}^{(n)}x_n = a_{n-2,n+1}^{(n)} \quad (8)$$

$$x_{n-2} = \frac{a_{n-2,n+1}^{(n)} - (a_{n-2,n}^{(n)}x_n + a_{n-2,n-1}^{(n)}x_{n-1})}{a_{n-2,n-2}^{(n)}} \quad (9)$$

In general, the formula for getting the numerical values for x_i is

$$x_n = \frac{a_{n,n+1}^{(n)}}{a_{nn}^{(n)}} \text{ and } x_i = \frac{a_{i,n+1}^{(n)} - \sum_{j=i+1}^n a_{ij}^{(n)}x_j}{a_{ii}^{(n)}}$$

for $i = n-1, n-2, n-3, \dots, 2, 1$.

Gauss Elimination - Example 1

Consider this example:

$$[A] = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \\ 2 & 3 & 2 \end{bmatrix},$$

$$\{X\} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}, \quad \{C\} = \begin{Bmatrix} 14 \\ 16 \\ 17 \end{Bmatrix}.$$

First augment $[A]$ with $\{C\}$.

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 14 \\ 3 & 2 & 2 & 16 \\ 2 & 3 & 2 & 17 \end{array} \right]$$

Eliminate x_1 from second and third row.

Gauss Elimination - Example II

To eliminate x_1 from 2nd row, multiply row 1 by $a_{21}^{(1)}/a_{11}^{(1)}$ and subtract from row 2.

To eliminate x_1 from 3rd row, multiply row 1 by $a_{31}^{(1)}/a_{11}^{(1)}$ and subtract from row 3.

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 14 \\ 0 & 4 & 7 & 26 \\ 0 & 1 & 4 & 11 \end{array} \right]$$

Next, eliminate x_2 from third row.

To eliminate x_2 from the 3rd row, multiply row 2 by $a_{32}^{(2)}/a_{22}^{(2)}$ and subtract from row 3.

$$\left[\begin{array}{ccc|c} 1 & 2 & 3 & 14 \\ 0 & 4 & 7 & 26 \\ 0 & 0 & -9/4 & -18/4 \end{array} \right]$$

Now use back substitution.

Gauss Elimination - Example III

The last row gives us

$$x_3 = \frac{a_{3,4}^{(3)}}{a_{33}^{(3)}} = \frac{-18/4}{-9/4} = 2 \quad (10)$$

Next, look at 2nd last row

$$a_{2,2}^{(3)}x_2 + a_{2,3}^{(3)}x_3 = a_{2,4}^{(3)} \Rightarrow 4x_2 + 7 \cdot 2 = 26$$

$$x_2 = \frac{26 - 14}{4} = 3$$

Gauss Elimination - Example IV

Using the general formula for getting the numerical value for x_i for the last unknown value x_1 :

$$\begin{aligned}x_i &= \frac{a_{i,n+1}^{(n)} - \sum_{j=i+1}^n a_{ij}^{(n)} x_j}{a_{ii}^{(n)}} \Rightarrow x_1 = \frac{a_{1,4}^{(3)} - \sum_{j=2}^3 a_{1j}^{(3)} x_j}{a_{11}^{(3)}} \\&= \frac{14 - 2 \cdot 3 - 3 \cdot 2}{1} = 2\end{aligned}$$

Thus

$$\{X\} = \left\{ \begin{array}{c} 2 \\ 3 \\ 2 \end{array} \right\} .$$

Gauss Elimination - Pitfalls I

The pitfalls of the Gauss elimination method for solving a linear system of equations are:

- Very expensive, compared to other methods
- The formula for forward elimination stage requires we divide by diagonal components of matrix $[A]$.

Need to rearrange equations to ensure that we are not dividing by zero.

For example the following system

$$\begin{aligned} 2x_2 + 3x_3 &= 1 \\ 3x_1 + 5x_2 + 6x_3 &= 2 \\ 9x_1 + 2x_2 + 3x_3 &= 3 \end{aligned}$$

Gauss Elimination - Pitfalls II

For this case, $a_{11} = 0$, will run into problems in first forward elimination stage. Re-writing equation as

$$3x_1 + 5x_2 + 6x_3 = 2$$

$$2x_2 + 3x_3 = 1$$

$$9x_1 + 2x_2 + 3x_3 = 3$$

will eliminate problem. The process of interchanging rows so that you are not dividing by zero is called partial pivoting (also try to avoid dividing by small numbers).

Gauss Elimination - LU decomposition I

Aim is to solve

$$[A] \{X\} = \{C\}. \quad (11)$$

If $[A]$ is a 'full' matrix, it takes many steps to solve it.

Hence, we would like to re-express $[A]$ such that

$$[A] = [L][U], \quad (12)$$

where $[L]$ and $[U]$ are lower- and upper-triangular matrices.

The structure of $[L]$ and $[U]$ are shown below

$$[L] = \begin{bmatrix} l_{11} & 0 & 0 & 0 & \cdot & \cdot & 0 \\ l_{21} & l_{22} & 0 & 0 & \cdot & \cdot & 0 \\ l_{31} & l_{32} & l_{33} & 0 & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ l_{n1} & l_{n2} & l_{n3} & l_{n4} & \cdot & \cdot & l_{nn} \end{bmatrix} \quad [U] = \begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} & \cdot & \cdot & u_{1n} \\ 0 & 1 & u_{23} & u_{24} & \cdot & \cdot & u_{2n} \\ 0 & 0 & 1 & u_{34} & \cdot & \cdot & u_{3n} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & 0 & \cdot & \cdot & 1 \end{bmatrix}$$

Gauss Elimination - LU decomposition II

If we now substitute Eq. (12) into Eq. (4), then we will have

$$[L] [U] \{X\} = \{C\} \quad (13)$$

Let

$$[U] \{X\} = \{R\} \quad (14)$$

and substitute Eq. (14) into Eq. (13) to give

$$[L] \{R\} = \{C\} \quad (15)$$

Thus we have replaced the original problem,

$$[A] \{X\} = \{C\}. \quad (16)$$

by two problems

$$[L] \{R\} = \{C\} \quad , \quad [U] \{X\} = \{R\} \quad (17)$$

Gauss Elimination - LU decomposition III

However, the two equations are much easier to solve because $[L]$ and $[U]$ are lower- and upper-triangular matrices.

⇒ To solve the Eqs. (17), we only need backward and forward substitution, respectively.

Gauss Elimination - LU decomposition IV

Thus, use the following **procedure** to solve the system

$$[A] \{X\} = \{C\} \quad :$$

- Decompose $[A]$ into $[L]$ and $[U]$
- Solve $[L] \{R\} = \{C\}$ and obtain $\{R\}$ by forward substitution.
- Use $\{R\}$ obtained in above step and substitute into $[U] \{X\} = \{R\}$ so that you can solve for $\{X\}$ by backward substitution.

Gauss Elimination - LU decomposition V

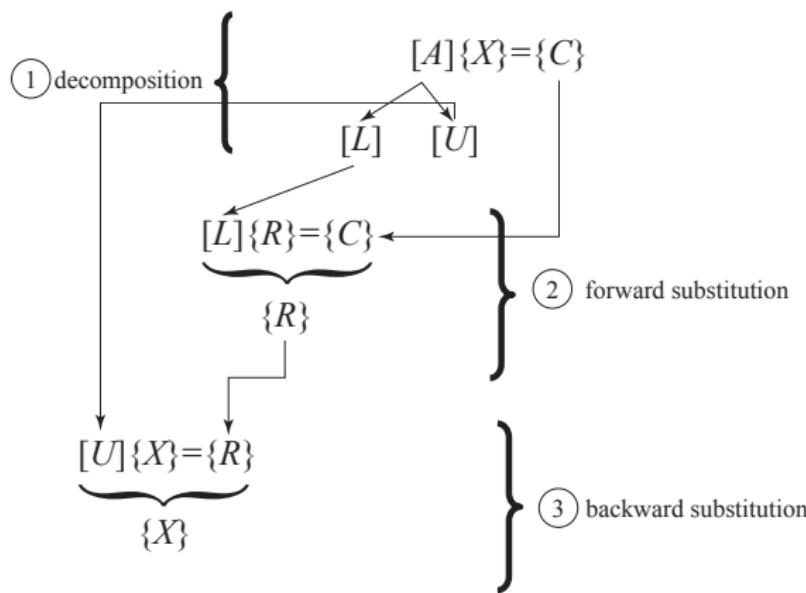


Figure: The flow of information in the LU decomposition method.

Gauss Elimination - LU decomposition VI

What is left is to find $[L]$ and $[U]$ from $[A]$.

To illustrate the procedure of finding $[L]$ and $[U]$, we will as example use a small 4×4 $[A]$ matrix.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we multiply rows of $[L]$ by first column of $[U]$ and compare with $[A]$, we get

$$l_{11} = a_{11}, \quad l_{21} = a_{21}, \quad l_{31} = a_{31}, \quad l_{41} = a_{41}.$$

Thus, the first column of $[L]$ is first column of $[A]$.

Multiply the first row of $[L]$ by the columns of $[U]$ to get

$$l_{11}u_{12} = a_{12}, \quad l_{11}u_{13} = a_{13}, \quad l_{11}u_{14} = a_{14}.$$

Gauss Elimination - LU decomposition VII

Hence, the first row of $[U]$ can be determined to be

$$u_{12} = \frac{a_{12}}{l_{11}}, u_{13} = \frac{a_{13}}{l_{11}}, u_{14} = \frac{a_{14}}{l_{11}} \quad (18)$$

We keep alternating between getting a column of $[L]$ and a row of $[U]$. So we next get the equation for the second column of $[L]$ by multiplying the rows of $[L]$ by the second column of $[U]$.

$$\begin{aligned} l_{21}u_{12} + l_{22} &= a_{22} \\ l_{31}u_{12} + l_{32} &= a_{32} \\ l_{41}u_{12} + l_{42} &= a_{42} \end{aligned}$$

This gives 2nd column of $[L]$ to be

$$\begin{aligned} l_{22} &= a_{22} - l_{21}u_{12} \\ l_{32} &= a_{32} - l_{31}u_{12} \\ l_{42} &= a_{42} - l_{41}u_{12} \end{aligned} \quad (19)$$

Gauss Elimination - LU decomposition VIII

since we already know u_{12} , l_{21} , l_{31} and l_{41} from $[L]\{R\} = \{C\}$
and $u_{12} = \frac{a_{12}}{l_{11}}$, $u_{13} = \frac{a_{13}}{l_{11}}$, $u_{14} = \frac{a_{14}}{l_{11}}$.

Then multiply second row of $[L]$ by columns of $[U]$ to get

$$\begin{aligned} l_{21}u_{13} + l_{22}u_{23} &= a_{23} \\ l_{21}u_{14} + l_{22}u_{24} &= a_{24} \end{aligned}$$

This allows us to obtain 2nd row of $[U]$

$$\begin{aligned} u_{23} &= \frac{a_{23} - l_{21}u_{13}}{l_{22}} \\ u_{24} &= \frac{a_{24} - l_{21}u_{14}}{l_{22}} \end{aligned}$$

since Eqs. (15), (21) and (23) will give you l_{21} , u_{13} , u_{14} and l_{22} .

Gauss Elimination - LU decomposition IX

If you keep going you get

$$\begin{aligned}l_{33} &= a_{33} - l_{31}u_{13} - l_{32}u_{23}, & l_{43} &= a_{43} - l_{41}u_{13} - l_{42}u_{23} \\u_{34} &= \frac{a_{34} - l_{31}u_{14} - l_{32}u_{24}}{l_{33}} \\l_{44} &= a_{44} - l_{41}u_{14} - l_{42}u_{24} - l_{43}u_{34}\end{aligned}$$

Gauss Elimination - LU decomposition X

In general, the formula for getting elements of $[L]$ and $[U]$ from elements of an $n \times n$ $[A]$ matrix can be written as

$$\begin{aligned}l_{i1} &= a_{i1} \quad \text{for } i = 1, 2, \dots, n \\u_{1j} &= \frac{a_{1j}}{l_{11}} \quad \text{for } j = 2, 3, \dots, n\end{aligned}$$

Begin for loop $j = 2, 3, \dots, n$

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \quad \text{for } i = j, j+1, \dots, n$$

$$u_{jk} = \frac{a_{jk} - \sum_{i=1}^{j-1} l_{ji} u_{ik}}{l_{jj}} \quad \text{for } k = j+1, j+2, \dots, n$$

End for loop

$$u_{ii} = 1.0 \quad \text{for } i = 1, 2, \dots, n$$

(20)

This procedure is known as Crout's LU decomposition algorithm.

Gauss Elimination - LU decomposition XI

After obtaining $[L]$ and $[U]$, need to solve for $\{R\}$:

$$[L]\{R\} = \{C\}$$
$$\begin{bmatrix} l_{11} & 0 & 0 & 0 \\ l_{21} & l_{22} & 0 & 0 \\ l_{31} & l_{32} & l_{33} & 0 \\ l_{41} & l_{42} & l_{43} & l_{44} \end{bmatrix} \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{Bmatrix} = \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{Bmatrix} \quad (21)$$

by forward substitution. From the first row,

$$l_{11}r_1 = c_1 \Rightarrow r_1 = \frac{c_1}{l_{11}}$$

From the 2nd row,

$$l_{21}r_1 + l_{22}r_2 = c_2 \Rightarrow r_2 = \frac{c_2 - l_{21}r_1}{l_{22}}$$

we can get a numerical value for r_2 because we know r_1 .

Gauss Elimination - LU decomposition XII

From the third and fourth rows of Eq. (21) we can get

$$r_3 = \frac{c_3 - l_{31}r_1 - l_{32}r_2}{l_{33}}$$
$$r_4 = \frac{c_4 - l_{41}r_1 - l_{42}r_2 - l_{43}r_3}{l_{44}}$$

In general, the formula for r_i for a $n \times n$ system is

$$r_1 = c_1/l_{11}$$
$$r_i = \frac{\left(c_i - \sum_{j=1}^{i-1} l_{ij}r_j \right)}{l_{ii}} \quad \text{for } i = 2, 3, 4, \dots, n \quad (22)$$

Gauss Elimination - LU decomposition XIII

Lastly, we need to solve

$$[U]\{X\} = \{R\}$$
$$\begin{bmatrix} 1 & u_{12} & u_{13} & u_{14} \\ 0 & 1 & u_{23} & u_{24} \\ 0 & 0 & 1 & u_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{Bmatrix} = \begin{Bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \end{Bmatrix} \quad (23)$$

Again, we use back substitution. Start with last (fourth row)

$$x_4 = r_4 \quad (24)$$

Next, use the 2nd last (third row) to give

$$x_3 + u_{34}x_4 = r_3$$

$$x_3 = r_3 - u_{34}x_4 \quad (25)$$

Gauss Elimination - LU decomposition XIV

You can get a value for x_3 because we know x_4 from Eq. (24).
The second and first row will give

$$x_2 = r_2 - u_{23}x_3 - u_{24}x_4$$

$$x_1 = r_1 - u_{12}x_2 - u_{13}x_3 - u_{14}x_4$$

which can be used to solve for x_2 and x_1 (in that sequence).
The general formula for an $n \times n$ system is

$$x_n = r_n$$

$$x_i = r_i - \sum_{j=i+1}^n u_{ij}x_j \quad \text{for } i = n-1, n-2, n-3, \dots, 2, 1$$

(26)

Gauss Elimination - LU decomposition XV

In above example, the LU decomposition was conducted using a $[U]$ matrix with ones on the diagonal.

This is called the [Crout's](#) method. One can also factorize a matrix by assuming ones along the diagonal of the $[L]$ matrix instead. This is called the [Doolittle's](#) method, which for a 4×4 system, would look like

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ l_{21} & 1 & 0 & 0 \\ l_{31} & l_{32} & 1 & 0 \\ l_{41} & l_{42} & l_{43} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

The algorithm to perform Doolittle's method is very similar to that for Crout's method.

LU decomposition - Example I

Use again the following problem:

$$[A] = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \\ 2 & 3 & 2 \end{bmatrix},$$

$$\{X\} = \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}, \quad \{C\} = \begin{Bmatrix} 14 \\ 16 \\ 17 \end{Bmatrix}.$$

but solve it using L-U decomposition.

LU decomposition - Example II

Use the formulae for getting elements of $[L]$ and $[U]$

$$l_{i1} = a_{i1} \quad \text{for } i = 1, 2, \dots, n$$
$$u_{1j} = \frac{a_{1j}}{l_{11}} \quad \text{for } j = 2, 3, \dots, n$$

Begin for loop $j = 2, 3, \dots, n$

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \quad \text{for } i = j, j+1, \dots, n \quad (27)$$

$$u_{jk} = \frac{a_{jk} - \sum_{i=1}^{j-1} l_{ji} u_{ik}}{l_{jj}} \quad \text{for } k = j+1, j+2, \dots, n$$

End for loop

$$u_{ii} = 1.0 \quad \text{for } i = 1, 2, \dots, n$$

Thus, for $j = 1$:

$$l_{i1} = a_{i1} \quad \text{for } i = 1, 2, 3 \quad \Rightarrow \quad l_{11} = a_{11} = 1$$
$$l_{21} = a_{21} = 3$$
$$l_{31} = a_{31} = 2$$

LU decomposition - Example III

$$u_{1j} = \frac{a_{1j}}{l_{11}} \quad \text{for } j = 2, 3 \quad \Rightarrow \quad u_{12} = \frac{a_{12}}{l_{11}} = 2$$
$$u_{13} = \frac{a_{13}}{l_{11}} = 3$$

Now for $j = 2$:

$$l_{i2} = a_{i2} - \sum_{k=1}^{2-1} l_{ik} u_{k2} \quad \text{for } i = 2, 3$$
$$u_{2k} = \frac{a_{2k} - \sum_{i=1}^{2-1} l_{2i} u_{ik}}{l_{22}} \quad \text{for } k = 3$$

LU decomposition - Example IV

$$\begin{aligned}l_{22} &= a_{22} - l_{21}u_{12} = 2 - 3 \cdot 2 = -4 \\l_{32} &= a_{32} - l_{31}u_{12} = 3 - 2 \cdot 2 = -1 \\u_{23} &= \frac{a_{23} - l_{21}u_{13}}{l_{22}} = \frac{2 - 3 \cdot 3}{-4} = \frac{7}{4}\end{aligned}$$

Now for $j = 3$:

$$\begin{aligned}l_{i3} &= a_{i3} - \sum_{k=1}^{3-1} l_{ik}u_{k3} \quad \text{for } i = 3 \\l_{33} &= a_{33} - (l_{31}u_{13} + l_{32}u_{23}) = 2 - \left(2 \cdot 3 - 1 \cdot \frac{7}{4}\right) = -\frac{9}{4}\end{aligned}$$

Thus we get the upper and lower diagonal matrices

$$[L] = \begin{bmatrix} 1 & 0 & 0 \\ 3 & -4 & 0 \\ 2 & -1 & -9/4 \end{bmatrix}, \quad [U] = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 7/4 \\ 0 & 0 & 1 \end{bmatrix}$$

LU decomposition - Example V

Now solve for $\{R\}$ by forward substitution. From the first row,

$$l_{11}r_1 = c_1 \Rightarrow r_1 = \frac{c_1}{l_{11}} = \frac{14}{1} = 14$$

From the 2nd row,

$$l_{21}r_1 + l_{22}r_2 = c_2 \Rightarrow r_2 = \frac{c_2 - l_{21}r_1}{l_{22}} = \frac{16 - 3 \cdot 14}{-4} = 6.5$$

From the third row,

$$r_3 = \frac{c_3 - l_{31}r_1 - l_{32}r_2}{l_{33}} = \frac{17 - 2 \cdot 14 + 6.5}{-9/4} = 2$$

Lastly, we need to solve

$$[U]\{X\} = \{R\}$$

We can now use back substitution. Start with last (third) row

$$x_3 = r_3 = 2$$

$$x_2 = r_2 - u_{23}x_3 = 6.5 - \frac{7}{4} \cdot 2 = 3$$

$$x_1 = r_1 - u_{12}x_2 - u_{13}x_3 = 14 - 2 \cdot 3 - 3 \cdot 2 = 2 .$$

LU decomposition - Example VI

The solution is therefore

$$\{X\} = \begin{Bmatrix} 2 \\ 3 \\ 2 \end{Bmatrix}.$$

Gauss Elimination - Cost

Cost of LU decomposition

To solve the system $[A] \{X\} = \{C\}$, the number of operations is:

1. decompose $[A]$ into $[L][U]$: $2/3n^3$ operations
2. solve $[L]\{R\} = \{C\}$: n^2 operations
3. solve $[U]\{X\} = \{R\}$: n^2 operations

Compare that with directly inverting matrix $[A]$:

1. compute $[A]^{-1}$: $2n^3$ operations
2. multiply $\{X\} = [A]^{-1}\{C\}$: $2n^2$ operations

Iterative Methods

If the size of $[A]$ becomes very large, the cost of direct methods can be prohibitive (scales as n^3).

Alternative → [iterative methods](#).

Let's rewrite $[A]$ as

$$[A] = [M] - [N].$$

With this, our system of equations ($[A]\{X\} = \{C\}$) becomes

$$[M]\{X\} = [N]\{X\} + \{C\} \tag{28}$$

Iterative Methods

An iterative method of solving for the vector $\{X\}$ would be

$$[M] \{X\}^{(k+1)} = [N] \{X\}^{(k)} + \{C\}, \quad (29)$$

where $\{X\}^{(k)}$ is current (k 'th) guess of true solution, $\{X\}$.

We would like to choose the matrix $[M]$ such that Eq. (29) is easier to solve than the original expression $[A] \{X\} = \{C\}$.

Here, we will cover two different methods:

1. **Point Jacobi method:** $[M]$ consists of only the diagonal elements of $[A]$.
2. **Gauss–Seidel method:** $[M]$ consists of the lower triangular elements of $[A]$.

Iterative Methods

The general **procedure** to solving linear systems of equations with an iterative method is

1. Define the $[M]$ and $[N]$ matrices
2. Obtain an initial guess, $\{X\}^{(0)}$
3. Solve $[M] \{X\}^{(k+1)} = [N] \{X\}^{(k)} + \{C\}$ for new guess
4. Define and obtain maximum absolute value of residual vector,
 $\{r\} = [A] \{X\}^{(k)} - \{C\}$
If maximum value of $|\{r\}| > \varepsilon$, repeat step 3.
Else, end.

Iterative Methods - Point Jacobi I

In the point Jacobi method, $[M]$ consist of only the diagonal elements of $[A]$. The negative of all other elements is placed into $[N]$. This method will be illustrated with a 3×3 system

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \end{Bmatrix}$$

$[M]$ and $[N]$ are

$$[M] = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix}, \quad [N] = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{bmatrix}$$

For this case, the procedure to solve the system will look like

$$\begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ 0 & 0 & a_{33} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}^{(k+1)} = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ -a_{21} & 0 & -a_{23} \\ -a_{31} & -a_{32} & 0 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix}^{(k)} + \begin{Bmatrix} c_1 \\ c_2 \\ c_3 \end{Bmatrix}$$

Iterative Methods - Point Jacobi II

Solving this system of equation will give

$$x_1^{(k+1)} = \frac{1}{a_{11}} (-a_{12}x_2^{(k)} - a_{13}x_3^{(k)} + c_1)$$

$$x_2^{(k+1)} = \frac{1}{a_{22}} (-a_{21}x_1^{(k)} - a_{23}x_3^{(k)} + c_2)$$

$$x_3^{(k+1)} = \frac{1}{a_{33}} (-a_{31}x_1^{(k)} - a_{32}x_2^{(k)} + c_3)$$

From this we see that if $[A]$ is an $N \times N$ matrix, then the generic formula for $x_i^{(k+1)}$ is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(\sum_{j=1, j \neq i}^N -a_{ij}x_j^{(k)} + c_i \right)$$

Iterative Methods - Gauss–Seidel I

The Gauss–Seidel method is the most commonly used iterative method. For conciseness, the method will be illustrated with a 3×3 system

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = c_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 = c_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 = c_3$$

For this method, the matrix $[M]$ consists of only the lower triangular elements of $[A]$.

$[M]$ will contain the diagonal elements of $[A]$

$$[M] = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Iterative Methods - Gauss–Seidel II

The negative of all other elements is placed in $[N]$.

$$[N] = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{bmatrix}$$

Thus,

$$\begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \left\{ \begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array} \right\}^{(k+1)} = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{bmatrix} \left\{ \begin{array}{c} x_1 \\ x_2 \\ x_3 \end{array} \right\}^{(k)} + \left\{ \begin{array}{c} c_1 \\ c_2 \\ c_3 \end{array} \right\}$$

Solving this system in sequence from the top row to the bottom row will give you

$$x_1^{(k+1)} = \frac{c_1 - a_{12}x_2^{(k)} - a_{13}x_3^{(k)}}{a_{11}} \quad (30)$$

Iterative Methods - Gauss–Seidel III

$$x_2^{(k+1)} = \frac{c_2 - a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)}}{a_{22}} \quad (31)$$

$$x_3^{(k+1)} = \frac{c_3 - a_{31}x_1^{(k+1)} - a_{32}x_2^{(k+1)}}{a_{33}} \quad (32)$$

We can now start the iterative process. **WARNING**, you have to be careful that none of your a_{ii} 's are zero !!

1. Assume initial guess values for x_1 , x_2 , and x_3 : $x_1^{(0)}$, $x_2^{(0)}$ and $x_3^{(0)}$.
2. Substitute the guess values for $x_2^{(0)}$, and $x_3^{(0)}$ into Eq. (30) to obtain a better estimate for x_1 . Call this value $x_1^{(1)}$.

Iterative Methods - Gauss–Seidel IV

3. Substitute $x_1^{(1)}$ and $x_3^{(0)}$ into Eq. (31) to obtain a new estimate for $x_2 = x_2^{(1)}$.
4. Use $x_1^{(1)}$ and $x_2^{(1)}$ in Eq. (32) to obtain a better estimate for $x_3 = x_3^{(1)}$.

Iterative Methods - Gauss–Seidel V

$$\left. \begin{array}{l} x_1 = \frac{c_1 - a_{12}x_2 - a_{13}x_3}{a_{11}} \\ x_2 = \frac{c_2 - a_{21}x_1 - a_{23}x_3}{a_{22}} \\ x_3 = \frac{c_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} \end{array} \right\} \text{First iteration}$$
$$\left. \begin{array}{l} x_1 = \frac{c_1 - a_{12}\hat{x}_2 - a_{13}\hat{x}_3}{a_{11}} \\ x_2 = \frac{c_2 - a_{21}\hat{x}_1 - a_{23}\hat{x}_3}{a_{22}} \\ x_3 = \frac{c_3 - a_{31}\hat{x}_1 - a_{32}\hat{x}_2}{a_{33}} \end{array} \right\} \text{Second iteration}$$

Figure: Flow of information in the Gauss–Seidel method.

Iterative Methods - Gauss–Seidel VI

If $[A]$ is an $N \times N$ matrix, then the generic formula for $x_i^{(k+1)}$ is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(- \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^N a_{ij} x_j^{(k)} + c_i \right)$$

Note that one can extend the point Jacobi and Gauss–Seidel methods to solve nonlinear systems too !

Convergence of Iterative Methods I

Which method converges faster ?

To answer that question, consider under what conditions approximated solution, $\{X\}^{(k)}$, converges to exact solution $\{X\}$.

Define error matrix at the k 'th iteration to be

$$\{\epsilon\}^{(k)} = \{X\}^{(k)} - \{X\}$$

Subtract

$$[M] \{X\} = [N] \{X\} + \{C\}$$

from

$$[M] \{X\}^{(k+1)} = [N] \{X\}^{(k)} + \{C\}$$

Convergence of Iterative Methods II

We get

$$\begin{aligned}\{\epsilon\}^{(k+1)} &= [M]^{-1} [N] \{\epsilon\}^{(k)} \\ \{\epsilon\}^{(k+1)} &= [P] \{\epsilon\}^{(k)}, \quad \text{where } [P] = [M]^{-1} [N]\end{aligned}$$

Thus

$$\{\epsilon\}^{(k+1)} = [P]^{k+1} \{\epsilon\}^{(0)},$$

i.e. error at the $k + 1$ 'th iteration is related to initial error.

Now assume that all eigenvalues of the matrix $[P]$ are linearly independent, then

$$[P] = [S] [\Lambda] [S]^{-1}$$

where columns of $[S]$ are eigenvectors of $[P]$ and $[\Lambda]$ is a diagonal matrix whose elements are the eigenvalues of $[P]$.

Convergence of Iterative Methods III

Thus we can write the error at the k 'th iteration as

$$\{\epsilon\}^{(k+1)} = [S] [\Lambda]^{k+1} [S]^{-1} \{\epsilon\}^{(0)}$$

So in order for the error to go to zero as fast as possible, we would need the magnitude of all the eigenvalues of $[P]$ to be less than 1.

Thus when

$$[\Lambda] = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix}$$

we require

$$|\lambda_i| < 1 .$$

Convergence of Iterative Methods

Let's again consider our example

$$[A] = \begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \\ 2 & 3 & 2 \end{bmatrix}$$

When using the Gauss–Seidel approach, we get $[M]$ as

$$[M] = \begin{bmatrix} a_{11} & 0 & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 2 & 0 \\ 2 & 3 & 2 \end{bmatrix}$$

$[N]$ is computed as $[N] = [M] - [A]$, thus

$$[N] = \begin{bmatrix} 0 & -a_{12} & -a_{13} \\ 0 & 0 & -a_{23} \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & -2 & -3 \\ 0 & 0 & -2 \\ 0 & 0 & 0 \end{bmatrix}$$

Convergence of Iterative Methods

$$[M]^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -1.5 & 0.5 & 0 \\ 1.25 & -0.75 & 0.5 \end{bmatrix}$$

Can now compute $[P] = [M]^{-1} [N]$

$$[P] = \begin{bmatrix} 0 & -2.0 & -3.0 \\ 0 & 3.0 & 3.5 \\ 0 & -2.5 & -2.25 \end{bmatrix}$$

The eigenvalues of $[P]$ are

$$[\Lambda] = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.375 + 1.3636i & 0 \\ 0 & 0 & 0.375 + 1.3636i \end{bmatrix}$$

Therefore, $|\lambda_{i=2,3}| = \sqrt{2} > 1$, and thus the method will not converge for this choice of $[A]$.

Convergence of Iterative Methods

For the example used in workshop/tutorial

$$[A] = \begin{bmatrix} 4 & -1 & -1 \\ -2 & 6 & 1 \\ -1 & 1 & 7 \end{bmatrix}$$

we get

$$[P] = \begin{bmatrix} 0 & 0.25 & 0.25 \\ 0 & 0.0833 & -0.0833 \\ 0 & 0.0238 & 0.0476 \end{bmatrix}$$

$$[\Lambda] = \begin{bmatrix} \lambda_1 & 0 & 0 \\ 0 & \lambda_2 & 0 \\ 0 & 0 & \lambda_3 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.0655 + 0.0408i & 0 \\ 0 & 0 & 0.0655 + 0.0408i \end{bmatrix}$$

Here, $|\lambda_{i=2,3}| = 0.0772 < 1$, and thus method will converge.

Cost - direct vs iterative

Iterative methods

To solve the system $[A] \{X\} = \{C\}$, the number of operations of iterative methods can be estimated as follows: Each step of

$$x_i^{(k+1)} = x_i^k + \frac{1}{a_{ii}} \left(\sum_{j=1}^n a_{ij} x_j^{(k)} \right)$$

requires n multiplications and one division, i.e. $n + 1$ operations.
Required for n components of x_i , $\Rightarrow n(n + 1) \approx n^2$ operations

Total number of operations approximately n^2 times **number of iterations**.

Cost - direct vs iterative

Cost of Iterative methods

$$\approx n^2 k \text{ operations}$$

Cost of LU decomposition:

Decomposition $[A]$ into $[L][U]$: $2/3n^3$ operations

Iterative methods are more efficient when $k < n$.

Assume computer takes 10^{-9} s/operation, $k < 20$:

$n = 1,000$: direct method: 0.66s, iterative: 0.02s

$n = 10,000$: direct method: 666s, iterative: 2s

$n = 100,000$: direct method: 185h, iterative: 200s

Newton's Method for nonlinear equations

When discussing finding roots, we found the roots for two nonlinear equations.

What if we want to find roots of 3 or more nonlinear equations?

The linear algebra methods that we have discussed can only cope with linear equations.

For simplicity, the theory for solving a system of 3 or more simultaneous nonlinear equations will be developed for 3×3 system. Extension to larger systems is straight forward.

Newton's Method for nonlinear equations

Consider system of 3 simultaneous nonlinear equations

$$u(x, y, z) = 0 \quad (33)$$

$$v(x, y, z) = 0 \quad (34)$$

$$w(x, y, z) = 0 \quad (35)$$

Perform Taylor-series expansion on the left hand side and truncate after first-order terms to get

$$\begin{aligned} u(x_{i+1}, y_{i+1}, z_{i+1}) &\approx u(x_i, y_i, z_i) + u_x(x_i, y_i, z_i)(x_{i+1} - x_i) \\ &+ u_y(x_i, y_i, z_i)(y_{i+1} - y_i) + u_z(x_i, y_i, z_i)(z_{i+1} - z_i) = 0 \\ v(x_{i+1}, y_{i+1}, z_{i+1}) &\approx v(x_i, y_i, z_i) + v_x(x_i, y_i, z_i)(x_{i+1} - x_i) \\ &+ v_y(x_i, y_i, z_i)(y_{i+1} - y_i) + v_z(x_i, y_i, z_i)(z_{i+1} - z_i) = 0 \\ w(x_{i+1}, y_{i+1}, z_{i+1}) &\approx w(x_i, y_i, z_i) + w_x(x_i, y_i, z_i)(x_{i+1} - x_i) \\ &+ w_y(x_i, y_i, z_i)(y_{i+1} - y_i) + w_z(x_i, y_i, z_i)(z_{i+1} - z_i) = 0 \end{aligned}$$

Newton's Method for nonlinear equations

$$\begin{aligned} u_x &= \frac{\partial u}{\partial x}, u_y = \frac{\partial u}{\partial y}, u_z = \frac{\partial u}{\partial z} \\ v_x &= \frac{\partial v}{\partial x}, v_y = \frac{\partial v}{\partial y}, v_z = \frac{\partial v}{\partial z} \\ w_x &= \frac{\partial w}{\partial x}, w_y = \frac{\partial w}{\partial y}, w_z = \frac{\partial w}{\partial z} \end{aligned}$$

We can put the above system of equations into matrix form

$$\begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ w_x & w_y & w_z \end{bmatrix} \begin{Bmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \\ z_{i+1} - z_i \end{Bmatrix} = - \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \quad (36)$$

Equation (36) is a linear equation so we should be able to use any of the methods discussed earlier to solve it !

Newton's Method for nonlinear equations

Hence, to solve the original system of equations, do the following

1. Calculate all the required derivatives.
2. Guess the (initial) values for x_i , y_i and z_i .
3. Solve Eq. (36) to obtain $(x_{i+1} - x_i)$, $(y_{i+1} - y_i)$ and $(z_{i+1} - z_i)$. You can use Gauss elimination, LU decomposition, Gauss–Seidel etc.
4. Calculate x_{i+1} , y_{i+1} , z_{i+1} .
5. Repeat steps 2, 3 and 4 until converged solution obtained.

This method is a more general version of the Newton–Raphson method mentioned earlier.

Nonlinear equations - Example

It is instructive to look at how to solve the following example:
Solve the following set of nonlinear algebraic equations

$$3x + \cos(yz) = \frac{1}{2}$$

$$x^2 - 81(y + 0.1)^2 + \sin(z) = -1.06$$

$$e^{-xy} + 20z + \frac{10\pi - 3}{3} = 0$$

To use the Newton–Raphson method, we define the functions $u(x, y, z)$, $v(x, y, z)$, $w(x, y, z)$:

$$u(x, y, z) = 3x + \cos(yz) - \frac{1}{2}$$

$$v(x, y, z) = x^2 - 81(y + 0.1)^2 + \sin(z) + 1.06$$

$$w(x, y, z) = e^{-xy} + 20z + \frac{10\pi - 3}{3}.$$

Nonlinear equations - Example

The corresponding partial derivatives are

$$\begin{aligned}\frac{\partial u}{\partial x} &= 3 & \frac{\partial u}{\partial y} &= -z \sin(yz) & \frac{\partial u}{\partial z} &= -y \sin(yz) \\ \frac{\partial v}{\partial x} &= 2x & \frac{\partial v}{\partial y} &= -16.2 - 162y & \frac{\partial v}{\partial z} &= \cos(z) \\ \frac{\partial w}{\partial x} &= -ye^{-xy} & \frac{\partial w}{\partial y} &= -xe^{-xy} & \frac{\partial w}{\partial z} &= 20\end{aligned}$$

Initial guess $(x_0, y_0, z_0) = (0, 0, 0)$, substitute into Eq. (36):

$$\begin{bmatrix} 3.0 & 0.0 & 0.0 \\ 0.0 & -16.2 & 1.0 \\ 0.0 & 0.0 & 20.0 \end{bmatrix} \begin{Bmatrix} x_1 - x_0 \\ y_1 - y_0 \\ z_1 - z_0 \end{Bmatrix} = \begin{bmatrix} -0.5000000000 \\ -0.25 \\ -10.47197551 \end{bmatrix}$$

Nonlinear equations - Example

$$\begin{bmatrix} 3.0 & 0.0046301 & 0.00014934 \\ -0.33334 & -13.464 & 0.86602 \\ 0.016841 & 0.16619 & 20.0 \end{bmatrix} \begin{Bmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{Bmatrix} = \begin{bmatrix} 0.000039 \\ -0.02827 \\ 0.0028 \end{bmatrix}$$

$$\begin{bmatrix} 3.0 & 0.0040504 & 0.00011437 \\ -0.33331 & -13.805 & 0.86608 \\ 0.014744 & 0.166246 & 20.0 \end{bmatrix} \begin{Bmatrix} x_3 - x_2 \\ y_3 - y_2 \\ z_3 - z_2 \end{Bmatrix} = \begin{bmatrix} 0.00000061 \\ 0.000359 \\ -0.0000 \end{bmatrix}$$

and so on. With every iteration, the right hand side becomes smaller, until it meets convergence criterion (ε) and

$$x_{n+1} = x_n = -0.16667$$

$$y_{n+1} = y_n = -0.01481$$

$$z_{n+1} = z_n = -0.523476 .$$

Next week

Least Squares Approximation



THE UNIVERSITY OF

MELBOURNE



ENGR30003 Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne
Slides developed by Prof Richard Sandberg

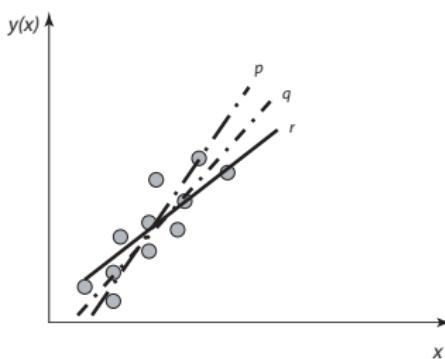
This week

LECTURE 15

Least Squares Approximation

Least Squares Approximations

Suppose you get a set of x, y values from an experiment.



How can you draw a line that best fits through the data ?

Which of lines p , q or r best fit data?

A method to systematically calculating the line of best fit is called
Least Squares approximation.

Linear Least Squares Approximation

Let's represent any one of the lines in previous figure as

$$y = ax + b \quad (1)$$

Task:

Find a and b such that line predicted by Eq. (1) is as close to actual values of $y_i(x_i)$.

- Define the error between the exact and approximated values as:

$$e_i = \underbrace{y_i}_{\text{actual value}} - \underbrace{(ax_i + b)}_{\text{predicted value}}$$

Linear Least Squares Approximation

- With N data points, compute square error at every point and sum all up to find total error:

$$\begin{aligned} S &= e_1^2 + e_2^2 + e_3^2 + e_4^2 + \dots + e_N^2 \\ &= \sum_{i=1}^N e_i^2 \\ S &= \sum_{i=1}^N (y_i - ax_i - b)^2 \end{aligned}$$

- Now find a and b , such that S is minimized.

How can we minimize S ?

⇒ Compute dS/da , dS/db , set derivatives to zero.

Here x_i , y_i constants (data points) when differentiating.

Linear Least Squares Approximation

$$\begin{aligned}\frac{\partial S}{\partial a} &= \sum_{i=1}^N 2(y_i - ax_i - b)(-x_i) = 0 \\ \frac{\partial S}{\partial b} &= \sum_{i=1}^N 2(y_i - ax_i - b)(-1) = 0\end{aligned}\tag{2}$$

Eq. (2) can be re-written as

$$\begin{aligned}a \sum_{i=1}^N x_i^2 + b \sum_{i=1}^N x_i &= \sum_{i=1}^N x_i y_i \\ a \sum_{i=1}^N x_i + bN &= \sum_{i=1}^N y_i\end{aligned}$$

Linear Least Squares Approximation

This is a system of 2×2 linear equations. They can be solved to get a and b .

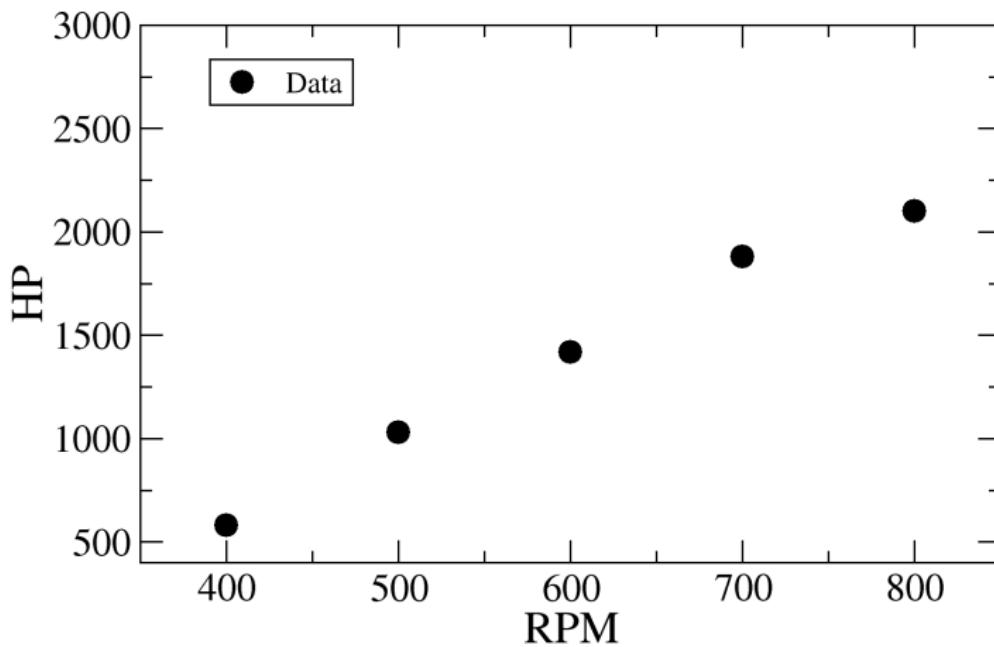
$$\begin{bmatrix} \sum_{i=1}^N x_i^2 & \sum_{i=1}^N x_i \\ \sum_{i=1}^N x_i & N \end{bmatrix} \begin{Bmatrix} a \\ b \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^N x_i y_i \\ \sum_{i=1}^N y_i \end{Bmatrix}$$

Linear Least Squares Approximation - Example 1

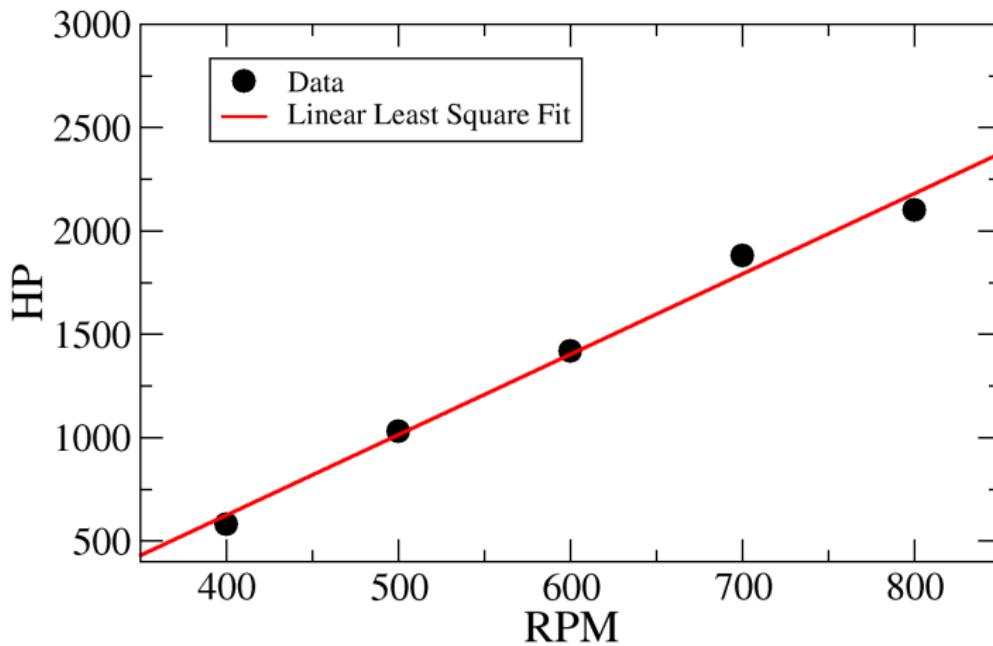
Fit a straight line to the given points (x, y) by the method of least squares.

RPM (x)	Diesel engine HP (y)
400	580
500	1030
600	1420
700	1880
800	2100

Linear Least Squares Approximation - Example 1

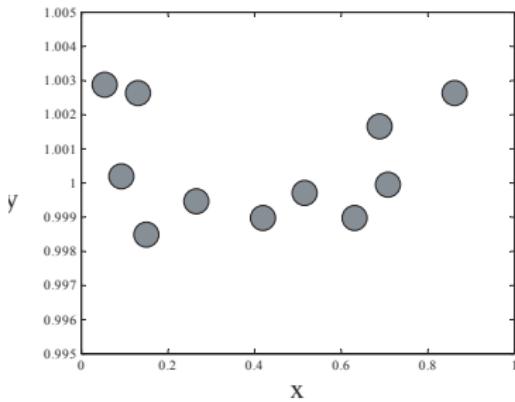


Linear Least Squares Approximation - Example 1



Polynomial Least Squares Approximation

Imagine you get following data from an experiment:



Can we draw a linear function to fit the data?

Clearly not!

For this data, it is more appropriate to fit a **polynomial** through it.

Polynomial Least Squares Approximation

Assume functional relationship of form

$$y = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

The error between the actual data and the polynomial is again defined as

$$e_i = \underbrace{y_i}_{\text{actual value}} - \underbrace{(a_0 + a_1x_i + a_2x_i^2 + \dots + a_nx_i^n)}_{\text{predicted value}}$$

As before, square errors and sum them

$$S = \sum_{i=1}^N e_i^2 = \sum_{i=1}^N (y_i - a_0 - a_1x_i - a_2x_i^2 - \dots - a_nx_i^n)^2 \quad (3)$$

This is the total error, that we want to minimize.

Polynomial Least Squares Approximation

How can we minimize S ?

⇒ Compute dS/da_i , set derivatives to zero.

Again, x_i, y_i are constants (data points) when differentiating.

$$\begin{aligned}\frac{\partial S}{\partial a_0} &= \sum_{i=1}^N 2(y_i - a_0 - a_1x_i - a_2x_i^2 - \dots - a_nx_i^n)(-1) = 0, \\ \frac{\partial S}{\partial a_1} &= \sum_{i=1}^N 2(y_i - a_0 - a_1x_i - a_2x_i^2 - \dots - a_nx_i^n)(-x_i) = 0, \\ \frac{\partial S}{\partial a_2} &= \sum_{i=1}^N 2(y_i - a_0 - a_1x_i - a_2x_i^2 - \dots - a_nx_i^n)(-x_i^2) = 0, \quad (4) \\ &\vdots \\ &\vdots \\ \frac{\partial S}{\partial a_n} &= \sum_{i=1}^N 2(y_i - a_0 - a_1x_i - a_2x_i^2 - \dots - a_nx_i^n)(-x_i^n) = 0,\end{aligned}$$

Polynomial Least Squares Approximation

Write in matrix form:

$$\begin{bmatrix} N & \sum x_i & \sum x_i^2 & \sum x_i^3 & \cdots & \sum x_i^n \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \cdots & \sum x_i^{n+1} \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \cdots & \sum x_i^{n+2} \\ \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \sum x_i^6 & \cdots & \sum x_i^{n+3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sum x_i^n & \sum x_i^{n+1} & \sum x_i^{n+2} & \sum x_i^{n+3} & \cdots & \sum x_i^{2n} \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \\ \sum x_i^3 y_i \\ \vdots \\ \sum x_i^n y_i \end{Bmatrix}$$

or

$$[XX] \{A\} = \{XY\}$$

This system of linear equations can be solved with tools such as **Gauss elimination** or **LU decomposition** to give the a_i 's.

Note: here the solution vector is $\{A\}$ and $\{X\}$ contains the known data points that fill matrix $[XX]$.

Quadratic Least Squares Approximation I

Assume we have N data pairs (x, y) .

We want to use the polynomial $y = a_0 + a_1x + a_2x^2$.

The error between the actual data and the polynomial is then defined as

$$e_i = \underbrace{y_i}_{\text{actual value}} - \underbrace{(a_0 + a_1x_i + a_2x_i^2)}_{\text{predicted value}}$$

Square errors and sum them

$$S = \sum_{i=1}^N e_i^2 = \sum_{i=1}^N (y_i - a_0 - a_1x_i - a_2x_i^2)^2$$

This is the total error, that we want to minimize.

Quadratic Least Squares Approximation II

Compute dS/da_i , set derivatives to zero, using x_i, y_i as constants (data points) when differentiating.

$$\frac{\partial S}{\partial a_0} = \sum_{i=1}^N 2(y_i - a_0 - a_1x_i - a_2x_i^2)(-1) = 0,$$

$$\frac{\partial S}{\partial a_1} = \sum_{i=1}^N 2(y_i - a_0 - a_1x_i - a_2x_i^2)(-x_i) = 0,$$

$$\frac{\partial S}{\partial a_2} = \sum_{i=1}^N 2(y_i - a_0 - a_1x_i - a_2x_i^2)(-x_i^2) = 0,$$

Write in matrix form:

$$\begin{bmatrix} N & \sum x_i & \sum x_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \end{Bmatrix} = \begin{Bmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \end{Bmatrix}$$

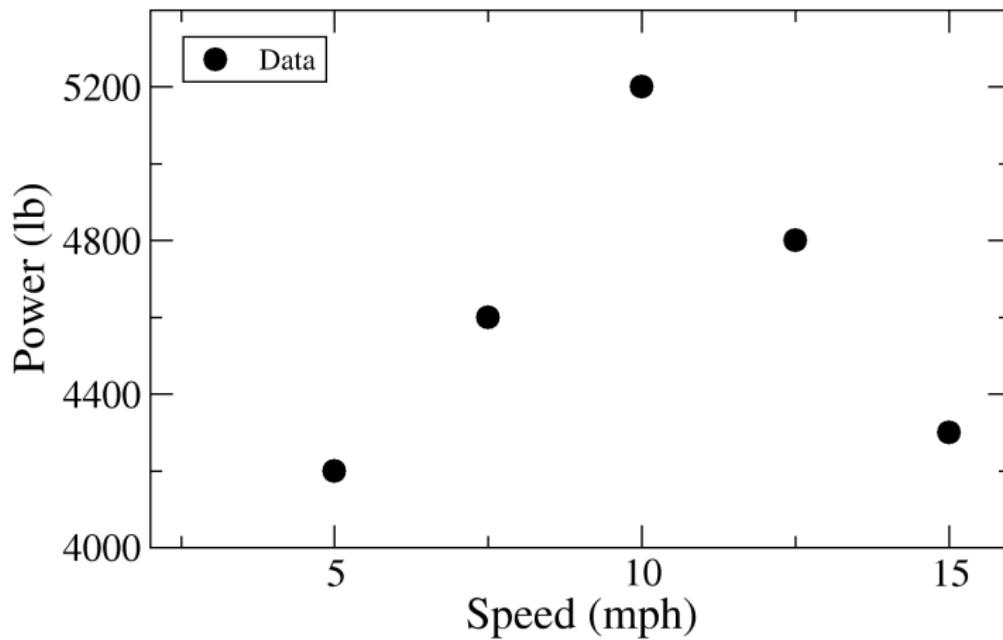
Solve this system of linear equations using to give the a_i 's (e.g. [Gauss elimination](#)).

Quadratic Least Squares Approximation -Example

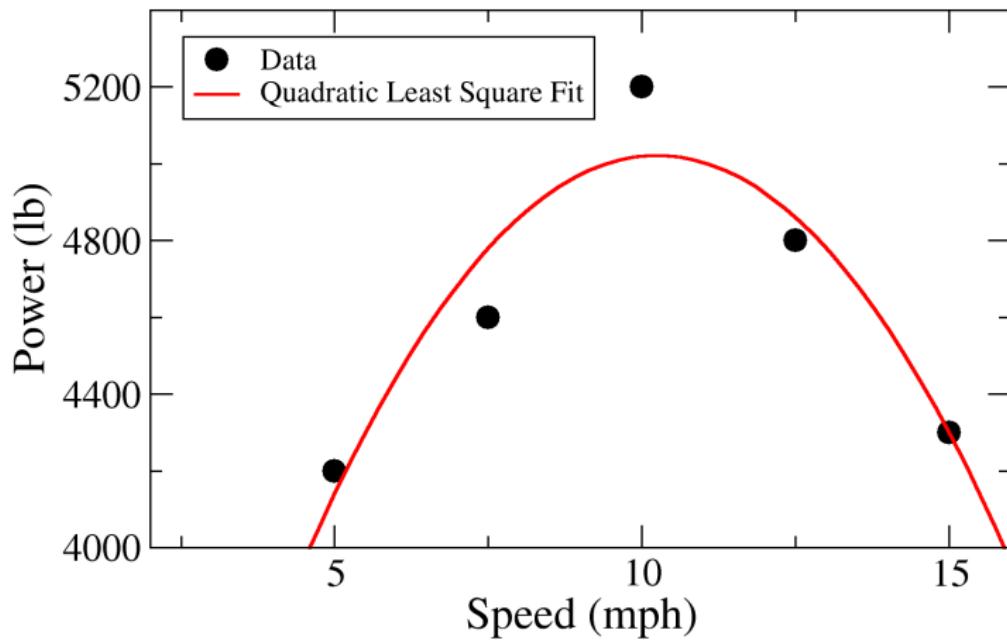
Fit a parabola to the given points (x, y) by the method of least squares.

Speed of snowplow mph (x)	Power of plow in lb (y)
5	4200
7.5	4600
10	5200
12.5	4800
15	4300

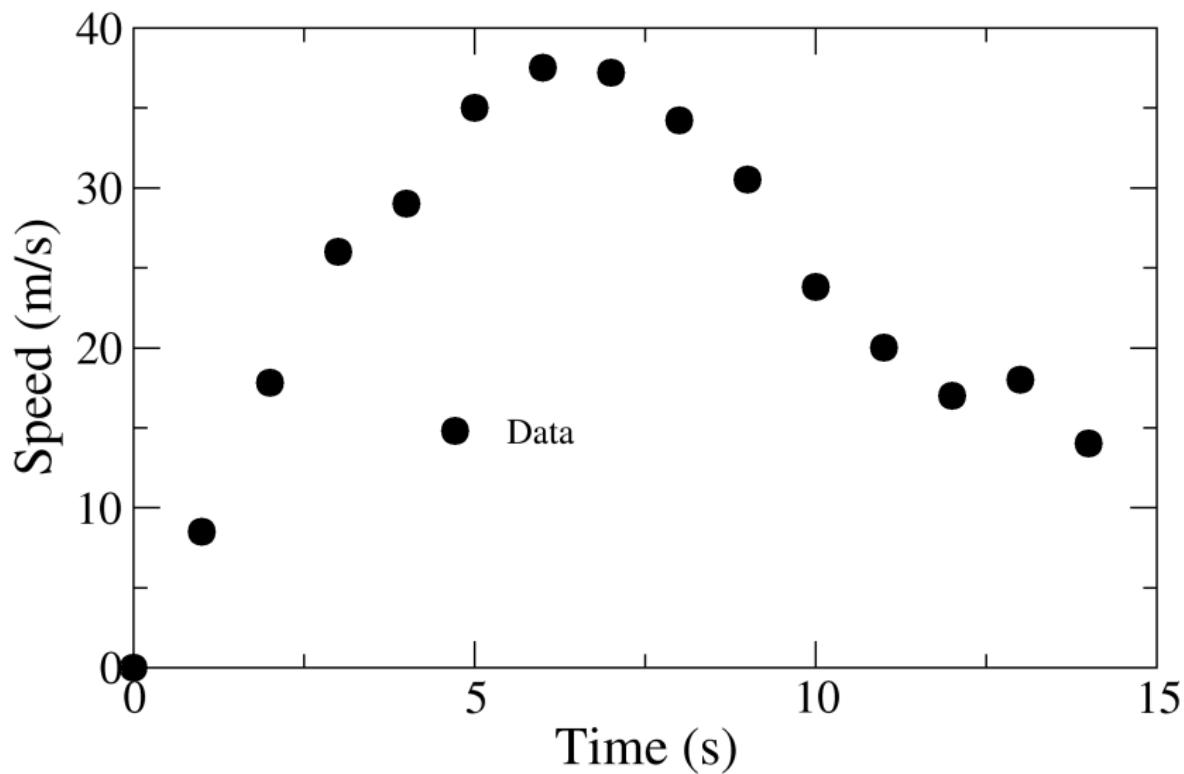
Quadratic Least Squares Approximation -Example



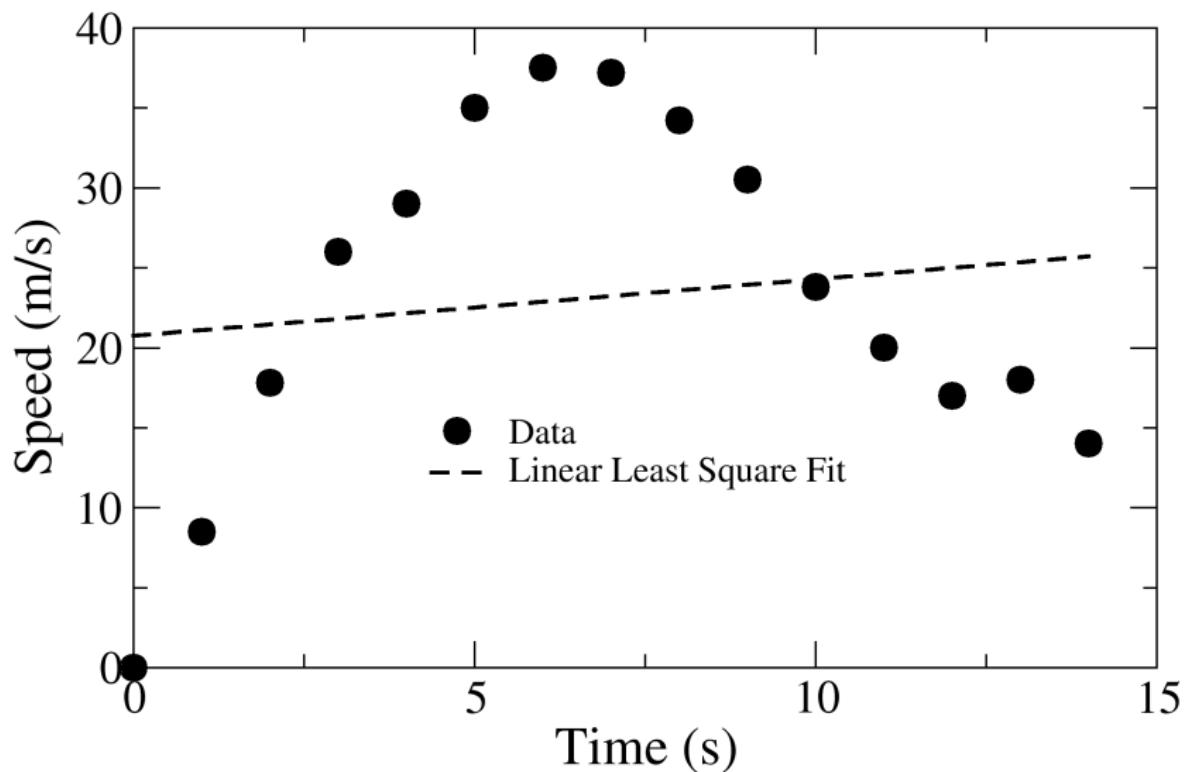
Quadratic Least Squares Approximation -Example



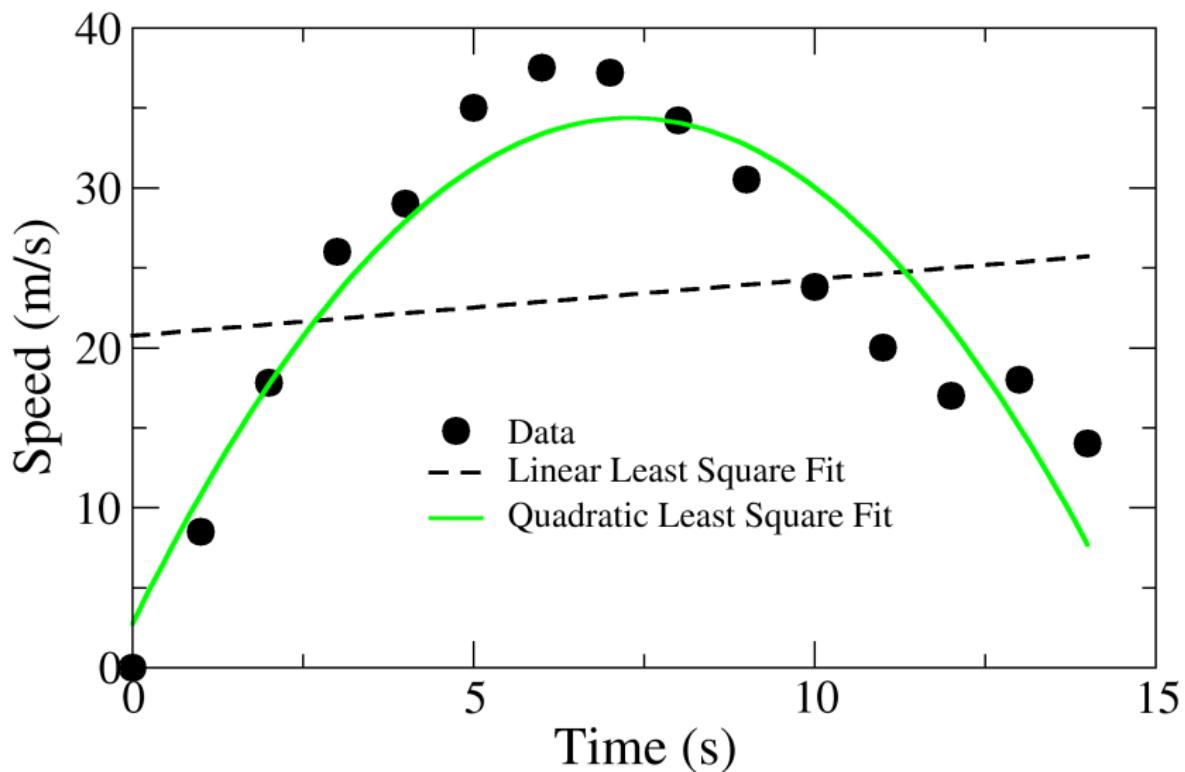
Polynomial Least Squares Approximation



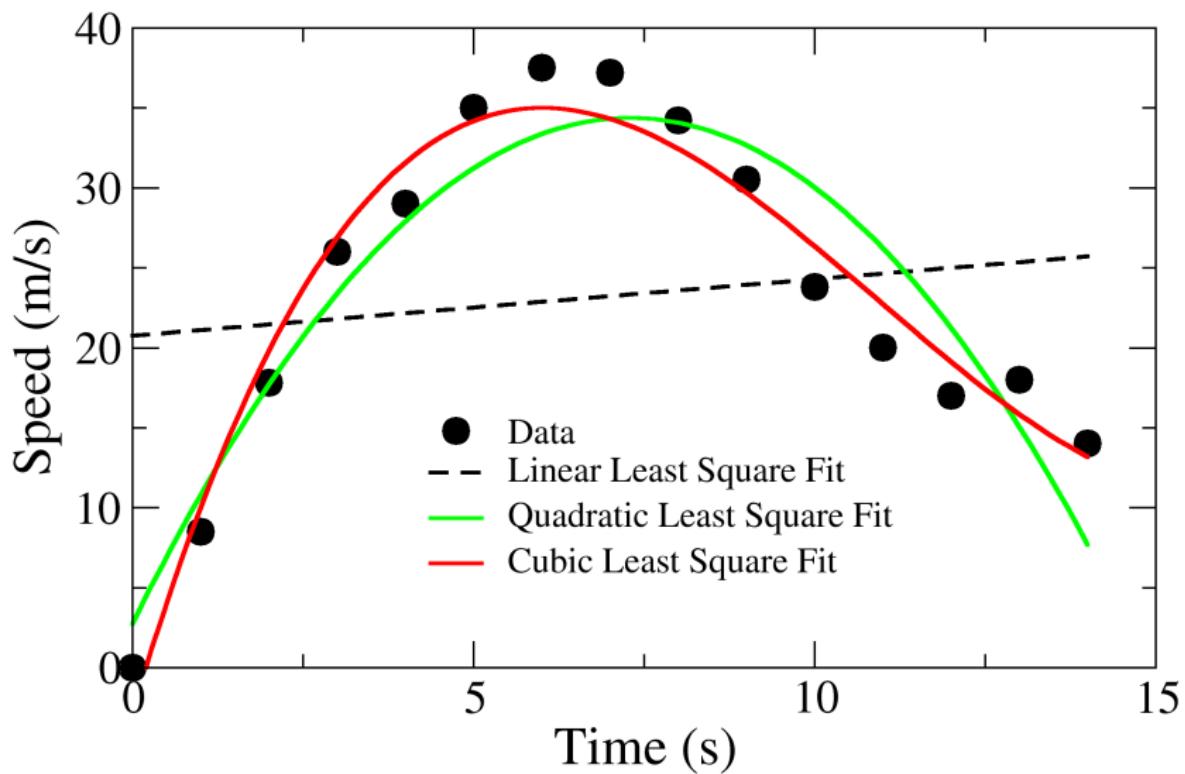
Polynomial Least Squares Approximation



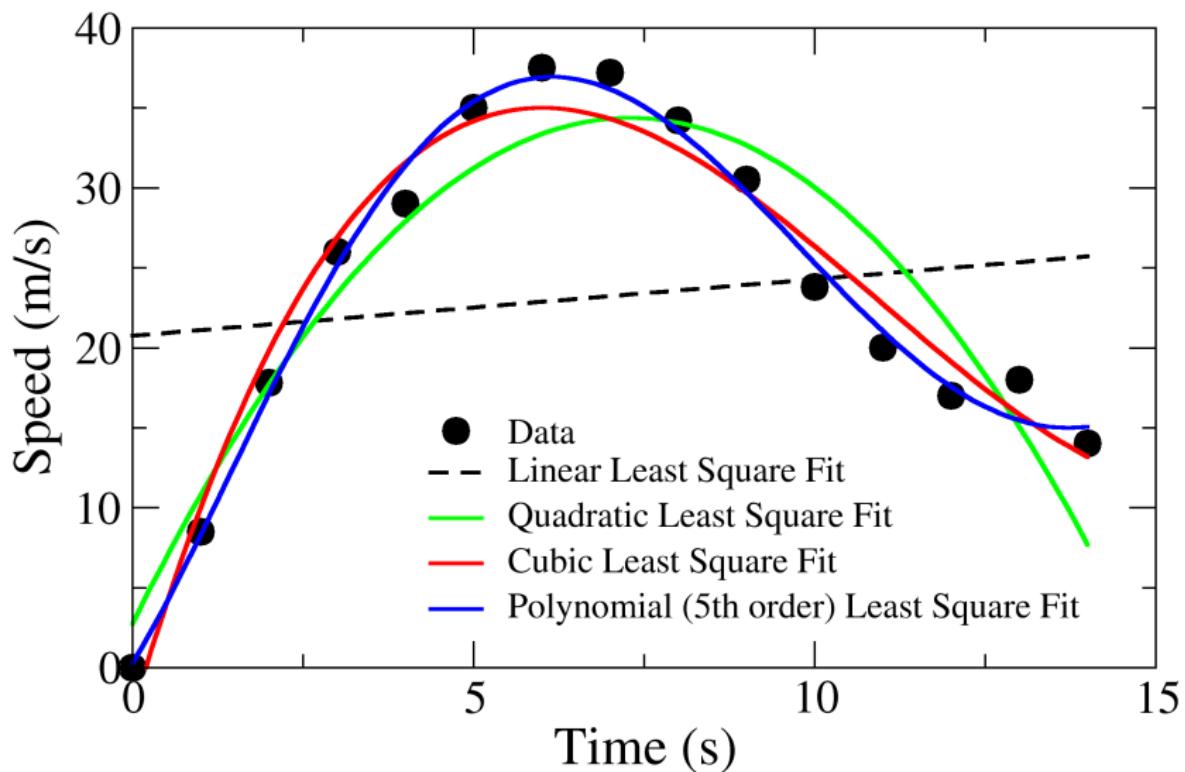
Polynomial Least Squares Approximation



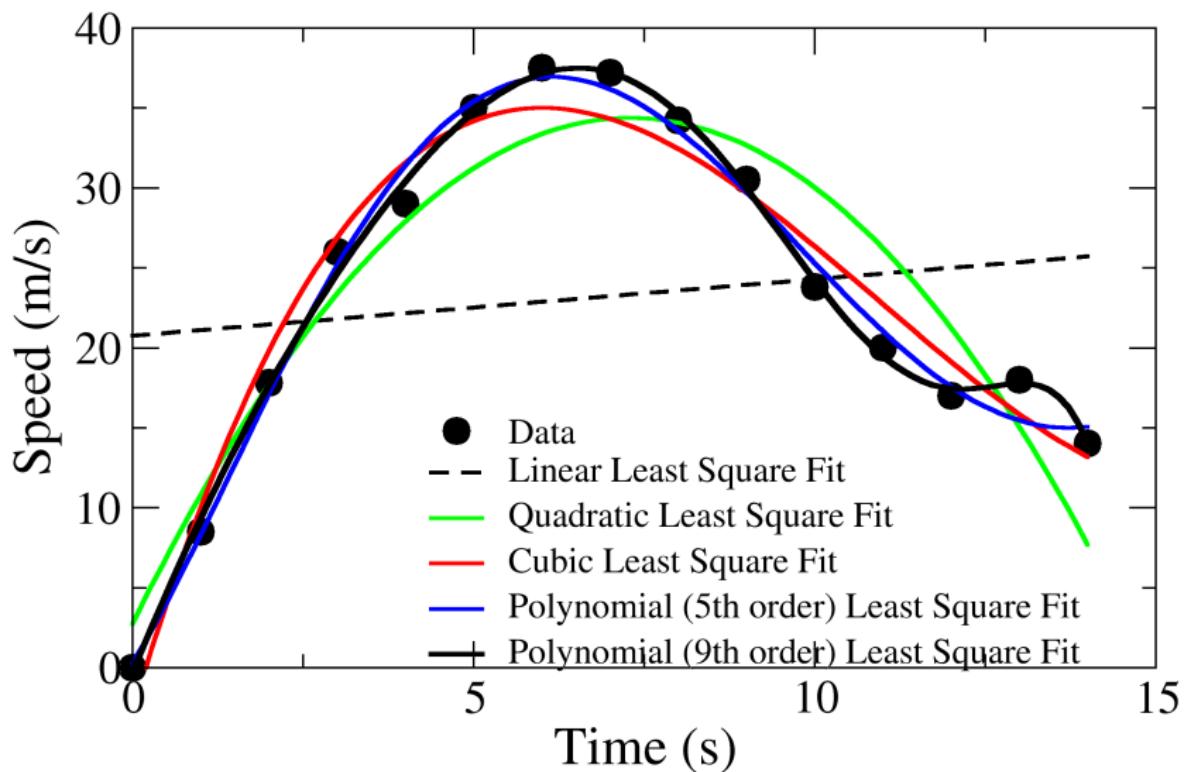
Polynomial Least Squares Approximation



Polynomial Least Squares Approximation



Polynomial Least Squares Approximation



Other Least Squares Approximations

So far only considered polynomial least squares approximations
(linear LSQ is a subset).

Occasionally, it might be appropriate to use other 'basis' functions,
e.g.

- Exponential: $y = be^{ax}$
- Power laws: $y = bx^a$

Other Least Squares Approximations

When using exponential basis, we get the error between the actual data and the approximation

$$e_i = \underbrace{y_i}_{\text{actual value}} - \underbrace{be^{ax_i}}_{\text{predicted value}}$$

As before, square errors and sum them

$$S = \sum_{i=1}^N e_i^2 = \sum_{i=1}^N (y_i - be^{ax_i})^2 \quad (5)$$

This is the total error, that we want to minimize.

Other Least Squares Approximations

How can we minimize S ?

⇒ Compute dS/da and dS/db and set derivatives to zero.

Again, x_i, y_i are constants (data points) when differentiating.

$$\begin{aligned}\frac{\partial S}{\partial a} &= \sum_{i=1}^N 2(y_i - be^{ax_i})(-bx_ie^{ax_i}) = 0, \\ \frac{\partial S}{\partial b} &= \sum_{i=1}^N 2(y_i - be^{ax_i})(-e^{ax_i}) = 0,\end{aligned}\tag{6}$$

Unfortunately, no exact solution to this system can be found.

Other Least Squares Approximations

When using power law basis, we get the error between the actual data and the approximation

$$e_i = \underbrace{y_i}_{\text{actual value}} - \underbrace{bx_i^a}_{\text{predicted value}}$$

As before, square errors and sum them

$$S = \sum_{i=1}^N e_i^2 = \sum_{i=1}^N (y_i - bx_i^a)^2 \quad (7)$$

This is the total error, that we want to minimize.

Other Least Squares Approximations

How can we minimize S ?

⇒ Compute dS/da and dS/db and set derivatives to zero.

Again, x_i, y_i are constants (data points) when differentiating.

$$\begin{aligned}\frac{\partial S}{\partial a} &= \sum_{i=1}^N 2(y_i - bx_i^a)(-b \ln x_i x_i^a) = 0, \\ \frac{\partial S}{\partial b} &= \sum_{i=1}^N 2(y_i - bx_i^a)(-x_i^a) = 0,\end{aligned}\tag{8}$$

Unfortunately, again no exact solution to this system can be found.

Other Least Squares Approximations

The method commonly used when data suspected to be exponentially related is to consider **logarithm of approximation equation**

Replace original equations with

- $y = be^{ax} \Rightarrow \ln y = \ln b + ax$
- $y = bx^a \Rightarrow \ln y = \ln b + a \ln x$

A linear problem now appears and solutions can be found as in the linear least squares approach.

However, approximation obtained this way is not the least squares approximation of the original problem.

Next week

Interpolation



THE UNIVERSITY OF

MELBOURNE



ENGR30003 Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

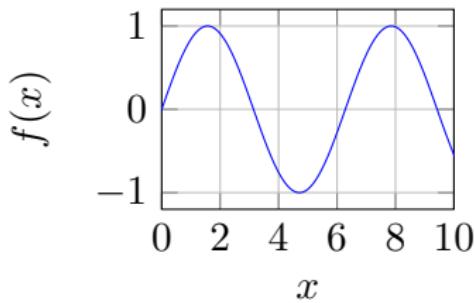
Copywrite © The University of Melbourne
Slides developed by Prof Richard Sandberg

This week

LECTURE 17/18

Interpolation

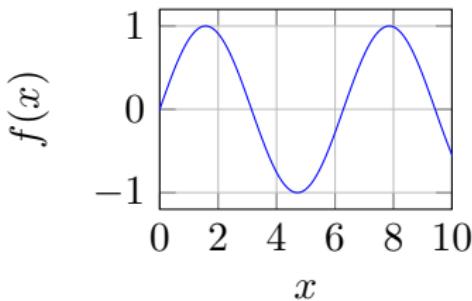
Interpolation - Motivation



Continuous Function

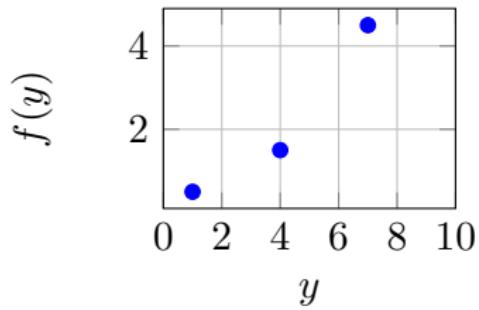
$$f : x \mapsto \sin(x)$$

Interpolation - Motivation



Continuous Function

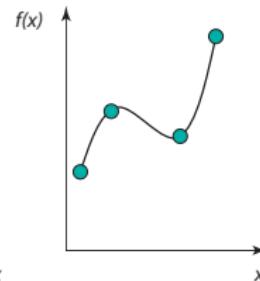
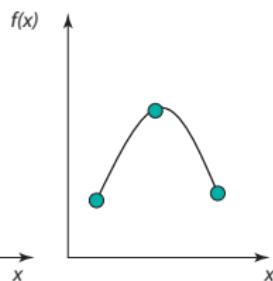
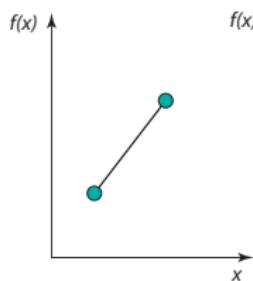
$$f : x \mapsto \sin(x)$$



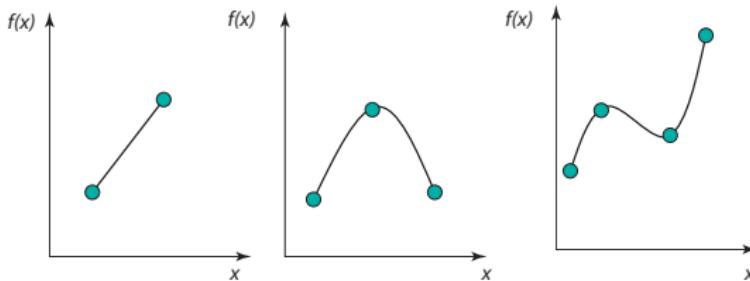
Function at discrete points

y	$f(y)$
1	.5
4	1.5
7	4.5

Polynomial Interpolation



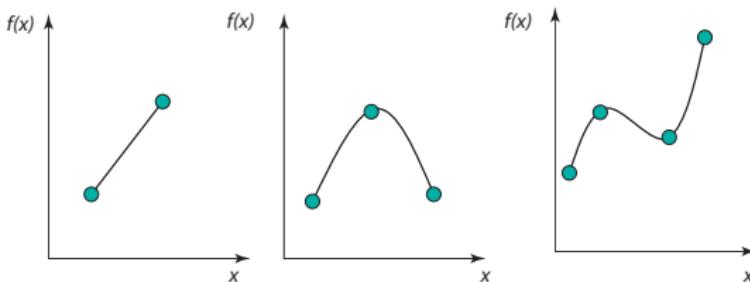
Polynomial Interpolation



We define a polynomial of the form

$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \quad (1)$$

Polynomial Interpolation

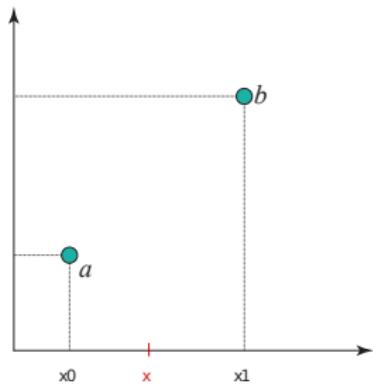


We define a polynomial of the form

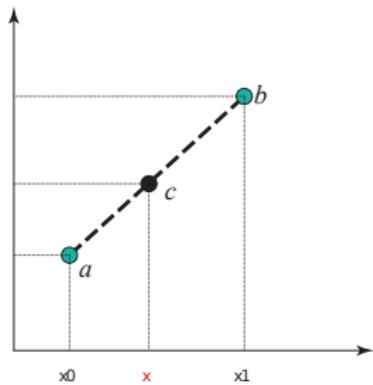
$$f(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n \quad (1)$$

- require polynomial to pass through known points
- compute coefficient a_i
- evaluate polynomial to obtain interpolated values

Newton Interpolation - Linear



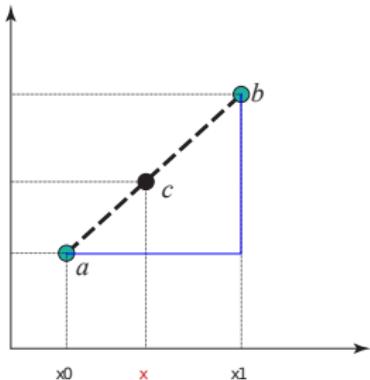
Newton Interpolation - Linear



Newton Interpolation - Linear

Using similar triangles:

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1(x) - f(x_0)}{x - x_0} \quad (2)$$



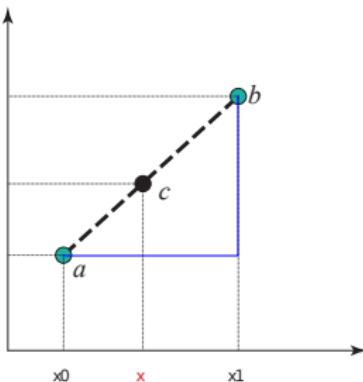
Newton Interpolation - Linear

Using similar triangles:

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1(x) - f(x_0)}{x - x_0} \quad (2)$$

Rearrange Eq. (2) to give

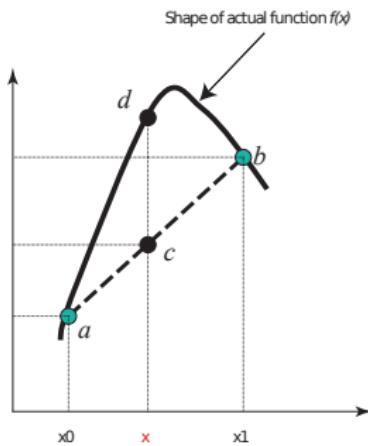
$$f_1(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0} (x - x_0) \quad (3)$$



Newton Interpolation - Linear

Using similar triangles:

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f_1(x) - f(x_0)}{x - x_0} \quad (2)$$



Rearrange Eq. (2) to give

$$f_1(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0} (x - x_0) \quad (3)$$

Linear Interpolation Properties

$$f_1(x) = f(x_0) \left(\frac{x_1 - x}{x_1 - x_0} \right) + f(x_1) \left(\frac{x - x_0}{x_1 - x_0} \right) \quad (4)$$

Linear Interpolation Properties

$$f_1(x) = f(x_0) \left(\frac{x_1 - x}{x_1 - x_0} \right) + f(x_1) \left(\frac{x - x_0}{x_1 - x_0} \right) \quad (4)$$

A few things worth noting about the linear interpolation

- Equation (4) is called the linear-interpolation formula
- In general, the smaller the interval between x_0 and x_1 , the better the approximation at point x .
- The subscript 1 for $f_1(x)$ denotes a first order interpolating polynomial: $f_1(x) = a_0 + a_1x$.

Linear Interpolation – Example 1

Use linear interpolation to estimate the value of the function

$$f(x) = 1 - e^{-x}$$

at $x = 1.0$. Use the interval $x_0 = 0$ and $x_1 = 5.0$. Repeat the calculation with $x_1 = 4.0$, $x_1 = 3.0$ and $x_1 = 2.0$. Illustrate on a graph how you are approaching the exact value of $f(1) = 0.6321$.

Linear Interpolation – Example 1

Use

$$f_1(x) = f(x_0) \left(\frac{x_1 - x}{x_1 - x_0} \right) + f(x_1) \left(\frac{x - x_0}{x_1 - x_0} \right)$$

$$f(x_0 = 0) = 1 - 1 = 0$$

$$f(x_1 = 5) = 1 - e^{-5} = 0.993269053$$

$$f(x_1 = 4) = 1 - e^{-4} = 0.981684361$$

$$f(x_1 = 3) = 1 - e^{-3} = 0.950212932$$

Linear Interpolation – Example 1

Use

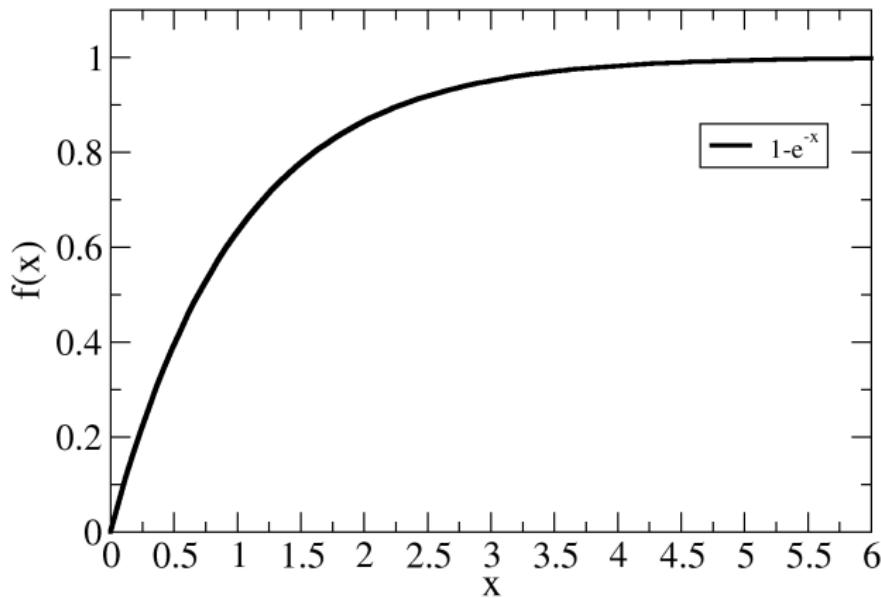
$$f_1(x) = f(x_0) \left(\frac{x_1 - x}{x_1 - x_0} \right) + f(x_1) \left(\frac{x - x_0}{x_1 - x_0} \right)$$

$$x_1 = 5 \quad \rightarrow \quad f_1(1) = 0.993269053 \left(\frac{1 - 0}{5 - 0} \right) = 0.19966538$$

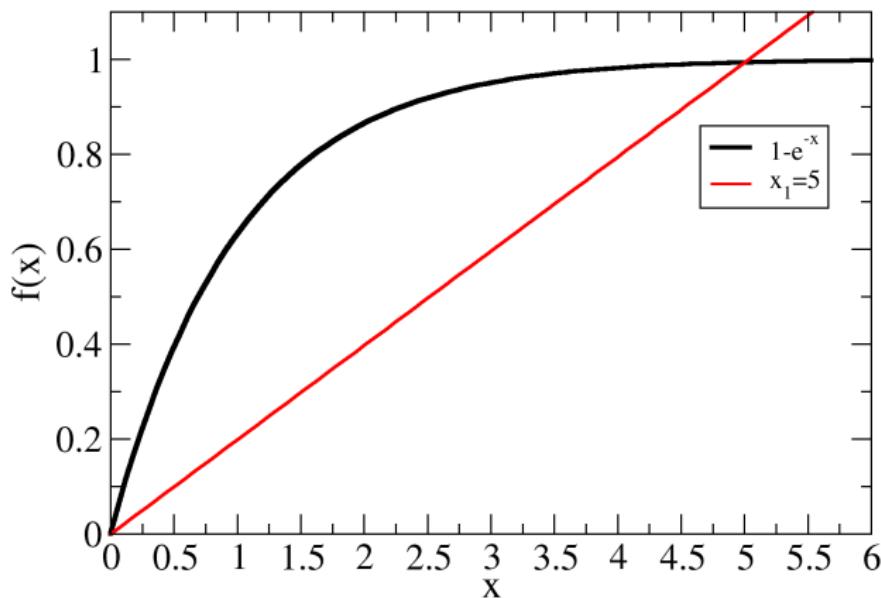
$$x_1 = 4 \quad \rightarrow \quad f_1(1) = 0.981684361 \left(\frac{1 - 0}{4 - 0} \right) = 0.24542109$$

$$x_1 = 3 \quad \rightarrow \quad f_1(1) = 0.950212932 \left(\frac{1 - 0}{3 - 0} \right) = 0.31673764$$

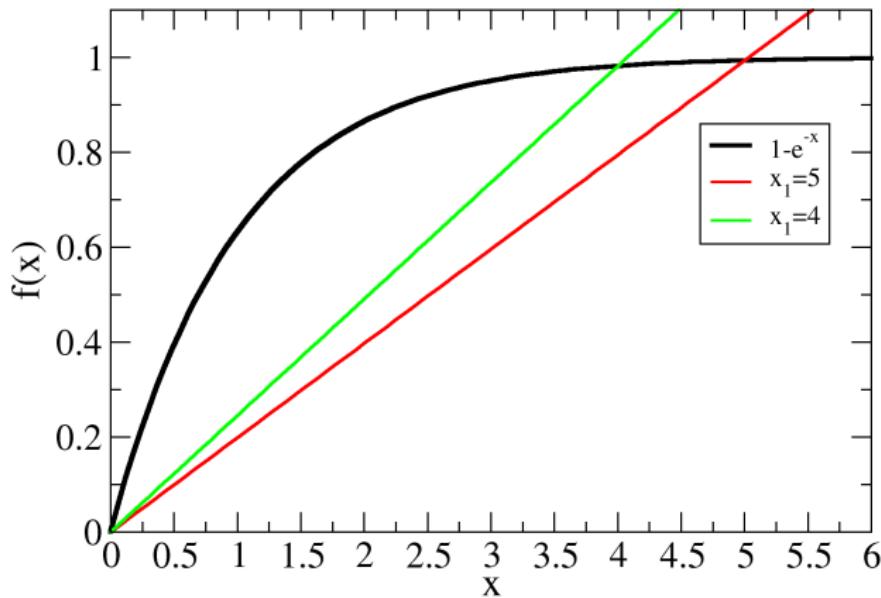
Linear Interpolation – Example 1



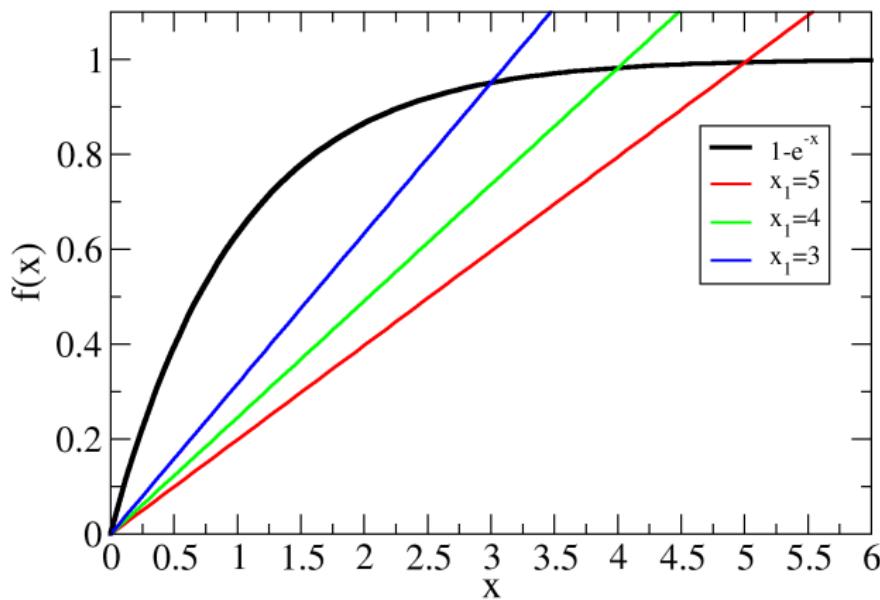
Linear Interpolation – Example 1



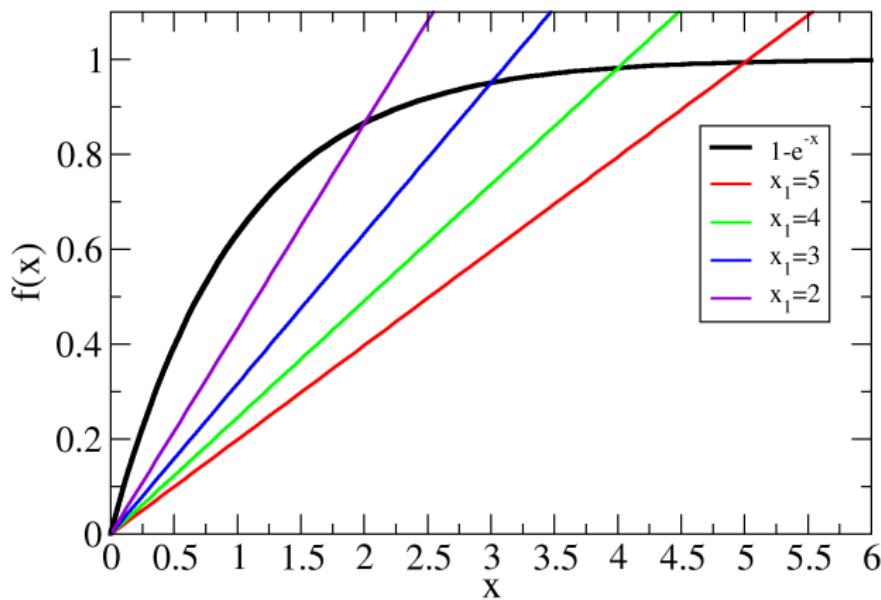
Linear Interpolation – Example 1



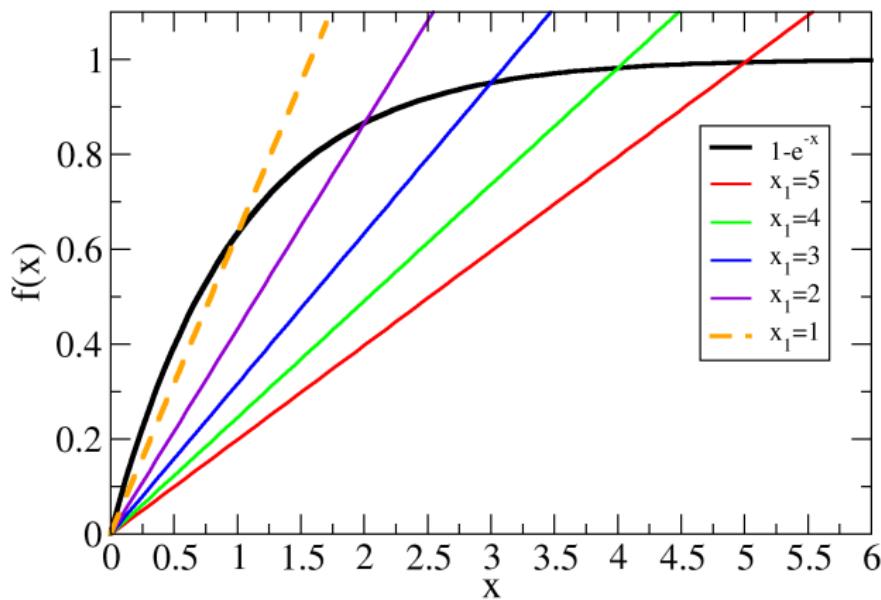
Linear Interpolation – Example 1



Linear Interpolation – Example 1

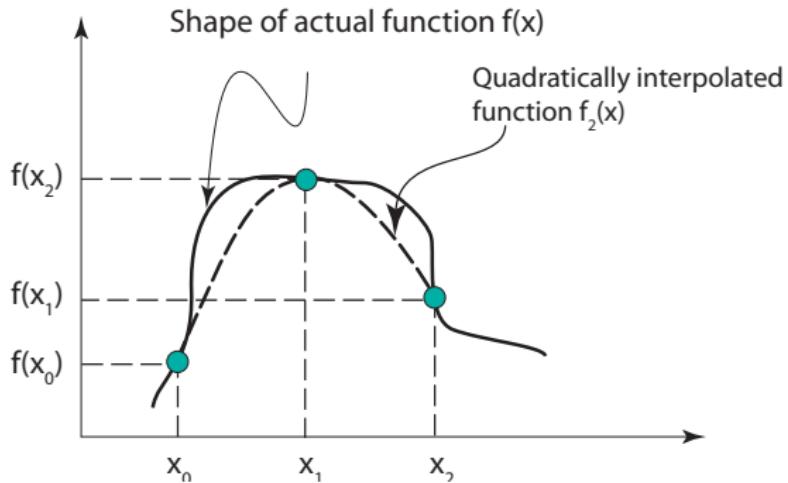


Linear Interpolation – Example 1



Newton Interpolation - Quadratic Polynomial I

$$f_2(x) = a_0 + a_1x + a_2x^2 \quad (5)$$



Newton Interpolation - Quadratic Polynomial II

The interpolated function is required to have the same values as the actual function at the data points, hence

- $f_2(x_0) = f(x_0)$
- $f_2(x_1) = f(x_1)$
- $f_2(x_2) = f(x_2)$

Note that the subscript 2 in $f_2(x)$ denotes a second order interpolating polynomial.

Eq. (5) is just another form of the polynomial function defined in Eq. (1). Alternatively, it can be written as

$$f_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) \quad (6)$$

where

$$a_0 = b_0 - b_1x_0 + b_2x_0x_1$$

$$a_1 = b_1 - b_2x_0 - b_2x_1$$

$$a_2 = b_2$$

Newton Interpolation - Quadratic Polynomial III

Hence, in order to do quadratic interpolation, we need to find all the b 's in Eq. (6).

- (1) Set $x = x_0$ in Eq. (6). Therefore

$$b_0 = f(x_0) \quad (7)$$

- (2) If we now let $x = x_1$ in Eq. (6) and use the result from step (1) above, we get

$$b_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \quad (8)$$

- (3) Since $f_2(x_2) = f(x_2)$, we can use Eqs. (7) and (8) together with Eq. (6) to obtain

$$b_2 = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0} \quad (9)$$

Newton Interpolation - Quadratic Polynomial IV

Equations (8) and (9) can be simplified by introducing the following notation

$$b_1 = f[x_1, x_0] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} \quad (10)$$

$$b_2 = f[x_2, x_1, x_0] = \frac{f[x_2, x_1] - f[x_1, x_0]}{x_2 - x_0} \quad (11)$$

Quadratic Interpolation – Example 2

Use quadratic interpolation to estimate the value of the function

$$f(x) = 1 - e^{-x}$$

at $x = 1.0$. Use the intervals $x_0 = 0$, $x_1 = 2.0$ and $x_2 = 6.0$. Repeat the calculation with $x_0 = 0$, $x_1 = 2.0$, $x_2 = 4.0$, then $x_0 = 0$, $x_1 = 2.0$, $x_2 = 3.0$, and finally $x_0 = 0$, $x_1 = 0.5$ and $x_2 = 2.0$. Illustrate on a graph how you are approaching the exact value of $f(1) = 0.6321$.

Quadratic Interpolation – Example 2

Use

$$f_2(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1)$$

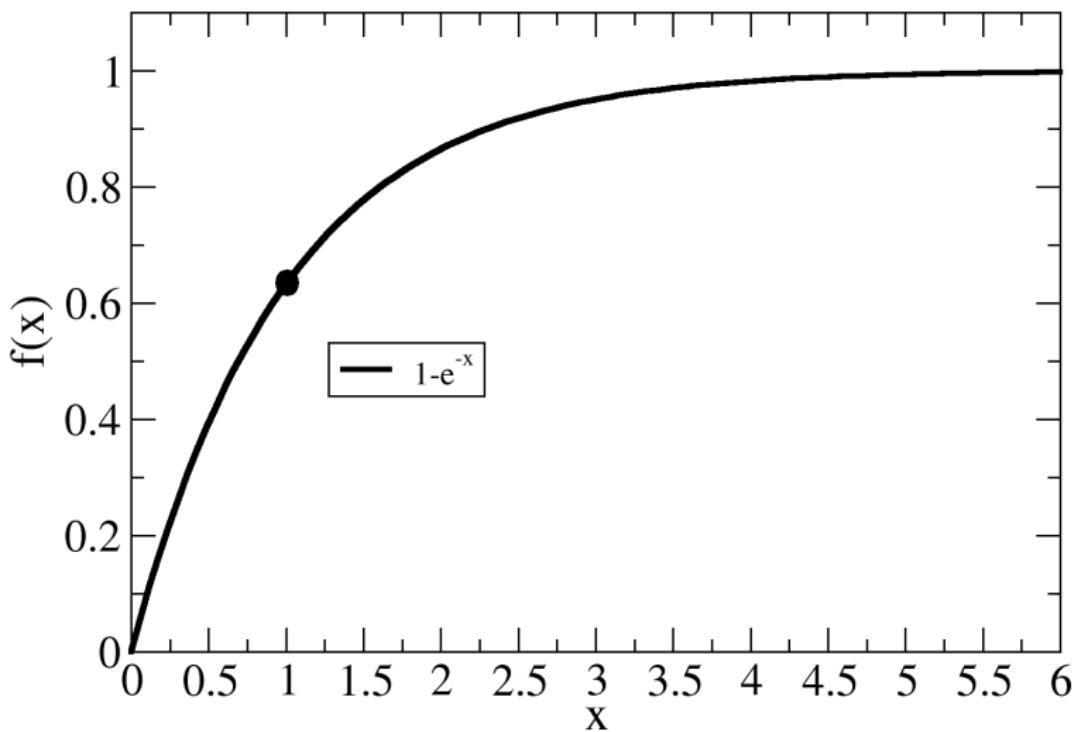
$$b_0 = f(x_0 = 0) = 0$$

$$b_1 = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f(x_1) - b_0}{x_1 - x_0}$$

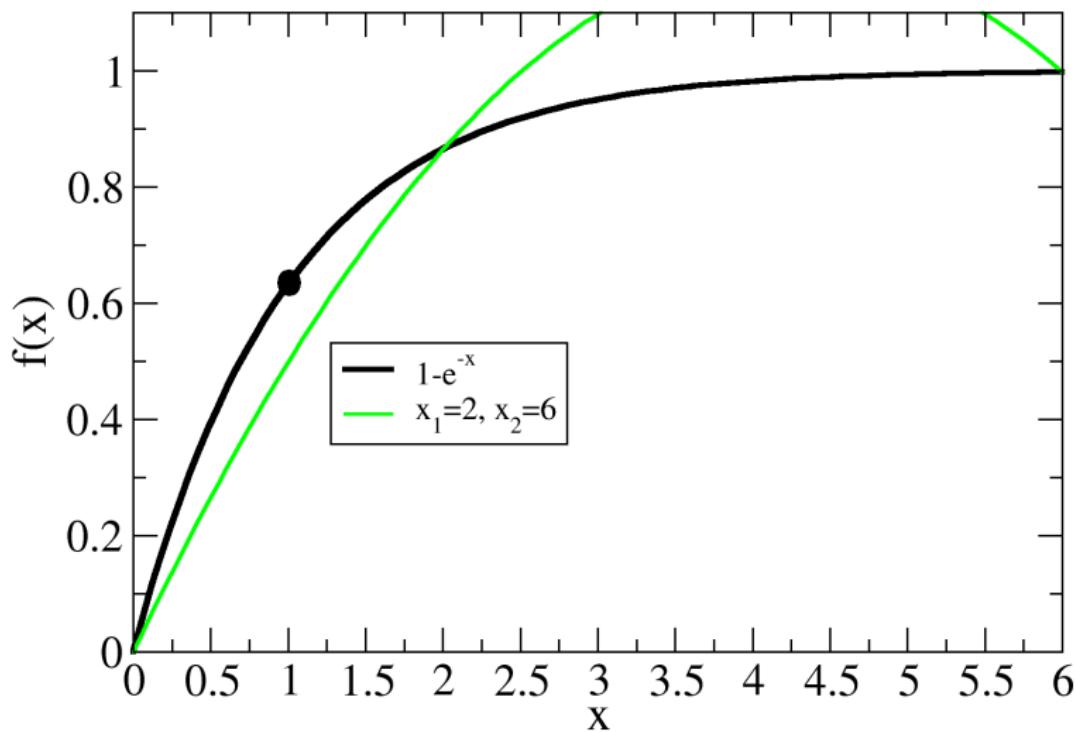
$$b_2 = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_0}$$

$$b_2 = \frac{\frac{f(x_2) - f(x_1)}{x_2 - x_1} - b_1}{x_2 - x_0}$$

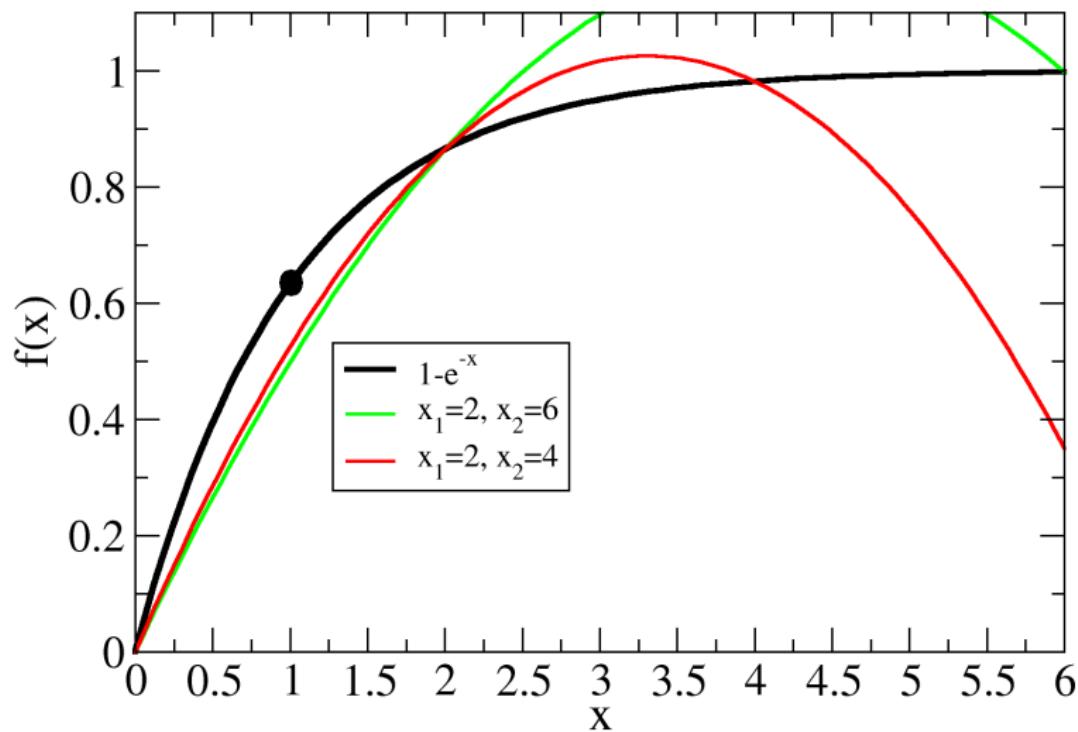
Quadratic Interpolation – Example 2



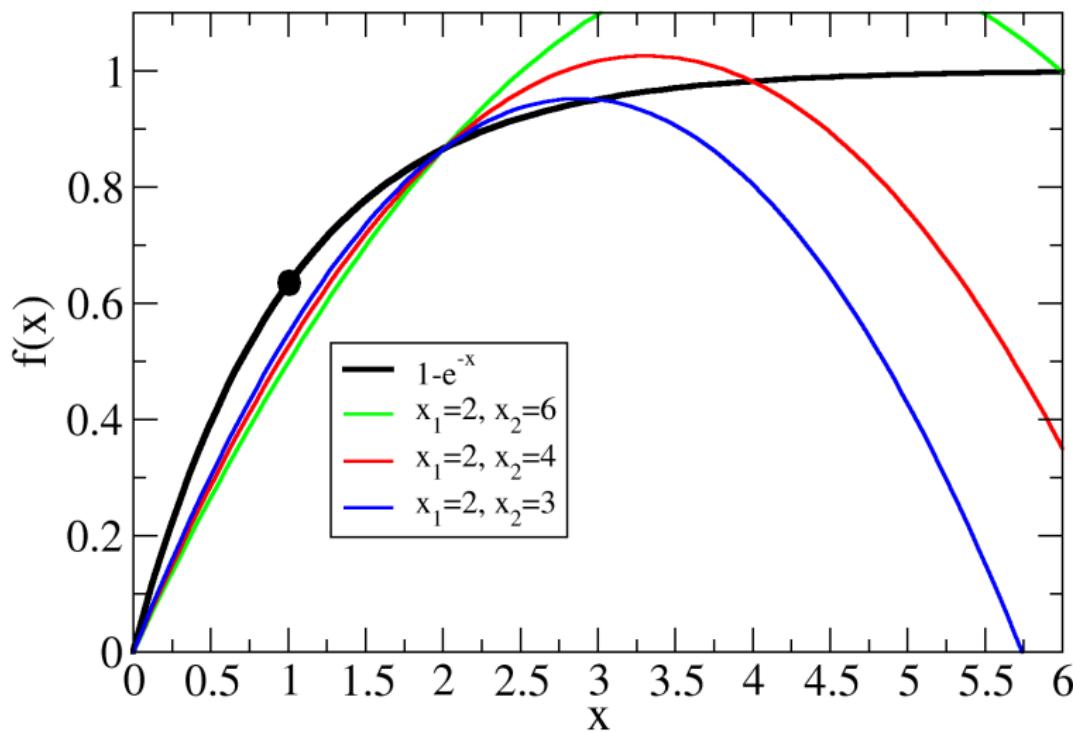
Quadratic Interpolation – Example 2



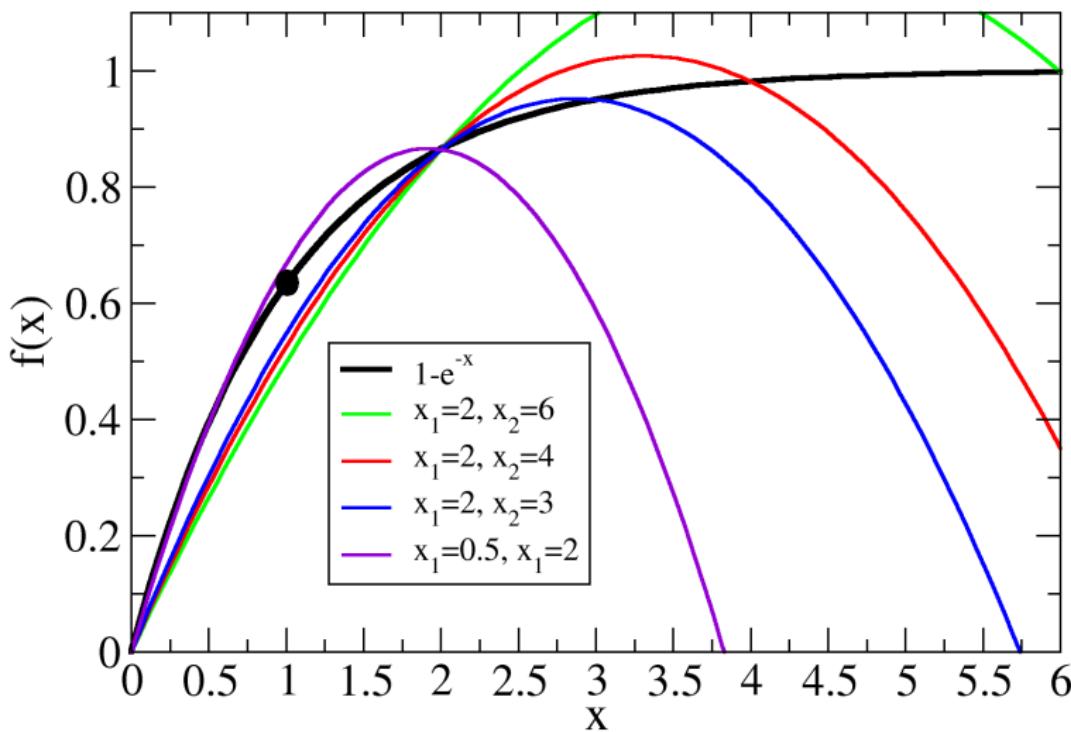
Quadratic Interpolation – Example 2



Quadratic Interpolation – Example 2



Quadratic Interpolation – Example 2



Newton Interpolation - General I

In general, if you have n data points, you will fit a polynomial of order $n - 1$ through all the n data points.

$$f_n(x) = b_0 + b_1(x - x_0) + b_2(x - x_0)(x - x_1) + \cdots + \\ + b_n(x - x_0)(x - x_1) \cdots (x - x_{n-1})$$

where all the b 's are defined as

$$b_0 = f(x_0)$$

$$b_1 = f[x_1, x_0]$$

$$b_2 = f[x_2, x_1, x_0]$$

$$\vdots \quad \vdots \quad \vdots$$

$$b_n = f[x_n, x_{n-1}, \dots, x_2, x_1, x_0]$$

Newton Interpolation - General II

$$f[x_i, x_j] = \frac{f(x_i) - f(x_j)}{x_i - x_j}$$

is called the first divided difference,

$$f[x_i, x_j, x_k] = \frac{f[x_i, x_j] - f[x_j, x_k]}{x_i - x_k}$$

is called the second divided difference

$$f[x_i, x_j, x_k, x_l] = \frac{f[x_i, x_j, x_k] - f[x_j, x_k, x_l]}{x_i - x_l}$$

is called the third divided difference and

$$f[x_n, \dots, x_0] = \frac{f[x_n, x_{n-1}, \dots, x_1] - f[x_{n-1}, x_{n-2}, \dots, x_0]}{x_n - x_0}$$

is called the n 'th divided difference.

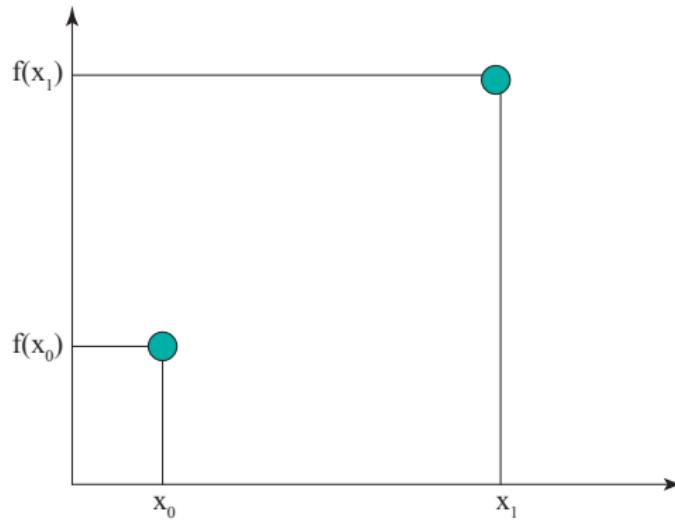
Newton Interpolation - General III

Note that the second divided difference is calculated from the first divided difference. The third divided difference is calculated from the second divided difference and so on.

i	x_i	$f(x_i)$	First	Second	Third
0	x_0	$f(x_0)$	$f[x_1, x_0]$	$f[x_2, x_1, x_0]$	$f[x_3, x_2, x_1, x_0]$
1	x_1	$f(x_1)$	$f[x_2, x_1]$	$f[x_3, x_2, x_1]$	
2	x_2	$f(x_2)$	$f[x_3, x_2]$		
3	x_3	$f(x_3)$			

Lagrange Interpolating Polynomials I

Suppose we want to interpolate between two data points,
 $(x_0, f(x_0)), (x_1, f(x_1))$



Lagrange Interpolating Polynomials II

Assume that we have two functions, $L_0(x)$ and $L_1(x)$. They are defined such that

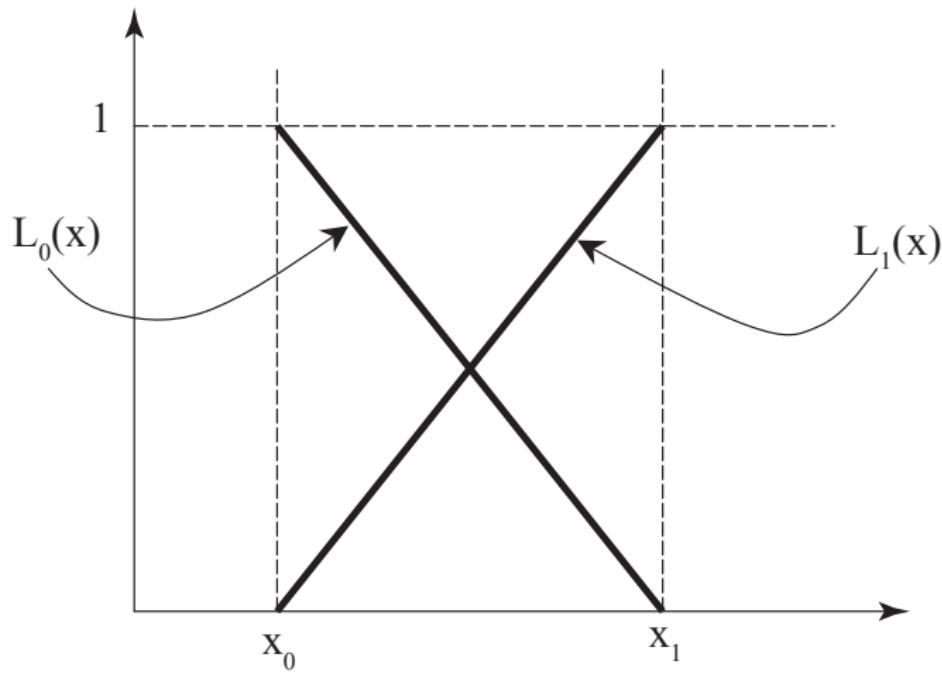
$$L_0(x) = \begin{cases} 1 & \text{if } x = x_0 \\ 0 & \text{if } x = x_1 \end{cases}$$

$$L_1(x) = \begin{cases} 0 & \text{if } x = x_0 \\ 1 & \text{if } x = x_1 \end{cases}$$

$$L_0(x) = \frac{(x - x_1)}{(x_0 - x_1)}, \quad L_1(x) = \frac{(x - x_0)}{(x_1 - x_0)}$$

$$L_0(x) = \frac{(x - x_1)}{(x_0 - x_1)}, \quad L_1(x) = \frac{(x - x_0)}{(x_1 - x_0)}$$

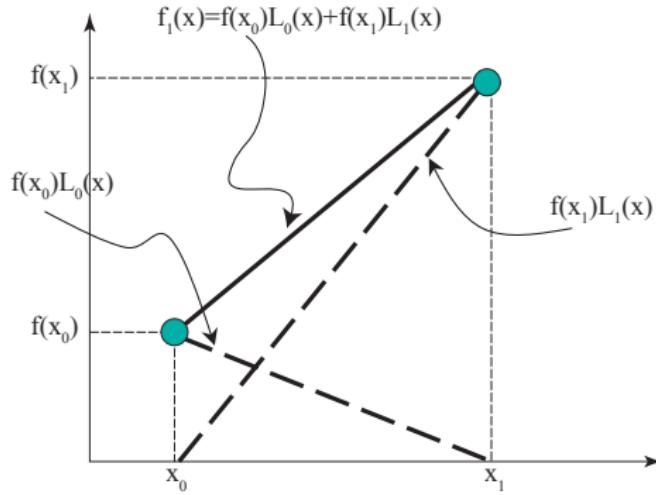
Lagrange Interpolating Polynomials III



Lagrange Interpolating Polynomials IV

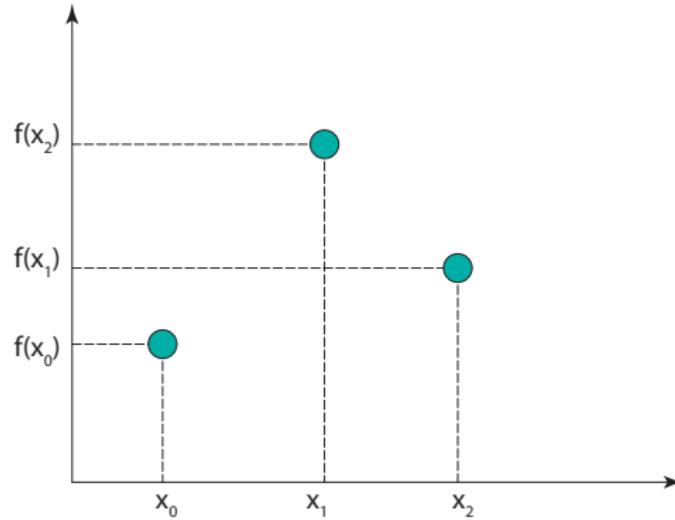
The first order Lagrange interpolating polynomial is obtained by a linear combination of $L_0(x)$ and $L_1(x)$.

$$f_1(x) = f(x_0)L_0(x) + f(x_1)L_1(x) \quad (12)$$



Lagrange Interpolating Polynomials V

Suppose we have three data points,
 $(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2))$ and we want to fit a
polynomial



Lagrange Interpolating Polynomials VI

Assume that we have three functions, $L_0(x)$, $L_1(x)$ and $L_2(x)$ defined such that

$$L_0(x) = \begin{cases} 1 & \text{if } x = x_0 \\ 0 & \text{if } x = x_1 \\ 0 & \text{if } x = x_2 \end{cases} \quad (13)$$

$$L_1(x) = \begin{cases} 0 & \text{if } x = x_0 \\ 1 & \text{if } x = x_1 \\ 0 & \text{if } x = x_2 \end{cases} \quad (14)$$

$$L_2(x) = \begin{cases} 0 & \text{if } x = x_0 \\ 0 & \text{if } x = x_1 \\ 1 & \text{if } x = x_2 \end{cases} \quad (15)$$

Lagrange Interpolating Polynomials VII

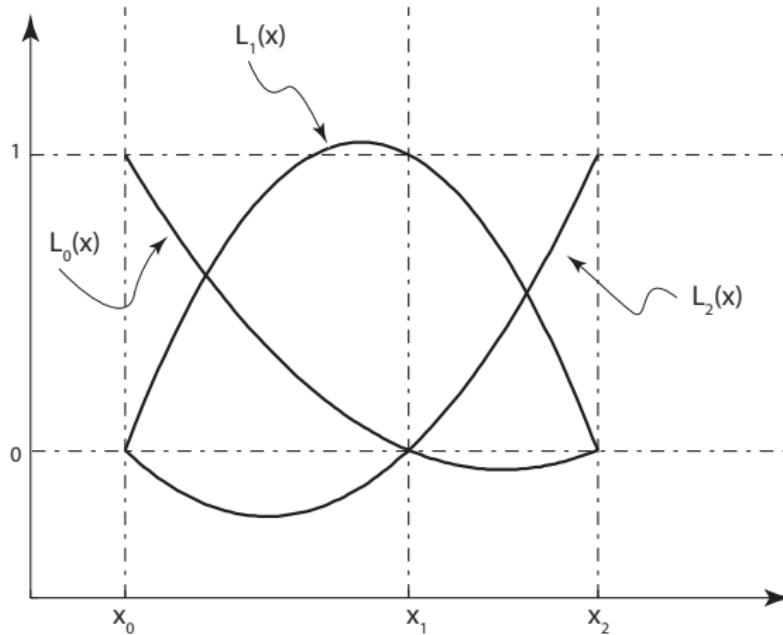
It should be easy to convince yourself that the following expressions for $L_0(x)$, $L_1(x)$ and $L_2(x)$ satisfy the conditions listed in Eqs. (13) - (15).

$$L_0(x) = \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} \quad (16)$$

$$L_1(x) = \frac{(x - x_2)(x - x_0)}{(x_1 - x_2)(x_1 - x_0)} \quad (17)$$

$$L_2(x) = \frac{(x - x_1)(x - x_0)}{(x_2 - x_1)(x_2 - x_0)} \quad (18)$$

Lagrange Interpolating Polynomials VIII



Lagrange Interpolating Polynomials IX

The second order Lagrange interpolating polynomial is obtained by a linear combination of $L_0(x)$, $L_1(x)$ and $L_2(x)$.

$$f_2(x) = f(x_0)L_0(x) + f(x_1)L_1(x) + f(x_2)L_2(x) \quad (19)$$

In general, the n th order Lagrange polynomial (i.e. the Lagrange polynomial that can be fit through $n + 1$ data points) can be represented concisely as

$$f_n(x) = \sum_{i=0}^n L_i(x)f(x_i) \quad (20)$$

where

$$L_i(x) = \prod_{j=0, j \neq i}^n \frac{x - x_j}{x_i - x_j} \quad (21)$$

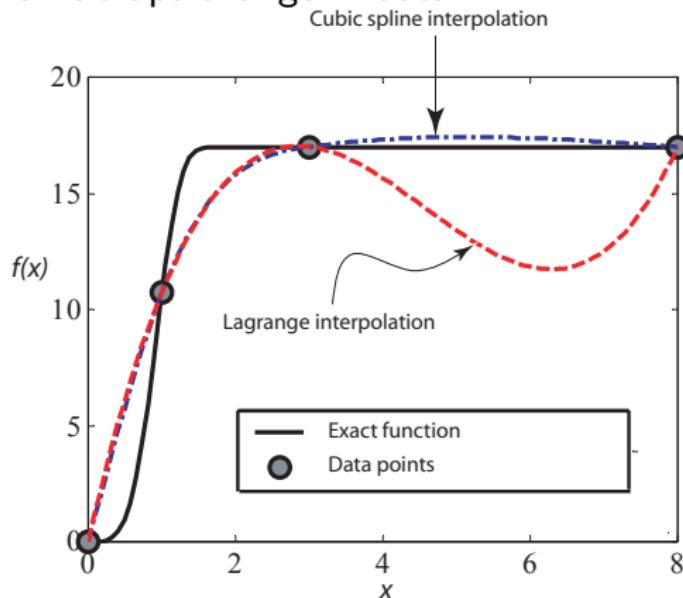
Lagrange Interpolating Polynomials X

Note that:

- The Lagrange interpolating polynomials are just a different form of the Newton interpolating polynomials.
- The main advantage is that they are slightly easier to program.
- They are slightly slower to compute than the Newton Interpolating polynomials.

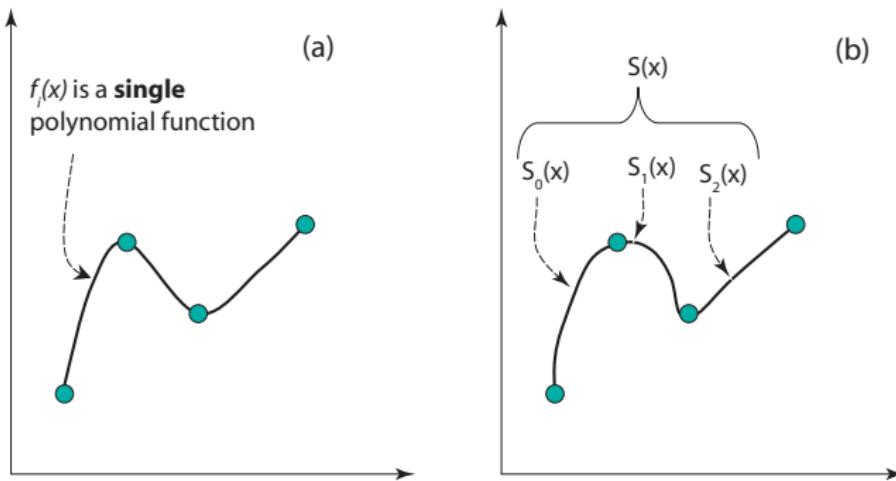
Spline Interpolation I

Newton or Lagrange interpolation can lead to erroneous results when there is an abrupt change in data



Spline Interpolation II

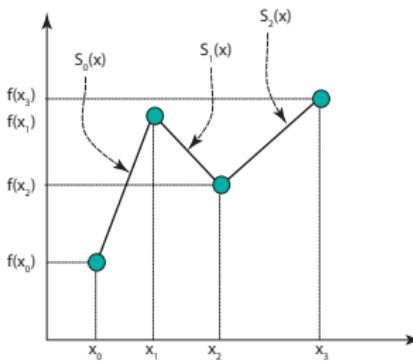
Splines are made up of piecewise polynomials connecting only two data points. This is different to Newton or Lagrange polynomials where you have only one polynomial connecting all data points.



Spline Interpolation III

- The spline function, $S(x)$, is made up of several polynomial functions, $S_i(x)$.
- $S(x) = S_i(x)$ for $x_i < x < x_{i+1}$.
- $S_i(x)$ is ONLY defined between x_{i+1} and x_i . Outside of the interval, $S_i(x)$ is not defined and has got NO meaning.
- If $S_i(x)$ are linear functions, then $S(x)$ is called a linear spline.
 $S_i(x)$ quadratic $\rightarrow S(x)$ quadratic spline.
 $S_i(x)$ cubic $\rightarrow S(x)$ cubic spline

Linear Spline Interpolation I



- $S_i(x) = a_i + b_i(x - x_i)$
- $S(x) = S_i(x)$ for $x_i < x < x_{i+1}$.
- Note that $S(x)$ is continuous but the derivative of $S(x)$, $S'(x)$, is not continuous at $x = x_i$.
- If there are $n + 1$ data points, there are only n intervals and hence, n number of defined $S_i(x)$.

Linear Spline Interpolation II

Need to find all the a_i 's and b_i 's in order to find $S(x)$. For this case, there are $2n$ number of unknowns. In order to find all the unknowns, we need to impose the following requirements

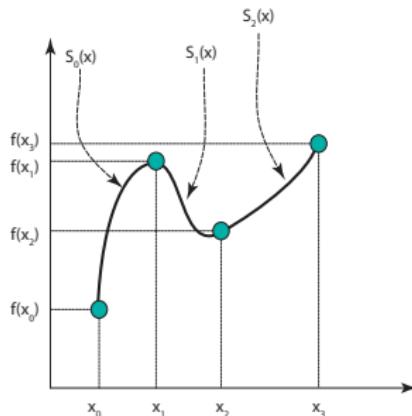
1. Require $S(x)$ to have the values of $f(x_i)$ at $x = x_i$, hence,

$$S_i(x_i) = a_i = f(x_i) \quad (22)$$

2. Require that the function values at adjacent polynomials must be equal at the interior nodes

$$\begin{aligned} S_i(x_{i+1}) &= S_{i+1}(x_{i+1}) \\ a_i + b_i(x_{i+1} - x_i) &= a_{i+1} \\ b_i &= \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \end{aligned} \quad (23)$$

Quadratic Spline Interpolation I



- $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2$
- $S(x) = S_i(x)$ for $x_i < x < x_{i+1}$.
- $S(x)$ is continuous and the derivative of $S(x)$, $S'(x)$, is also continuous at $x = x_i$.
- For $n + 1$ data points, there are n number of defined $S_i(x)$.

Quadratic Spline Interpolation II

Need to find all a_i 's, b_i 's and c_i 's to completely define $S(x)$, hence we have to impose $3n$ conditions. To do this we will require that

1. $S(x)$ to have the values of $f(x_i)$ at $x = x_i$, hence,

$$S_i(x_i) = a_i = f(x_i) \quad (24)$$

2. Function values at adjacent polynomials must be equal at interior nodes

$$S_i(x_{i+1}) = S_{i+1}(x_{i+1}) \quad (25)$$

$$a_i + b_i(x_{i+1} - x_i) + c_i(x_{i+1} - x_i)^2 = a_{i+1} \quad (26)$$

For the sake of conciseness, let $h_i = x_{i+1} - x_i$, so

$$a_i + b_i h_i + c_i h_i^2 = a_{i+1} \quad (27)$$

Quadratic Spline Interpolation III

- Derivative of $S_i(x)$, $S'_i(x)$, to be continuous at the interior nodes

$$S'_i(x_{i+1}) = S'_{i+1}(x_{i+1}) \quad (28)$$

Equation (28) leads to

$$b_i + 2c_i h_i = b_{i+1} \quad (29)$$

From Eq. (27) we can get

$$b_i = \frac{a_{i+1} - a_i}{h_i} - c_i h_i \quad (30)$$

Substitute Eq. (30) into (29) and rearrange to get

$$c_i = \frac{1}{h_i} \left(\frac{a_{i+1}}{h_i} - a_i \left[\frac{1}{h_{i-1}} + \frac{1}{h_i} \right] + \frac{a_{i-1}}{h_{i-1}} - c_{i-1} h_{i-1} \right) \quad (31)$$

Quadratic Spline Interpolation IV

Hence, to construct quadratic splines, do the following

1. Make use of Eq. (24) and set $a_i = f(x_i)$.
2. Assume $c_0 = 0$. This effectively makes the first spline segment linear!
3. Use Eq. (31) to obtain all other values of c_i 's.
4. Use Eq. (30) to obtain the b_i 's.

After steps 1-4, we can evaluate the function values at the point you are interested from the following formula

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2. \quad (32)$$

The only thing left is to figure out which interval your x value belongs to.

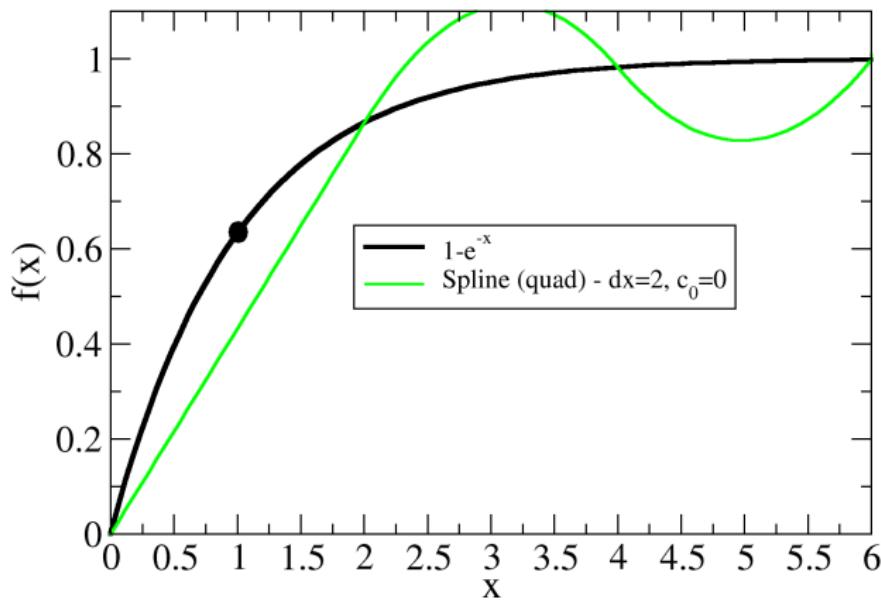
Piecewise quadratic Interpolation – Example 3

Use quadratic splines to estimate the value of the function

$$f(x) = 1 - e^{-x}$$

at $x = 1.0$. Use the intervals $x_0 = 0$, $x_1 = 2.0$, $x_2 = 4.0$ and $x_3 = 6.0$. Plot the spline for the interval $0 \leq x \leq 6$.

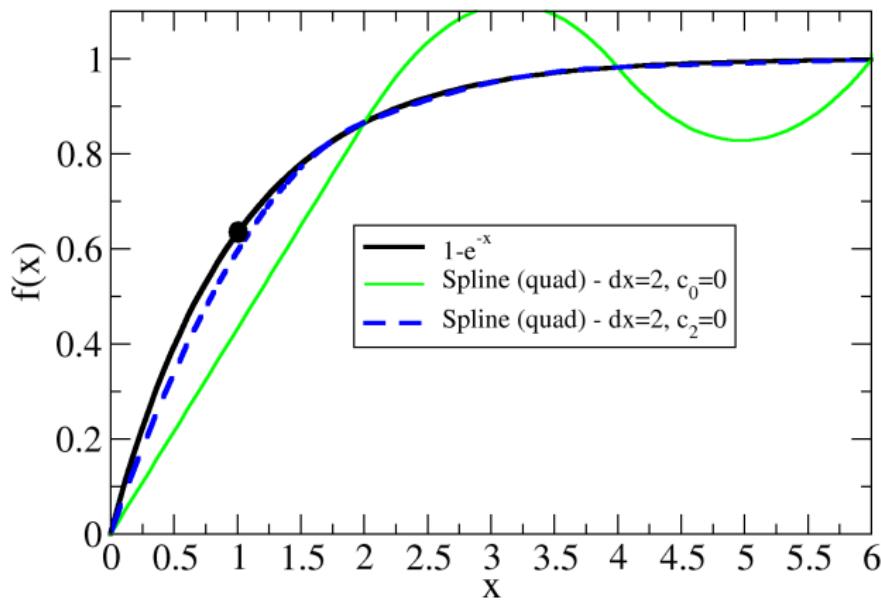
Piecewise quadratic Interpolation – Example 3



Notice, spline solution linear for first segment. For this function, this is probably not where we want it to be linear (and then quadratic for larger x).

Replace condition $c_0 = 0$ by $c_2 = 0$ and see what happens.

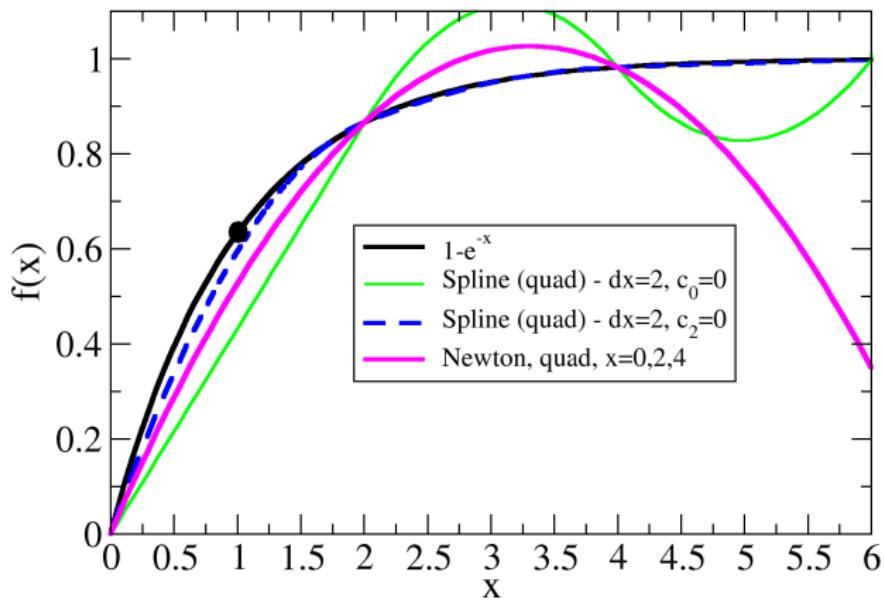
Piecewise quadratic Interpolation – Example 3



Notice, spline solution linear for first segment. For this function, this is probably not where we want it to be linear (and then quadratic for larger x).

Replace condition $c_0 = 0$ by $c_2 = 0$ and see what happens.

Piecewise quadratic Interpolation – Example 3

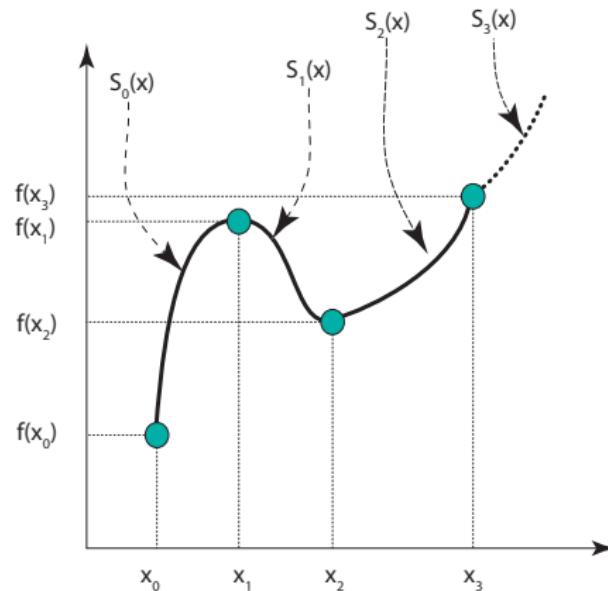


Notice, spline solution linear for first segment. For this function, this is probably not where we want it to be linear (and then quadratic for larger x).

Replace condition $c_0 = 0$ by $c_2 = 0$ and see what happens.

Spline Interpolation - Alternative I

Fit a quadratic spline through four data points



Spline Interpolation - Alternative II

Use three 'real' quadratic polynomial $S_0(x)$, $S_1(x)$, $S_2(x)$ and one 'imaginary' polynomial $S_3(x)$ as

$$S_0(x) = a_0 + b_0(x - x_0) + c_0(x - x_0)^2 \quad (33)$$

$$S_1(x) = a_1 + b_1(x - x_1) + c_1(x - x_1)^2 \quad (34)$$

$$S_2(x) = a_2 + b_2(x - x_2) + c_2(x - x_2)^2 \quad (35)$$

$$S_3(x) = a_3 + b_3(x - x_3) + c_3(x - x_3)^2 \quad (36)$$

In the first instance, we know that

$$a_i = f(x_i) \quad \text{for } i = 0, 1, 2, 3. \quad (37)$$

Spline Interpolation - Alternative III

By requiring that adjacent $S_i(x)$ have common values at the interior points, we find that

$$b_0 = \frac{a_1 - a_0}{h_0} - c_0 h_0 \quad (38)$$

$$b_1 = \frac{a_2 - a_1}{h_1} - c_1 h_1 \quad (39)$$

$$b_2 = \frac{a_3 - a_2}{h_2} - c_2 h_2 , \quad (40)$$

where $h_i = x_{i+1} - x_i$.

By requiring that the derivatives of the adjacent $S_i(x)$ have common values at the interior points, we obtain

$$b_0 + 2c_0 h_0 = b_1 \quad (41)$$

Spline Interpolation - Alternative IV

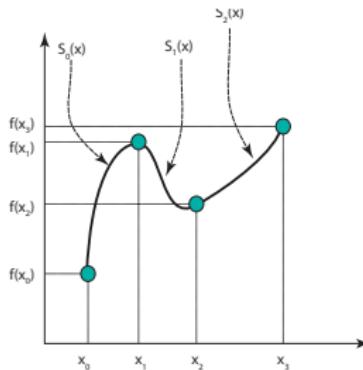
$$b_1 + 2c_1 h_1 = b_2 \quad (42)$$

Using Eqs. (38) to (42) together with the assumption that $c_0 = 0$, we get c_i 's by solving the following matrix equation

$$\begin{bmatrix} 1 & 0 & 0 \\ h_0 & h_1 & 0 \\ 0 & h_1 & h_2 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{a_2}{h_1} - a_1 \left(\frac{1}{h_0} + \frac{1}{h_1} \right) + \frac{a_0}{h_0} \\ \frac{a_3}{h_2} - a_2 \left(\frac{1}{h_1} + \frac{1}{h_2} \right) + \frac{a_1}{h_1} \end{bmatrix} \quad (43)$$

Thus, with c_0 , c_1 and c_2 known (and $a_i = f(x_i)$), can use (38)-(40) to solve for b_0 , b_1 and b_2 .

Cubic Spline Interpolation I



Spline consists of a cubic polynomials:

- $S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$
- $S(x) = S_i(x)$ for $x_i < x < x_{i+1}$.
- $S(x)$, its first and second derivatives $S'(x)$, $S''(x)$ are all continuous at $x = x_i$.
- For $n + 1$ data points, there are n number of defined $S_i(x)$.

Cubic Spline Interpolation II

Find a_i, b_i, c_i and d_i for $i = 0 \dots n - 1$ ($4n$ unknowns)

1. $S(x)$ have the values of f at $x = x_i$, hence,

$$S_i(x_i) = a_i = f(x_i) \quad i = 0, \dots, n - 1 \quad (44)$$

Equation (44) will give n conditions. In addition we know function value at the end of the last segment which gives another condition and we write $a_n = f(x_n)$. This notation will be handy later even though we do not have a spline S_n .

2. The function values at adjacent polynomials must be equal at the interior nodes. Defining $h_i = x_{i+1} - x_i$:

$$\begin{aligned} S_i(x_{i+1}) &= S_{i+1}(x_{i+1}) \\ a_i + b_i h_i + c_i h_i^2 + d_i h_i^3 &= a_{i+1} \end{aligned} \quad (45)$$

for $i = 0 \dots n - 2$. This gives us another $n - 1$ conditions.

Cubic Spline Interpolation III

3. The derivative of $S_i(x)$ shall be continuous at interior nodes

$$S'_i(x_{i+1}) = S'_{i+1}(x_{i+1}) \quad (46)$$

which leads to

$$b_i + 2c_i h_i + 3d_i h_i^2 = b_{i+1} \quad (47)$$

for $i = 0 \dots n - 2$. Note that for most cubic splines, b_n does not exist. Equation (47) will give us another $n - 1$ equations.

4. The second derivative of $S_i(x)$ needs to be continuous at interior nodes.

$$S''_i(x_{i+1}) = S''_{i+1}(x_{i+1}) \quad (48)$$

which gives

$$c_i + 3d_i h_i = c_{i+1} \quad (49)$$

where $i = 0 \dots n - 2$. This will give us another $n - 1$ equations.

Cubic Spline Interpolation IV

The above conditions provide $4n - 2$ equations. we still need two additional equations to solve the $4n$ number of unknowns.

- **clamped spline:** if more information about the function we are trying to approximate (e.g. its derivative at the end points) is known, it is used to get two equations
- **natural spline:** in the absence of any information, it is common to set the second derivative of the cubic spline to zero at the two end points. Thus:

$$c_0 = 0 \tag{50}$$

$$c_n = 0 \tag{51}$$

Cubic Spline Interpolation V

We are now ready to solve a_i , b_i , c_i and d_i . Re-arranging (49):

$$d_i = \frac{c_{i+1} - c_i}{3h_i} \quad (52)$$

Put Eq. (52) into Eq. (47) to get

$$b_{i+1} = b_i + (c_i + c_{i+1}) h_i \quad (53)$$

Put Eq. (52) into Eq. (45) to get

$$b_i = \frac{1}{h_i} (a_{i+1} - a_i) - \frac{h_i}{3} (2c_i + c_{i+1}) \quad (54)$$

Cubic Spline Interpolation VI

Substituting Eq. (54) into Eq. (53) and rearranging leads to the following relationship between a_i 's and c_i 's

$$h_{j-1}c_{j-1} + 2(h_j + h_{j-1})c_j + h_jc_{j+1} = \frac{3}{h_j}(a_{j+1} - a_j) + \frac{3}{h_{j-1}}(a_{j-1} - a_j) \quad (55)$$

Equation (55) is a tridiagonal system and can be solved if we assume that $c_0 = c_n = 0$. The linear system of equation equation that we need to solve looks like the following

$$[A]\{X\} = \{C\} \quad (56)$$

where

Cubic Spline Interpolation VII

$$\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ h_0 & 2(h_0 + h_1) & h_1 & 0 & \cdots & 0 \\ 0 & h_1 & 2(h_1 + h_2) & h_2 & \ddots & 0 \\ 0 & 0 & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & h_{n-2} & 2(h_{n-2} + h_{n-1}) & h_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & 1 \end{bmatrix}}_A \underbrace{\begin{Bmatrix} c_0 \\ c_1 \\ \vdots \\ \vdots \\ c_n \end{Bmatrix}}_X = \underbrace{\begin{Bmatrix} 0 \\ \frac{3}{h_1}(a_2 - a_1) + \frac{3}{h_0}(a_0 - a_1) \\ \frac{3}{h_2}(a_3 - a_2) + \frac{3}{h_1}(a_1 - a_2) \\ \vdots \\ \frac{3}{h_{n-1}}(a_n - a_{n-1}) + \frac{3}{h_{n-2}}(a_{n-2} - a_{n-1}) \\ 0 \end{Bmatrix}}_C$$

Cubic Spline Interpolation VIII

In summary, to construct cubic splines, do the following

1. Make use of Eq. (44) and set $a_i = f(x_i)$.
2. Solve Eq. (55) to obtain all other values of c_i 's. Note that by solving Eq. (55) we are assuming that $c_0 = 0$ and $c_n = 0$.
This is equivalent to saying that the second derivative at x_0 and x_n are zero !
3. Use Eq. (54) to obtain the b_i 's.
4. Use Eq. (52) to obtain the d_i 's.

Subsequently, we can evaluate the function values at any point x from the formula

$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3$. But we have to identify first which set of coefficients we need to use. That is, we have to find the interval $[x_i, x_{i+1}]$ that encompasses point x .

Next week

Differentiation/Integration



THE UNIVERSITY OF

MELBOURNE



ENGR30003

Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne
Slides developed by Prof Richard Sandberg

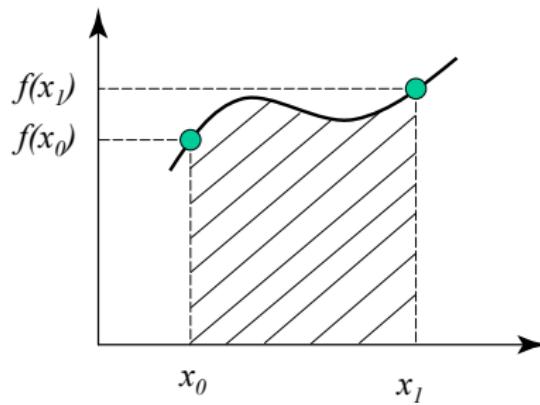
This week

LECTURE 19/20

Integration and Differentiation

Integration

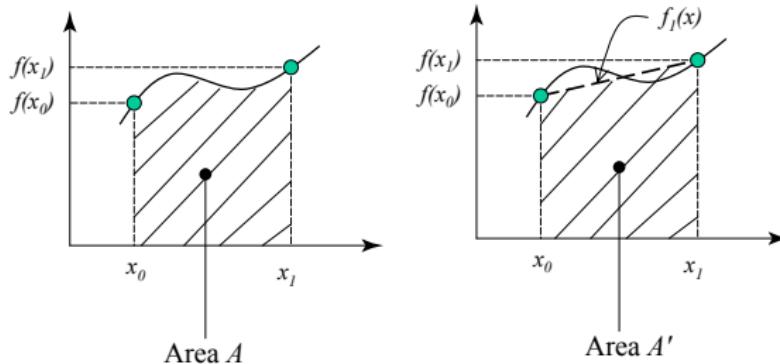
Integration means finding the area under a curve between two points, x_0 and x_1



If we only have information at these two points, the best we can do is fit straight line through these two points and find the area under it.

Integration - Trapezoidal Rule

We estimate A with A'



We know that $f(x)$ can be estimated by Lagrangian interpolation between the two points $f_1(x)$ as follows

$$f(x) = \underbrace{\frac{(x - x_1)}{(x_0 - x_1)} f(x_0) + \frac{(x - x_0)}{(x_1 - x_0)} f(x_1)}_{f_1(x)} + \underbrace{\frac{1}{2} f''(x) (x - x_0)(x - x_1)}_{\text{error}} \quad (1)$$

Integration

To obtain area A all we have to do is to integrate (1) from x_0 to x_1 .

$$\begin{aligned}\int_{x_0}^{x_1} f(x)dx &= \frac{f(x_0)}{(x_0 - x_1)} \int_{x_0}^{x_1} (x - x_1) dx \\ &\quad + \frac{f(x_1)}{(x_1 - x_0)} \int_{x_0}^{x_1} (x - x_0) dx \\ &\quad + \frac{1}{2} \int_{x_0}^{x_1} f''(x)(x - x_0)(x - x_1)dx ,\end{aligned}\quad (2)$$

Using the mean value theorem for integrals, the last term in (2) can be written as

$$\frac{1}{2} f''(\xi) \int_{x_0}^{x_1} (x - x_0)(x - x_1)dx = \frac{1}{12} f''(\xi) (x_0 - x_1)^3 \quad (3)$$

where ξ is some number in (x_0, x_1) .

Integration - Trapezoidal Rule

By integrating Eq. (2), using $\Delta = x_1 - x_0$, we get

$$\int_{x_0}^{x_1} f(x)dx = \frac{\Delta}{2} (f(x_0) + f(x_1)) - \frac{\Delta^3}{12} f''(\xi) , \quad (4)$$

Use Eq. (4) to derive **Trapezoidal rule**:

$$\int_{x_0}^{x_1} f(x)dx \approx \frac{\Delta}{2} (f(x_0) + f(x_1)) \quad (5)$$

Integration - Trapezoidal Rule

By integrating Eq. (2), using $\Delta = x_1 - x_0$, we get

$$\int_{x_0}^{x_1} f(x)dx = \frac{\Delta}{2} (f(x_0) + f(x_1)) - \frac{\Delta^3}{12} f''(\xi) , \quad (4)$$

Use Eq. (4) to derive **Trapezoidal rule**:

$$\int_{x_0}^{x_1} f(x)dx \approx \frac{\Delta}{2} (f(x_0) + f(x_1)) \quad (5)$$

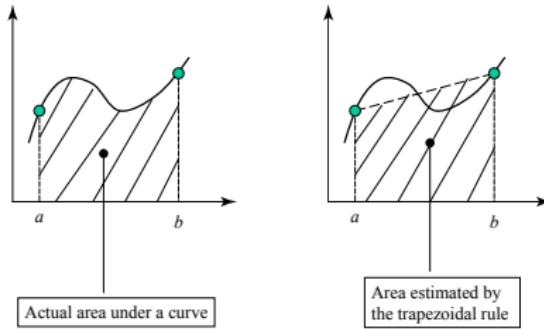
In general, at any two points, x_i and x_{i+1} , trapezoidal rule:

$$\boxed{\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{\Delta}{2} (f(x_i) + f(x_{i+1}))} , \quad (6)$$

with error of approximation

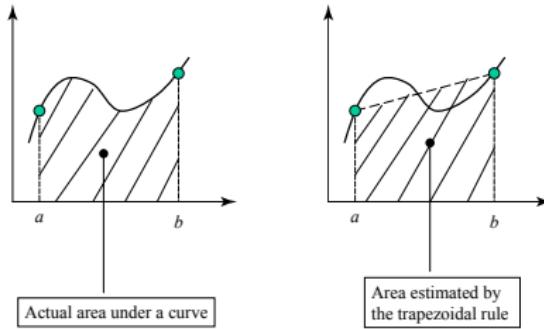
$$\frac{\Delta^3}{12} f''(\xi)$$

Integration - Trapezoidal Rule



If size of interval, Δ , is large, the error is large as well.

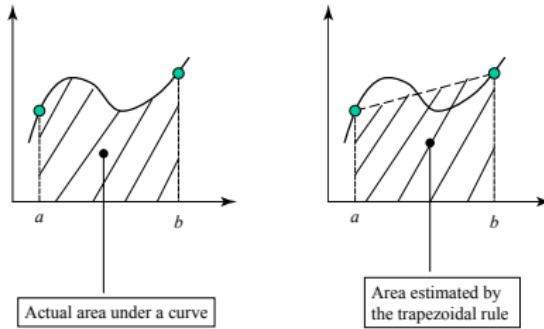
Integration - Trapezoidal Rule



If size of interval, Δ , is large, the error is large as well.

How can we minimize error as we integrate over a large interval?

Integration - Trapezoidal Rule



If size of interval, Δ , is large, the error is large as well.

How can we minimize error as we integrate over a large interval?

Divide domain into smaller intervals, usually of equal length, apply the Trapezoidal rule in each of the subintervals

Integration - Trapezoidal Rule

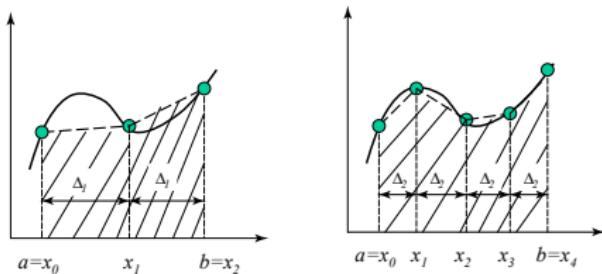


Figure: Decomposing large interval into smaller intervals.

Intuitively, you get more accurate answers as you divide the large interval up into smaller intervals.

Integration - Trapezoidal Rule

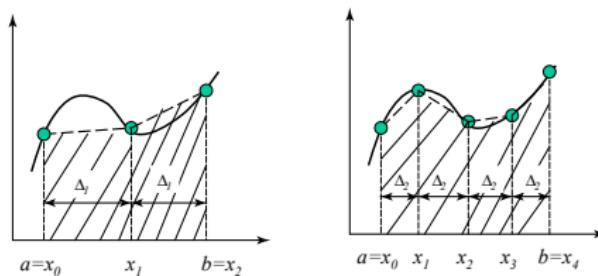


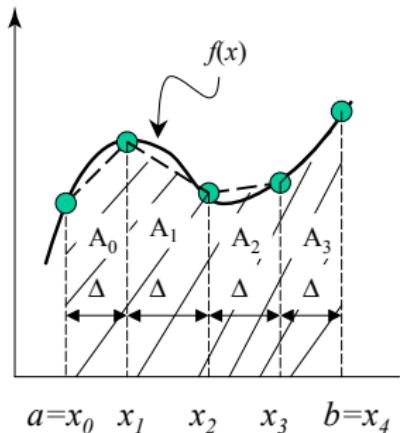
Figure: Decomposing large interval into smaller intervals.

Intuitively, you get more accurate answers as you divide the large interval up into smaller intervals.

What is general formula for Trapezoidal rule if we apply it to smaller subintervals ?

Integration - Trapezoidal Rule

Apply Trapezoidal rule to each subinterval



$$A_0 = \frac{\Delta}{2} (f(x_0) + f(x_1))$$

$$A_1 = \frac{\Delta}{2} (f(x_1) + f(x_2))$$

$$A_2 = \frac{\Delta}{2} (f(x_2) + f(x_3))$$

$$A_3 = \frac{\Delta}{2} (f(x_3) + f(x_4))$$

Trapezoidal rule estimate of area under curve: sum of all A_i 's.

Integration - Trapezoidal Rule

$$\begin{aligned}A &= A_0 + A_1 + A_2 + A_3 \\&= \frac{\Delta}{2} (f(x_0) + f(x_1)) + \frac{\Delta}{2} (f(x_1) + f(x_2)) + \frac{\Delta}{2} (f(x_2) + f(x_3)) + \frac{\Delta}{2} (f(x_3) + f(x_4)) \\&= \frac{\Delta}{2} (f(x_0) + 2f(x_1) + 2f(x_2) + 2f(x_3) + f(x_4)) \\&= \frac{\Delta}{2} \left(f(x_0) + 2 \sum_{i=1}^3 f(x_i) + f(x_4) \right)\end{aligned}$$

Integration - Trapezoidal Rule

$$\begin{aligned}A &= A_0 + A_1 + A_2 + A_3 \\&= \frac{\Delta}{2} (f(x_0) + f(x_1)) + \frac{\Delta}{2} (f(x_1) + f(x_2)) + \frac{\Delta}{2} (f(x_2) + f(x_3)) + \frac{\Delta}{2} (f(x_3) + f(x_4)) \\&= \frac{\Delta}{2} (f(x_0) + 2f(x_1) + 2f(x_2) + 2f(x_3) + f(x_4)) \\&= \frac{\Delta}{2} \left(f(x_0) + 2 \sum_{i=1}^3 f(x_i) + f(x_4) \right)\end{aligned}$$

General formula for Trapezoidal rule applied to interval $a < x < b$, divided into N subintervals of equal size Δ , is

$$\boxed{\int_a^b f(x)dx \approx \frac{\Delta}{2} \left(f(a) + 2 \sum_{i=1}^{N-1} f(x_i) + f(b) \right)} \quad (7)$$

Integration - Trapezoidal Rule

Exercise 1

Use the Trapezoidal rule to calculate integral:

$$\int_2^{10} \frac{1}{x} dx .$$

Use increasing numbers of intervals and compare with exact value of 1.6094379.

Integration - Simpson's Rule

Trapezoidal rule: fit Lagrange polynomial through two points.

Increase accuracy by fitting Lagrange polynomial through three equally spaced points and find area under Lagrange polynomial

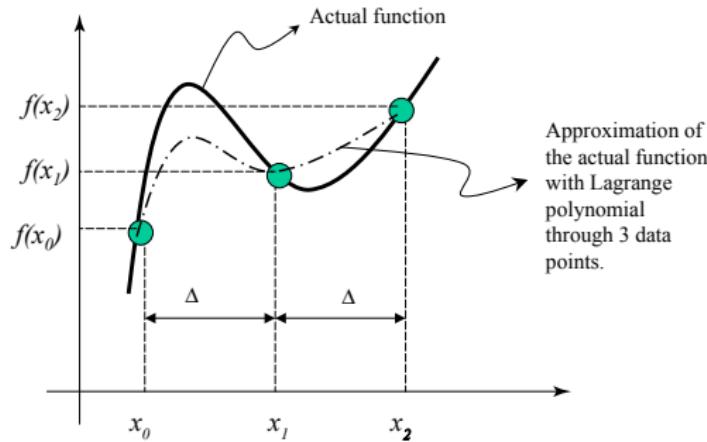


Figure: Fitting a Lagrange polynomial through three data points.

NOTE: Illustration only - looks more like cubic Lagrange function!

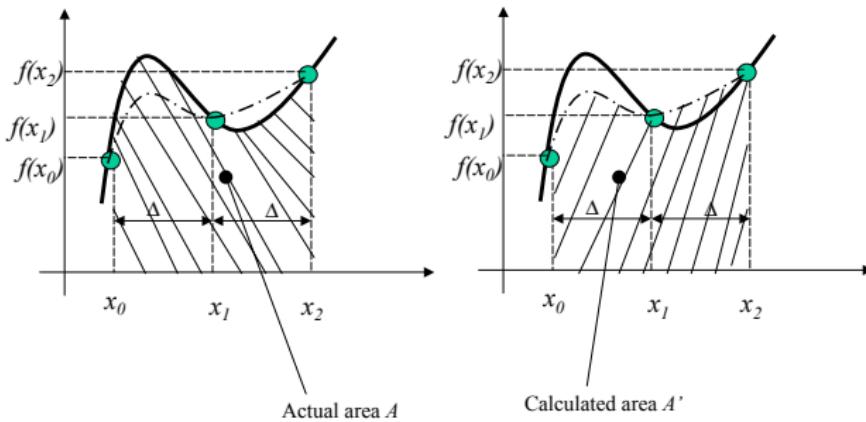
Integration - Simpson's Rule

The function $f(x)$ can be expressed in terms of the second-order Lagrange interpolating Polynomial as

$$\begin{aligned}f(x) &= f_2(x) + \text{error} \\&= \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} f(x_0) \\&\quad + \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} f(x_1) \\&\quad + \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)} f(x_2) \\&\quad + \frac{1}{6}(x - x_0)(x - x_1)(x - x_2)f'''(\xi(x)) ,\end{aligned}$$

where last term is error term.

Integration - Simpson's Rule



With $x_1 - x_0 = x_2 - x_1 = \Delta$, integrate between x_0 and x_2 :

$$\int_{x_0}^{x_2} f(x) \approx \frac{\Delta}{3} (f(x_0) + 4f(x_1) + f(x_2)) \quad (8)$$

Integration - Simpson's Rule

In general Simpson's rule can be written as

$$\int_{x_i}^{x_{i+2}} f(x) \approx \frac{\Delta}{3} (f(x_i) + 4f(x_{i+1}) + f(x_{i+2})) \quad (9)$$

More accurate than Trapezoidal rule.

For increased accuracy, split function into smaller intervals and use the Simpson's rule over sub-intervals.

Integration - Simpson's Rule

In general Simpson's rule can be written as

$$\int_{x_i}^{x_{i+2}} f(x) \approx \frac{\Delta}{3} (f(x_i) + 4f(x_{i+1}) + f(x_{i+2})) \quad (9)$$

More accurate than Trapezoidal rule.

For increased accuracy, split function into smaller intervals and use the Simpson's rule over sub-intervals.

For example, divide up domain of a function into 10 intervals, i.e. if we have 11 data points, $(x_0, f(x_0)), (x_1, f(x_1)), (x_2, f(x_2)), \dots, (x_{10}, f(x_{10}))$,

Integration - Simpson's Rule

$$\begin{aligned}\int_{x_0}^{x_{10}} f(x) dx &\approx \frac{\Delta}{3} (f(x_0) + 4f(x_1) + f(x_2)) + \frac{\Delta}{3} (f(x_2) + 4f(x_3) + f(x_4)) \\&+ \frac{\Delta}{3} (f(x_4) + 4f(x_5) + f(x_6)) + \frac{\Delta}{3} (f(x_6) + 4f(x_7) + f(x_8)) \\&+ \frac{\Delta}{3} (f(x_8) + 4f(x_9) + f(x_{10})) \\&= \frac{\Delta}{3} [f(x_0) + 4(f(x_1) + f(x_3) + f(x_5) + f(x_7) + f(x_9)) \\&+ 2(f(x_2) + f(x_4) + f(x_6) + f(x_8) + f(x_{10})]\end{aligned}$$

Simpson's rule can be written as

$$\boxed{\int_{x_0}^{x_N} f(x) dx \approx \frac{\Delta}{3} \left(f(x_0) + 4 \sum_{i=1 \text{ odd}}^{N-1} f(x_i) + 2 \sum_{i=1 \text{ even}}^{N-2} f(x_i) + f(x_N) \right)} \quad (10)$$

where N is number of intervals.

Integration - Simpson's Rule

$$\begin{aligned}\int_{x_0}^{x_{10}} f(x) dx &\approx \frac{\Delta}{3} (f(x_0) + 4f(x_1) + f(x_2)) + \frac{\Delta}{3} (f(x_2) + 4f(x_3) + f(x_4)) \\&+ \frac{\Delta}{3} (f(x_4) + 4f(x_5) + f(x_6)) + \frac{\Delta}{3} (f(x_6) + 4f(x_7) + f(x_8)) \\&+ \frac{\Delta}{3} (f(x_8) + 4f(x_9) + f(x_{10})) \\&= \frac{\Delta}{3} [f(x_0) + 4(f(x_1) + f(x_3) + f(x_5) + f(x_7) + f(x_9)) \\&+ 2(f(x_2) + f(x_4) + f(x_6) + f(x_8) + f(x_{10})]\end{aligned}$$

Simpson's rule can be written as

$$\boxed{\int_{x_0}^{x_N} f(x) dx \approx \frac{\Delta}{3} \left(f(x_0) + 4 \sum_{i=1 \text{ odd}}^{N-1} f(x_i) + 2 \sum_{i=1 \text{ even}}^{N-2} f(x_i) + f(x_N) \right)} \quad (10)$$

where N is number of intervals.

Simpson's rule only works if N is even i.e. the total number of data points ($N + 1$) is odd.

Differentiation

Differentiation

Differentiation

Derivatives are prevalent in many problems and a numerical approximation is often required.

Recap: Fluid flows represented by [Navier–Stokes eqs.:](#)

$$\frac{\partial \rho}{\partial t} + \frac{\partial}{\partial x_k} (\rho u_k) = 0 ,$$

$$\frac{\partial}{\partial t} (\rho u_i) + \frac{\partial}{\partial x_k} [\rho u_i u_k + p \delta_{ik} - \tau_{ik}] = 0 ,$$

$$\frac{\partial}{\partial t} (\rho E) + \frac{\partial}{\partial x_k} [\rho u_k H + q_k - u_i (\tau_{ik} - \rho \sigma_{ik})] = 0 ,$$

$$\tau_{ik} = \frac{2\mu}{Re} \left(S_{ik} - \frac{1}{3} S_{jj} \delta_{ik} \right) , \quad S_{ik} = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_k} + \frac{\partial u_k}{\partial x_i} \right) ,$$

$$q_k = - \frac{\kappa}{PrEcRe} \frac{\partial T}{\partial x_k}$$

Differentiation

To find a formula to differentiate a function, one can start with a Taylor series expansion

$$\begin{aligned} f(x_{i+1}) &= f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{1}{2!}f''(x_i)(x_{i+1} - x_i)^2 \\ &\quad + \frac{1}{3!}f'''(x_i)(x_{i+1} - x_i)^3 + \dots , \end{aligned} \tag{11}$$

for constant grid spacing, $\Delta = x_{i+1} - x_i$:

$$f(x_{i+1}) = f(x_i) + f'(x_i)\Delta + \frac{1}{2!}f''(x_i)\Delta^2 + \frac{1}{3!}f'''(x_i)\Delta^3 + \mathcal{O}(\Delta^4)$$

Differentiation

Re-arrange to obtain

$$\begin{aligned} f'(x_i) &= \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)} - \frac{1}{2!} f''(x_i) (x_{i+1} - x_i) \\ &\quad - \frac{1}{3!} f'''(x_i) (x_{i+1} - x_i)^2 - \dots \end{aligned} \quad (12)$$

Differentiation

Re-arrange to obtain

$$\begin{aligned}f'(x_i) &= \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)} - \frac{1}{2!} f''(x_i) (x_{i+1} - x_i) \\&\quad - \frac{1}{3!} f'''(x_i) (x_{i+1} - x_i)^2 - \dots\end{aligned}\tag{12}$$

Hence $f'(x)$ can be approximated as

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)}\tag{13}$$

This is called a Forward Difference Approximation (FDA).

Differentiation

Re-arrange to obtain

$$\begin{aligned}f'(x_i) &= \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)} - \frac{1}{2!} f''(x_i) (x_{i+1} - x_i) \\&\quad - \frac{1}{3!} f'''(x_i) (x_{i+1} - x_i)^2 - \dots\end{aligned}\tag{12}$$

Hence $f'(x)$ can be approximated as

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)}\tag{13}$$

This is called a Forward Difference Approximation (FDA).

Higher-order terms neglected are error in approximating $f'(x_i)$.

Differentiation

Re-arrange to obtain

$$\begin{aligned}f'(x_i) &= \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)} - \frac{1}{2!} f''(x_i) (x_{i+1} - x_i) \\&\quad - \frac{1}{3!} f'''(x_i) (x_{i+1} - x_i)^2 - \dots\end{aligned}\tag{12}$$

Hence $f'(x)$ can be approximated as

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{(x_{i+1} - x_i)}\tag{13}$$

This is called a Forward Difference Approximation (FDA).

Higher-order terms neglected are error in approximating $f'(x_i)$.

Leading term in truncation error is

$$E_{FDA} = \frac{1}{2!} f''(x_i) (x_{i+1} - x_i) .$$

Differentiation

A Taylor series expansion can also be used to obtain an expression for $f(x_{i-1})$.

$$\begin{aligned}f(x_{i-1}) &= f(x_i) - f'(x_i)(x_i - x_{i-1}) \\&\quad + \frac{1}{2!}f''(x_i)(x_i - x_{i-1})^2 - \frac{1}{3!}f'''(x_i)(x_i - x_{i-1})^3\end{aligned}\tag{14}$$

Differentiation

A Taylor series expansion can also be used to obtain an expression for $f(x_{i-1})$.

$$\begin{aligned}f(x_{i-1}) &= f(x_i) - f'(x_i)(x_i - x_{i-1}) \\&\quad + \frac{1}{2!}f''(x_i)(x_i - x_{i-1})^2 - \frac{1}{3!}f'''(x_i)(x_i - x_{i-1})^3\end{aligned}\tag{14}$$

Re-arrange to obtain

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{(x_i - x_{i-1})}\tag{15}$$

This is called Backward Difference Approximation (BDA).

Differentiation

A Taylor series expansion can also be used to obtain an expression for $f(x_{i-1})$.

$$\begin{aligned}f(x_{i-1}) &= f(x_i) - f'(x_i)(x_i - x_{i-1}) \\&\quad + \frac{1}{2!}f''(x_i)(x_i - x_{i-1})^2 - \frac{1}{3!}f'''(x_i)(x_i - x_{i-1})^3\end{aligned}\tag{14}$$

Re-arrange to obtain

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{(x_i - x_{i-1})}\tag{15}$$

This is called Backward Difference Approximation (BDA).

Leading term in truncation error for the BDA is

$$E_{BDA} = \frac{1}{2!}f''(x_i)(x_i - x_{i-1}) .$$

Differentiation

Exercise 2

Subtract Eq. (14) from Eq. (11) to derive:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \quad (16)$$

Differentiation

Exercise 2

Subtract Eq. (14) from Eq. (11) to derive:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}} \quad (16)$$

This formula is called the Central Difference Approximation (CDA) and its leading order error is given by

$$\begin{aligned} E_{CDA} &= f''(x_i) \frac{(x_{i+1} - x_i)^2 - (x_i - x_{i-1})^2}{2! (x_{i+1} - x_{i-1})} \\ &+ f'''(x_i) \frac{(x_{i+1} - x_i)^3 + (x_i - x_{i-1})^3}{3! (x_{i+1} - x_{i-1})} \end{aligned}$$

Differentiation

Exercise 3

Assuming all the x_i 's are equally spaced, i.e.

$x_{i+1} - x_i = \Delta = \text{const}$, simplify the FDA, BDA and CDA to

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta}$$

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{\Delta}$$

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta}$$

and show that leading error term is $\mathcal{O}(\Delta)$ in the FDA and BDA,
and $\mathcal{O}(\Delta^2)$ in the CDA.

Differentiation - 2nd-order derivatives

To approximate 2nd derivative, use Equations (14) and (11).

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{1}{2!}f''(x_i)(x_{i+1} - x_i)^2 + \dots$$

$$f(x_{i-1}) = f(x_i) - f'(x_i)(x_i - x_{i-1}) + \frac{1}{2!}f''(x_i)(x_i - x_{i-1})^2 - \dots$$

Differentiation - 2nd-order derivatives

To approximate 2nd derivative, use Equations (14) and (11).

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{1}{2!}f''(x_i)(x_{i+1} - x_i)^2 + \dots$$

$$f(x_{i-1}) = f(x_i) - f'(x_i)(x_i - x_{i-1}) + \frac{1}{2!}f''(x_i)(x_i - x_{i-1})^2 - \dots$$

Add both and re-arrange for $f''(x_i)$:

$$\boxed{f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{\Delta^2}} \quad (17)$$

with $\Delta = x_{i+1} - x_i = x_i - x_{i-1}$.

Differentiation - 2nd-order derivatives

To approximate 2nd derivative, use Equations (14) and (11).

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{1}{2!}f''(x_i)(x_{i+1} - x_i)^2 + \dots$$

$$f(x_{i-1}) = f(x_i) - f'(x_i)(x_i - x_{i-1}) + \frac{1}{2!}f''(x_i)(x_i - x_{i-1})^2 - \dots$$

Add both and re-arrange for $f''(x_i)$:

$$\boxed{f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{\Delta^2}} \quad (17)$$

with $\Delta = x_{i+1} - x_i = x_i - x_{i-1}$.

This formula is again a Central Difference Approximation (CDA) and its leading order error is $\mathcal{O}(\Delta^2)$.

Differentiation - 2nd-order derivatives

Exercise 4

Apply forward difference approximation (FDA) twice to find following approximation for the 2nd derivative

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{\Delta^2} \quad (18)$$

Similarly, apply backward difference approximation (BDA) twice to find following approximation for the 2nd derivative

$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{\Delta^2} \quad (19)$$

Differentiation - Wave number analysis

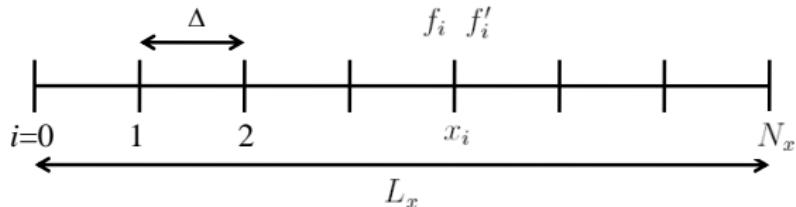
Finite-difference approximations used to approximate $f'(x_i)$

Can easily be extended to high-order accuracy, optimized for specific wave resolution characteristics

Differentiation - Wave number analysis

Finite-difference approximations used to approximate $f'(x_i)$

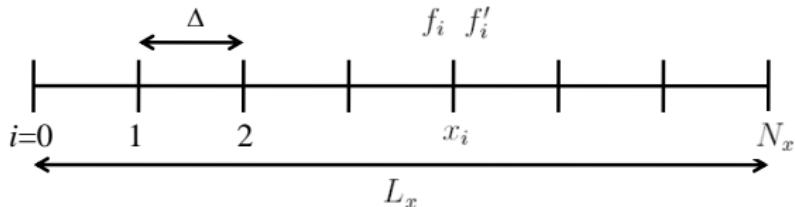
Can easily be extended to high-order accuracy, optimized for specific wave resolution characteristics



Differentiation - Wave number analysis

Finite-difference approximations used to approximate $f'(x_i)$

Can easily be extended to high-order accuracy, optimized for specific wave resolution characteristics



Assume an equidistantly spaced grid with length L_x , discretized with N_x points.

Spacing between points is $\Delta = L_x/N_x$.

Values of a function $f_i = f(x_i)$ are given on each grid point $x_i = \Delta i$, $i = 0, \dots, N_x$.

Differentiation - Wave number analysis

Most general form of CDA becomes

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n), i = 1, \dots, N_x \quad (20)$$

where $\mathcal{O}(\Delta^n)$ is truncation error of scheme.

Differentiation - Wave number analysis

Most general form of CDA becomes

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n), \quad i = 1, \dots, N_x \quad (20)$$

where $\mathcal{O}(\Delta^n)$ is truncation error of scheme.

Coefficients α_j and a_j typically derived using Taylor-series expansion and chosen to give the largest possible exponent $n = 2(N_\alpha + N_a)$

⇒ minimize truncation error or formal order-of-accuracy.

Differentiation - Wave number analysis

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n), i = 1, \dots, N_x$$

$N_\alpha = 0$:

derivative discretized with standard (explicit) finite-difference stencil

$\Rightarrow f'_i$ depends **only** on function values at neighboring nodes

Differentiation - Wave number analysis

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n), i = 1, \dots, N_x$$

$N_\alpha = 0$:

derivative discretized with standard (explicit) finite-difference stencil

$\Rightarrow f'_i$ depends **only** on function values at neighboring nodes

- Schemes with small N_a (small stencil width) require less operations (computationally cheap) than wider stencils
- But: accuracy lower as $n = 2N_a$

Differentiation - Wave number analysis

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n), i = 1, \dots, N_x$$

$N_\alpha \neq 0$:

derivative discretized with **compact** (Padé) finite-difference stencil
→ implicit finite-difference

⇒ f'_i depends on function values **and** derivatives at neighbouring nodes

Differentiation - Wave number analysis

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n), i = 1, \dots, N_x$$

$N_\alpha \neq 0$:

derivative discretized with **compact** (Padé) finite-difference stencil
→ implicit finite-difference

⇒ f'_i depends on function values **and** derivatives at neighbouring nodes

- Require more operations (solution of banded matrix)
- But: accuracy higher because $n = 2(N_\alpha + N_a)$

Differentiation - Wave number analysis

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) + \mathcal{O}(\Delta^n), i = 1, \dots, N_x$$

$N_\alpha \neq 0$:

derivative discretized with **compact** (Padé) finite-difference stencil
→ implicit finite-difference

⇒ f'_i depends on function values **and** derivatives at neighbouring nodes

- Require more operations (solution of banded matrix)
- But: accuracy higher because $n = 2(N_\alpha + N_a)$
- example: five-point stencil with $N_\alpha = 1$ and $N_a = 2$ results in **6th-order accurate** method, instead of 4th-order method

Differentiation - Wave number analysis

Formal order-of-accuracy not the only criterion by which the performance of a finite-difference schemes is judged

Dissipation and dispersion errors of respective schemes can be much more significant

To analyze difference approximation, assume equidistant grid and $f(x)$ periodic on interval $[0, L_x]$ to enable straight-forward Fourier representation

Then $f(x)$ can be represented as

$$f(x) = \sum_m \hat{f}_m \exp\left(\mathrm{i} k_m \frac{x}{\Delta}\right), \quad (21)$$

where k_m is the wavenumber, defined as

$$k_m = 2\pi m \Delta / L_x, \quad m = 0, 1, 2, \dots, N_x/2$$

spanning interval $[0, \pi]$.

Differentiation - Wave number analysis

Exact derivative:

$$f'(x) = \frac{d}{dx} \left[\sum_m \hat{f}_m \exp\left(i k_m \frac{x}{\Delta}\right) \right] = \sum_m i \frac{k_m}{\Delta} \hat{f}_m \exp\left(i k_m \frac{x}{\Delta}\right), \quad (22)$$

Differentiation - Wave number analysis

Exact derivative:

$$f'(x) = \frac{d}{dx} \left[\sum_m \hat{f}_m \exp\left(i k_m \frac{x}{\Delta}\right) \right] = \sum_m i \frac{k_m}{\Delta} \hat{f}_m \exp\left(i k_m \frac{x}{\Delta}\right) , \quad (22)$$

$$f'(x) = \sum_m \hat{f}'_m \exp\left(i k_m \frac{x}{\Delta}\right) ,$$

Differentiation - Wave number analysis

Exact derivative:

$$f'(x) = \frac{d}{dx} \left[\sum_m \hat{f}_m \exp\left(i k_m \frac{x}{\Delta}\right) \right] = \sum_m i \frac{k_m}{\Delta} \hat{f}_m \exp\left(i k_m \frac{x}{\Delta}\right), \quad (22)$$

$$f'(x) = \sum_m \hat{f}'_m \exp\left(i k_m \frac{x}{\Delta}\right),$$

Linearity of Fourier series:

we can consider **each Fourier coefficient** individually.

Error made by particular finite-difference scheme can be quantified by **comparing** $(\hat{f}'_m)_{FD}$ with exact Fourier mode \hat{f}'_m .

Differentiation - Wave number analysis

Illustrate with simple example: Consider the FDA

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta}$$

Differentiation - Wave number analysis

Illustrate with simple example: Consider the FDA

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta}$$

Represent the function with Fourier series

$$f(x_i) = \sum_m \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)$$

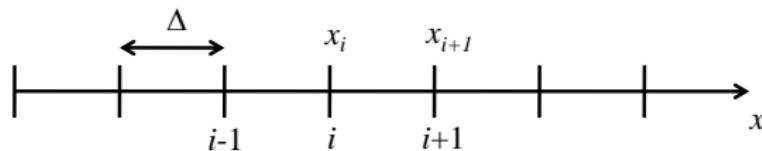
Differentiation - Wave number analysis

Illustrate with simple example: Consider the FDA

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{\Delta}$$

Represent the function with Fourier series

$$f(x_i) = \sum_m \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)$$



$$f(x_{i+1}) = \sum_m \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i + \Delta}{\Delta}\right)$$

Differentiation - Wave number analysis

Insert Fourier representation of $f(x_{i+1})$ and $f(x_i)$ into FDA:

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta} = \frac{1}{\Delta} \sum_m \hat{f}_m \left[\exp\left(\text{i}k_m \frac{x_i + \Delta}{\Delta}\right) - \exp\left(\text{i}k_m \frac{x_i}{\Delta}\right) \right]$$

Differentiation - Wave number analysis

Insert Fourier representation of $f(x_{i+1})$ and $f(x_i)$ into FDA:

$$\begin{aligned} f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta} &= \frac{1}{\Delta} \sum_m \hat{f}_m \left[\exp\left(\mathrm{i}k_m \frac{x_i + \Delta}{\Delta}\right) - \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \right] \\ &= \sum_m \frac{1}{\Delta} \left[\exp\left(\mathrm{i}k_m \frac{\Delta}{\Delta}\right) - 1 \right] \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \end{aligned}$$

Differentiation - Wave number analysis

Insert Fourier representation of $f(x_{i+1})$ and $f(x_i)$ into FDA:

$$\begin{aligned} f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta} &= \frac{1}{\Delta} \sum_m \hat{f}_m \left[\exp\left(\mathrm{i}k_m \frac{x_i + \Delta}{\Delta}\right) - \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \right] \\ &= \sum_m \frac{1}{\Delta} \left[\exp\left(\mathrm{i}k_m \frac{\Delta}{\Delta}\right) - 1 \right] \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \end{aligned}$$

The exact derivative is

$$f'(x_i) = \sum_m \frac{\mathrm{i}k_m}{\Delta} \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)$$

Differentiation - Wave number analysis

Insert Fourier representation of $f(x_{i+1})$ and $f(x_i)$ into FDA:

$$\begin{aligned} f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta} &= \frac{1}{\Delta} \sum_m \hat{f}_m \left[\exp\left(\mathrm{i}k_m \frac{x_i + \Delta}{\Delta}\right) - \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \right] \\ &= \sum_m \frac{1}{\Delta} \left[\exp\left(\mathrm{i}k_m \frac{\Delta}{\Delta}\right) - 1 \right] \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \end{aligned}$$

The exact derivative is

$$f'(x_i) = \sum_m \frac{\mathrm{i}k_m}{\Delta} \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)$$

Compare:

$$\frac{\mathrm{i}k_m}{\Delta} = \frac{1}{\Delta} [\exp(\mathrm{i}k_m) - 1]$$

Differentiation - Wave number analysis

Insert Fourier representation of $f(x_{i+1})$ and $f(x_i)$ into FDA:

$$\begin{aligned} f'(x_i) \approx \frac{f(x_{i+1}) - f(x_i)}{\Delta} &= \frac{1}{\Delta} \sum_m \hat{f}_m \left[\exp\left(\mathrm{i}k_m \frac{x_i + \Delta}{\Delta}\right) - \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \right] \\ &= \sum_m \frac{1}{\Delta} \left[\exp\left(\mathrm{i}k_m \frac{\Delta}{\Delta}\right) - 1 \right] \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right) \end{aligned}$$

The exact derivative is

$$f'(x_i) = \sum_m \frac{\mathrm{i}k_m}{\Delta} \hat{f}_m \exp\left(\mathrm{i}k_m \frac{x_i}{\Delta}\right)$$

Compare:

$$\frac{\mathrm{i}k_m}{\Delta} = \frac{1}{\Delta} [\exp(\mathrm{i}k_m) - 1] \equiv \frac{\mathrm{i}k_m^{mod}}{\Delta}$$

Differentiation - Wave number analysis

Thus, the modified wavenumber for this FDA scheme is

$$k_m^{mod} = i [1 - \exp(i k_m)]$$

Using $\exp(ix) = \cos(x) + i \sin(x)$

$$k_m^{mod} = i [1 - (\cos(k_m) + i \sin(k_m))]$$

Differentiation - Wave number analysis

Thus, the modified wavenumber for this FDA scheme is

$$k_m^{mod} = i[1 - \exp(i k_m)]$$

Using $\exp(ix) = \cos(x) + i \sin(x)$

$$\begin{aligned} k_m^{mod} &= i[1 - (\cos(k_m) + i \sin(k_m))] \\ &= \sin(k_m) + i[1 - \cos(k_m)] \end{aligned}$$

Real and imaginary parts of modified wavenumber:

$$k_{mR}^{mod} = \sin(k_m) \Rightarrow \text{Phase/Dispersion error}$$

Differentiation - Wave number analysis

Thus, the modified wavenumber for this FDA scheme is

$$k_m^{mod} = i[1 - \exp(i k_m)]$$

Using $\exp(ix) = \cos(x) + i \sin(x)$

$$\begin{aligned} k_m^{mod} &= i[1 - (\cos(k_m) + i \sin(k_m))] \\ &= \sin(k_m) + i[1 - \cos(k_m)] \end{aligned}$$

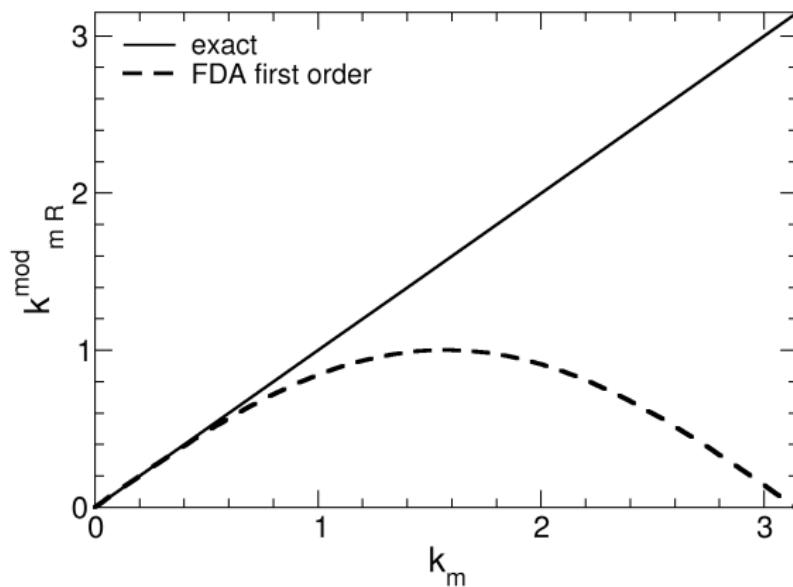
Real and imaginary parts of modified wavenumber:

$$k_{m_R}^{mod} = \sin(k_m) \Rightarrow \text{Phase/Dispersion error}$$

$$k_{m_I}^{mod} = 1 - \cos(k_m) \Rightarrow \text{Amplitude/Dissipation error}$$

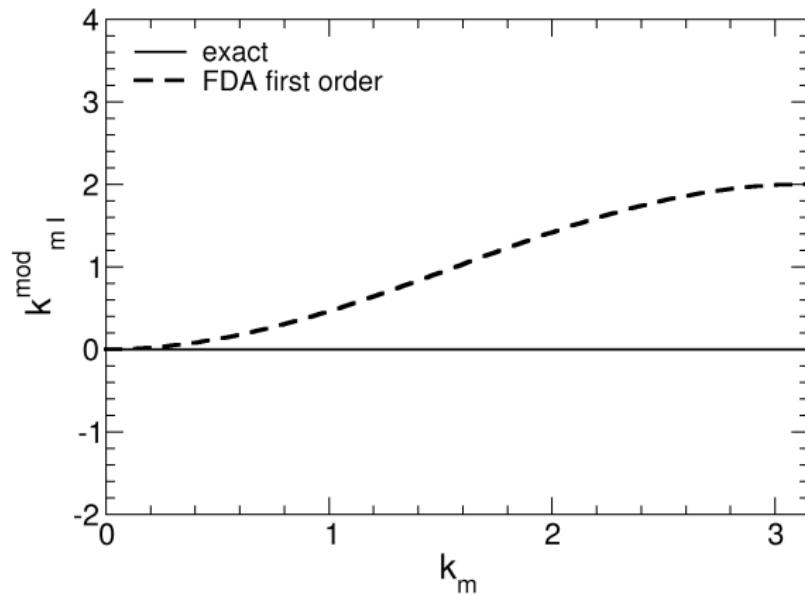
Differentiation - Wave number analysis

Plotting real part



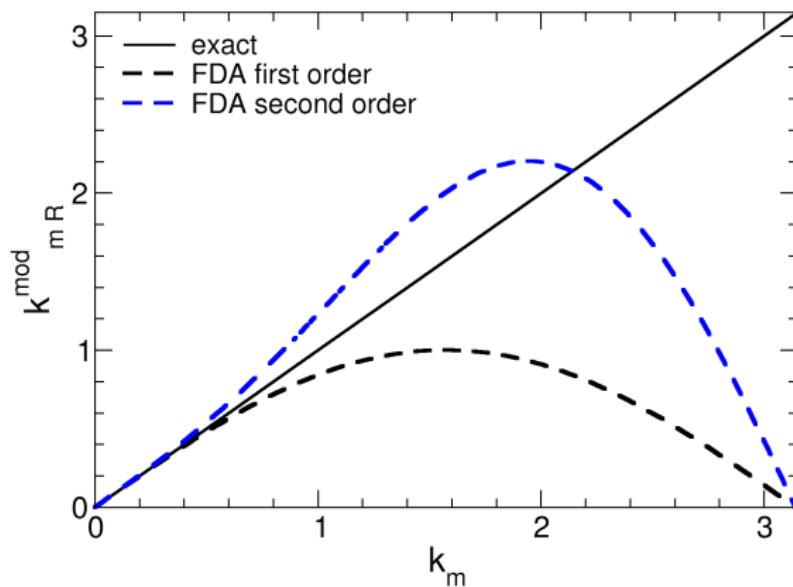
Differentiation - Wave number analysis

Plotting imaginary part



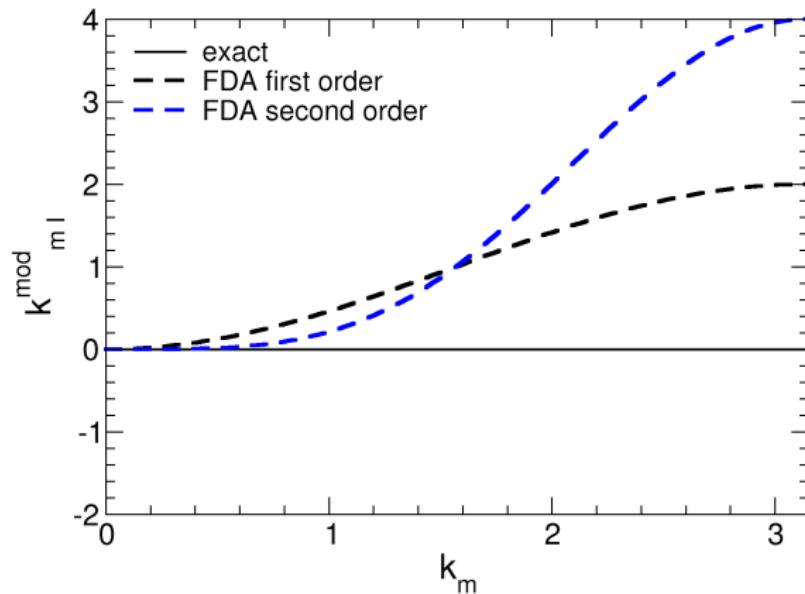
Differentiation - Wave number analysis

Plotting real part



Differentiation - Wave number analysis

Plotting imaginary part



Differentiation - Wave number analysis

Exercise

If the finite difference approximation is a central scheme (CDA),
e.g.:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2\Delta}$$

show that the modified wavenumber is

$$k_m^{mod} = \sin(k_m)$$

Note that the central scheme does not have an imaginary part (i.e. zero amplitude error!).

Differentiation - Wave number analysis

Modified wavenumber for general central finite-difference scheme

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) :$$

$$k_m^{mod} = \frac{\sum_j^{N_a} 2a_j \sin(jk_m)}{1 + \sum_j^{N_\alpha} 2\alpha_j \cos(jk_m)}$$

(23)

Differentiation - Wave number analysis

Modified wavenumber for general central finite-difference scheme

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) :$$

$$k_m^{mod} = \frac{\sum_j^{N_a} 2a_j \sin(jk_m)}{1 + \sum_j^{N_\alpha} 2\alpha_j \cos(jk_m)}$$

(23)

Examples: for CDA used before, $N_a = 1$ and $N_\alpha = 0$

$$k_m^{mod} = \frac{2 \frac{1}{2} \sin(k_m)}{1 + 0} = \sin(k_m)$$

Differentiation - Wave number analysis

Modified wavenumber for general central finite-difference scheme

$$\sum_j^{N_\alpha} \alpha_j (f'_{i+j} + f'_{i-j}) + f'_i = \frac{1}{\Delta} \left(\sum_j^{N_a} a_j (f_{i+j} - f_{i-j}) \right) :$$

$$k_m^{mod} = \frac{\sum_j^{N_a} 2a_j \sin(jk_m)}{1 + \sum_j^{N_\alpha} 2\alpha_j \cos(jk_m)}$$

(23)

Examples: for CDA used before, $N_a = 1$ and $N_\alpha = 0$

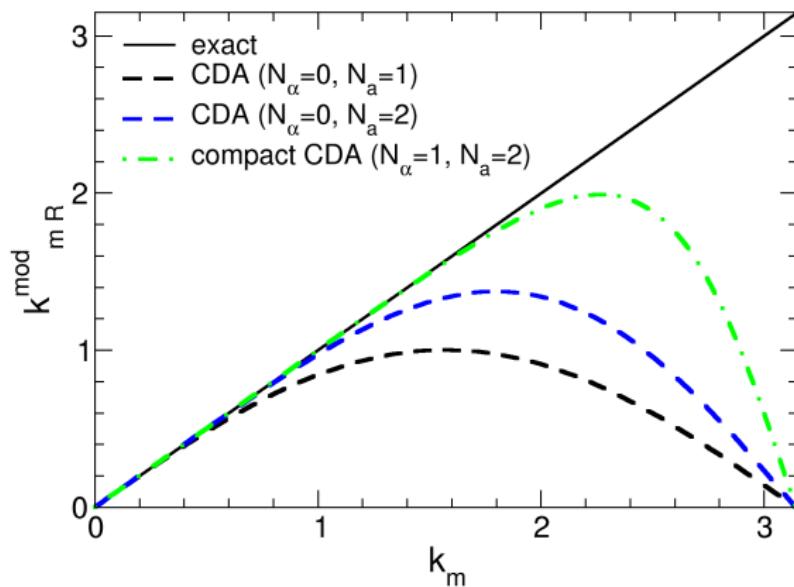
$$k_m^{mod} = \frac{2 \frac{1}{2} \sin(k_m)}{1 + 0} = \sin(k_m)$$

compact finite difference scheme with $N_a = 2$ and $N_\alpha = 1$

$$k_m^{mod} = \frac{2a_1 \sin(k_m) + 2a_2 \sin(2k_m)}{1 + 2\alpha_1 \cos(k_m)}$$

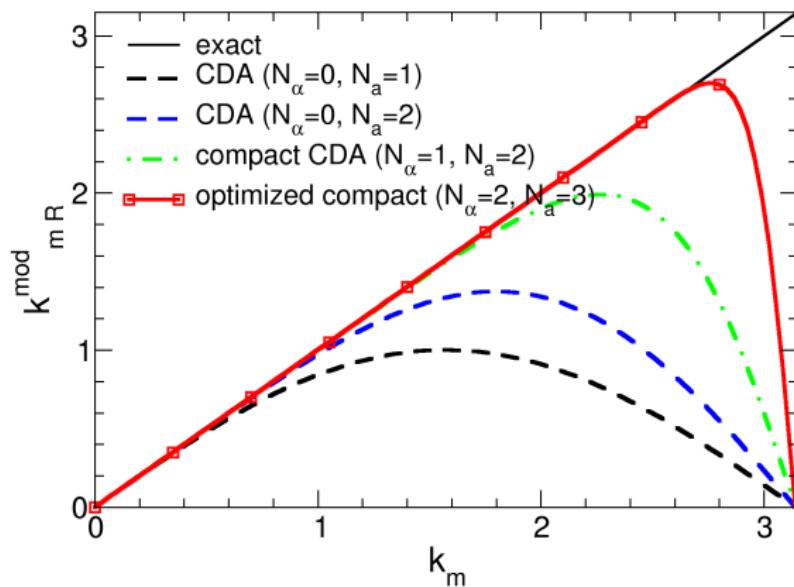
Differentiation - Wave number analysis

Plotting real part



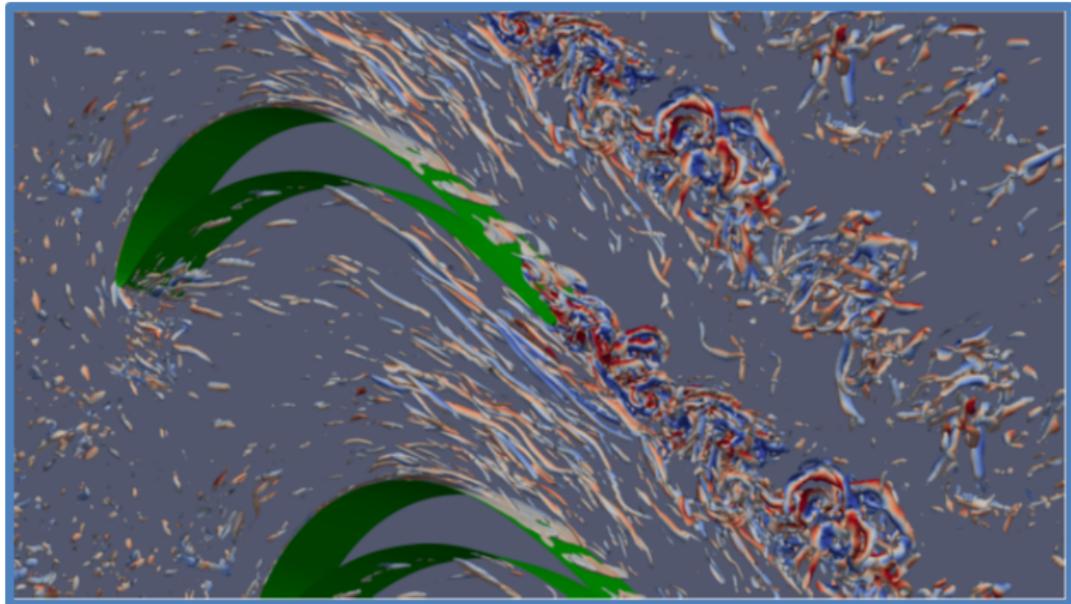
Differentiation - Wave number analysis

Plotting real part



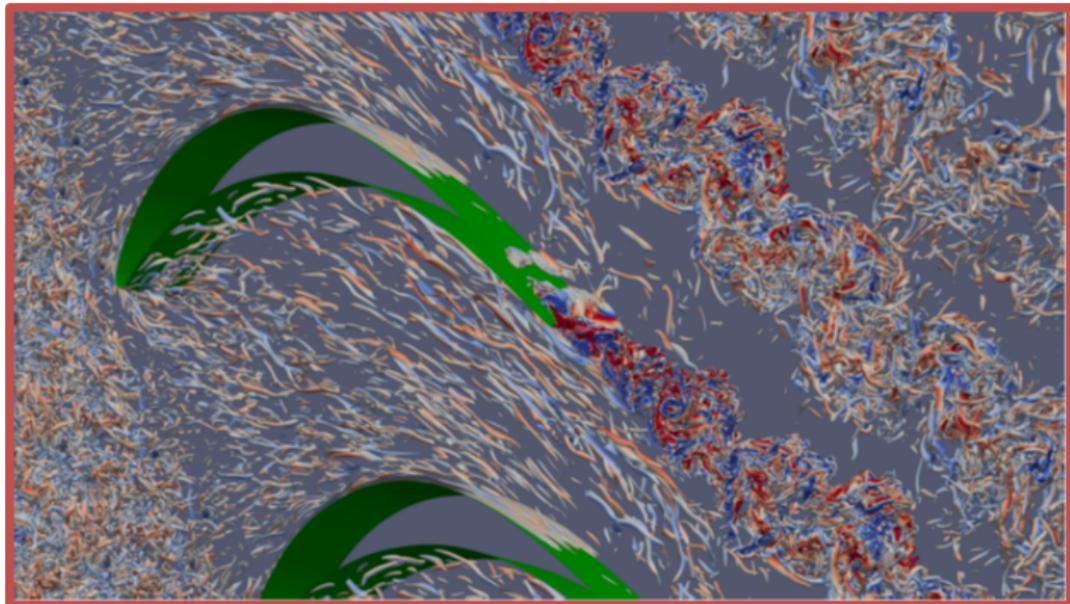
Impact of Wavenumber Accuracy

'Low resolution' scheme - e.g. $N_\alpha = 0, N_a = 2$



Impact of Wavenumber Accuracy

'High resolution' scheme - optimised compact $N_\alpha = 2$, $N_a = 3$



Next week

Ordinary Differential Equations



THE UNIVERSITY OF

MELBOURNE



ENGR30003 Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne
Slides developed by Prof Richard Sandberg

This week

LECTURE 21

Ordinary Differential Equations

Ordinary Differential Equations

In many engineering problems, you will need to solve differential equations that look something like

$$\frac{dx}{dt} = f(t, x) \quad (1)$$

in the domain

$$a \leq t \leq b$$

with the initial condition

$$x(t = a) = \alpha$$

Forward Euler Method

Euler's method is an approach to solve $\frac{dx}{dt} = f(t, x)$.

Consider Taylor's theorem

$$x(t_{n+1}) = x(t_n) + hf(t_n, x_n) + \frac{h^2}{2} \frac{d^2x}{dt^2}(t_n) , \quad (2)$$

where $t_{n+1} - t_n = h$.

Forward Euler Method

Euler's method is an approach to solve $\frac{dx}{dt} = f(t, x)$.

Consider Taylor's theorem

$$x(t_{n+1}) = x(t_n) + hf(t_n, x_n) + \frac{h^2}{2} \frac{d^2x}{dt^2}(t_n) , \quad (2)$$

where $t_{n+1} - t_n = h$.

If we assume h small, can neglect second order term. Thus, we get formula for Euler's method

$$\boxed{x_{n+1} = x_n + hf(t_n, x_n)} , \quad (3)$$

where x_n is numerical approximation of exact solution $x(t_n)$.

Forward Euler Method

Euler's method is an approach to solve $\frac{dx}{dt} = f(t, x)$.

Consider Taylor's theorem

$$x(t_{n+1}) = x(t_n) + hf(t_n, x_n) + \frac{h^2}{2} \frac{d^2x}{dt^2}(t_n) , \quad (2)$$

where $t_{n+1} - t_n = h$.

If we assume h small, can neglect second order term. Thus, we get formula for Euler's method

$$\boxed{x_{n+1} = x_n + hf(t_n, x_n)} , \quad (3)$$

where x_n is numerical approximation of exact solution $x(t_n)$.

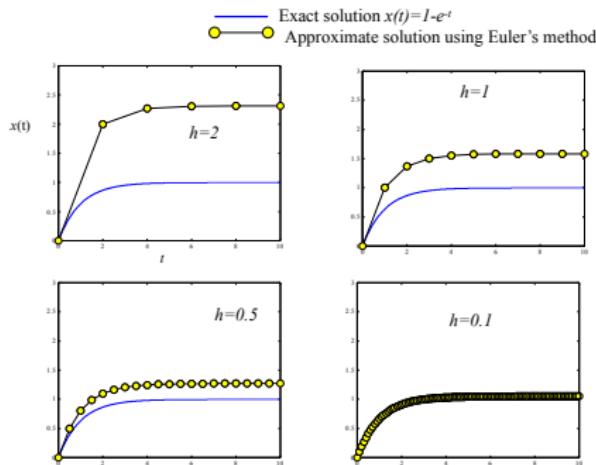
Called **explicit/forward Euler** $\Rightarrow f$ only contains t_n and x_n .

Forward Euler Method

Example: Using forward Euler method, solve

$$\frac{dx}{dt} = e^{-t}$$

for $0 < t < 10$. Use $x(t=0) = 0$ and $h = 2$, $h = 1$, $h = 0.5$ and $h = 0.1$.



Numerical solution gets closer to exact solution for smaller h .

Backward Euler Method

The forward Euler method can often be **unstable**.

A more stable method is **backward/implicit** Euler scheme:

$$x_{n+1} = x_n + h f(t_{n+1}, x_{n+1}) \quad (4)$$

Backward Euler Method

The forward Euler method can often be **unstable**.

A more stable method is **backward/implicit** Euler scheme:

$$x_{n+1} = x_n + h f(t_{n+1}, x_{n+1}) \quad (4)$$

x_{n+1} exist on both sides of the equation \Rightarrow implicit method.

Backward Euler Method

The forward Euler method can often be **unstable**.

A more stable method is **backward/implicit** Euler scheme:

$$x_{n+1} = x_n + h f(t_{n+1}, x_{n+1}) \quad (4)$$

x_{n+1} exist on both sides of the equation \Rightarrow implicit method.

If f is simple function, may be possible to obtain an explicit expression for x_{n+1} .

Backward Euler Method

The forward Euler method can often be **unstable**.

A more stable method is **backward/implicit** Euler scheme:

$$x_{n+1} = x_n + h f(t_{n+1}, x_{n+1}) \quad (4)$$

x_{n+1} exist on both sides of the equation \Rightarrow implicit method.

If f is simple function, may be possible to obtain an explicit expression for x_{n+1} .

If f complicated function, use root finding methods such as the Newton–Raphson method to solve for x_{n+1} .

Crank–Nicolson method

The solution to $\frac{dx}{dt} = f(t, x)$ can also be obtained by integration

$$x(t_{n+1}) = x(t_n) + \int_{t_n}^{t_{n+1}} f [t, x(t)] dt$$

Crank–Nicolson method

The solution to $\frac{dx}{dt} = f(t, x)$ can also be obtained by integration

$$x(t_{n+1}) = x(t_n) + \int_{t_n}^{t_{n+1}} f [t, x(t)] dt$$

Approximate using Trapezoidal rule (earlier notes)

$$\int_{t_n}^{t_{n+1}} f(t, x(t)) dt = \frac{h}{2} [f(t_n, x_n) + f(t_{n+1}, x_{n+1})] + O(h^3) .$$

Crank–Nicolson method

The solution to $\frac{dx}{dt} = f(t, x)$ can also be obtained by integration

$$x(t_{n+1}) = x(t_n) + \int_{t_n}^{t_{n+1}} f [t, x(t)] dt$$

Approximate using Trapezoidal rule (earlier notes)

$$\int_{t_n}^{t_{n+1}} f(t, x(t)) dt = \frac{h}{2} [f(t_n, x_n) + f(t_{n+1}, x_{n+1})] + O(h^3) .$$

$$x_{n+1} = x_n + \frac{h}{2} [f(t_n, x_n) + f(t_{n+1}, x_{n+1})] \quad (5)$$

This is the Crank–Nicolson method (or Trapezoidal method). This is an implicit formula for x_{n+1} .

Crank–Nicolson method

The solution to $\frac{dx}{dt} = f(t, x)$ can also be obtained by integration

$$x(t_{n+1}) = x(t_n) + \int_{t_n}^{t_{n+1}} f [t, x(t)] dt$$

Approximate using Trapezoidal rule (earlier notes)

$$\int_{t_n}^{t_{n+1}} f(t, x(t)) dt = \frac{h}{2} [f(t_n, x_n) + f(t_{n+1}, x_{n+1})] + O(h^3) .$$

$$x_{n+1} = x_n + \frac{h}{2} [f(t_n, x_n) + f(t_{n+1}, x_{n+1})] \quad (5)$$

This is the **Crank–Nicolson method** (or Trapezoidal method). This is an **implicit** formula for x_{n+1} .

If f complicated function, use root finding methods such as Newton–Raphson to solve for x_{n+1} .

Exercise

Exercise 1

Solve

$$\frac{dx}{dt} + 2x = 0$$

for $0 < t < 3$ with $x(t = 0) = 1$ using

- (a) Euler method
- (b) Backward Euler method
- (c) Crank–Nicolson method

Example - Crank–Nicolson

Example: Consider nonlinear first order ordinary differential equation

$$\frac{dx}{dt} + e^x = 0 \quad (6)$$

with initial condition $x(0) = 1.0$.

Example - Crank–Nicolson

Example: Consider nonlinear first order ordinary differential equation

$$\frac{dx}{dt} + e^x = 0 \quad (6)$$

with initial condition $x(0) = 1.0$. The exact solution is

$$x(t) = -\log \left(t + \frac{1}{e} \right) \quad (7)$$

Example - Crank–Nicolson

Example: Consider nonlinear first order ordinary differential equation

$$\frac{dx}{dt} + e^x = 0 \quad (6)$$

with initial condition $x(0) = 1.0$. The exact solution is

$$x(t) = -\log \left(t + \frac{1}{e} \right) \quad (7)$$

Obtain approximate solution with Crank–Nicolson method

$$x_{n+1} + \frac{h}{2}e^{x_{n+1}} = x_n - \frac{h}{2}e^{x_n}$$

Example - Crank–Nicolson

Example: Consider nonlinear first order ordinary differential equation

$$\frac{dx}{dt} + e^x = 0 \quad (6)$$

with initial condition $x(0) = 1.0$. The exact solution is

$$x(t) = -\log \left(t + \frac{1}{e} \right) \quad (7)$$

Obtain approximate solution with Crank–Nicolson method

$$x_{n+1} + \frac{h}{2}e^{x_{n+1}} = x_n - \frac{h}{2}e^{x_n}$$

Rearrange to obtain

$$x_{n+1} + \frac{h}{2}e^{x_{n+1}} - x_n + \frac{h}{2}e^{x_n} = 0. \quad (8)$$

Example - Crank–Nicolson

Example: Consider nonlinear first order ordinary differential equation

$$\frac{dx}{dt} + e^x = 0 \quad (6)$$

with initial condition $x(0) = 1.0$. The exact solution is

$$x(t) = -\log \left(t + \frac{1}{e} \right) \quad (7)$$

Obtain approximate solution with Crank–Nicolson method

$$x_{n+1} + \frac{h}{2}e^{x_{n+1}} = x_n - \frac{h}{2}e^{x_n}$$

Rearrange to obtain

$$x_{n+1} + \frac{h}{2}e^{x_{n+1}} - x_n + \frac{h}{2}e^{x_n} = 0. \quad (8)$$

x_n is known from previous time step \Rightarrow need to find x_{n+1} .

Example - Crank–Nicolson

Use root finding methods, e.g. iterative Newton–Raphson

$$x_{n+1}^{(k+1)} = x_{n+1}^{(k)} - \frac{x_{n+1}^{(k)} + \frac{h}{2}e^{x_{n+1}^{(k)}} - x_n + \frac{h}{2}e^{x_n}}{1 + \frac{h}{2}e^{x_{n+1}^{(k)}}}$$

where $x_{n+1}^{(k)}$ is k 'th guess for the value of x_{n+1} .

Example - Crank–Nicolson

Use root finding methods, e.g. iterative Newton–Raphson

$$x_{n+1}^{(k+1)} = x_{n+1}^{(k)} - \frac{x_{n+1}^{(k)} + \frac{h}{2}e^{x_{n+1}^{(k)}} - x_n + \frac{h}{2}e^{x_n}}{1 + \frac{h}{2}e^{x_{n+1}^{(k)}}}$$

where $x_{n+1}^{(k)}$ is k 'th guess for the value of x_{n+1} .

May use any value for $x_{n+1}^{(k)}$, but logical to use $x_{n+1}^{(0)} = x_n$.

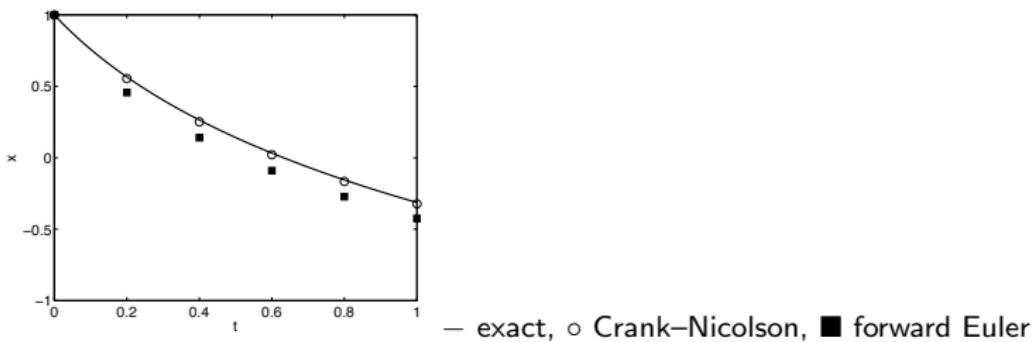
Example - Crank–Nicolson

Use root finding methods, e.g. iterative Newton–Raphson

$$x_{n+1}^{(k+1)} = x_{n+1}^{(k)} - \frac{x_{n+1}^{(k)} + \frac{h}{2}e^{x_{n+1}^{(k)}} - x_n + \frac{h}{2}e^{x_n}}{1 + \frac{h}{2}e^{x_{n+1}^{(k)}}}$$

where $x_{n+1}^{(k)}$ is k 'th guess for the value of x_{n+1} .

May use any value for $x_{n+1}^{(k)}$, but logical to use $x_{n+1}^{(0)} = x_n$.



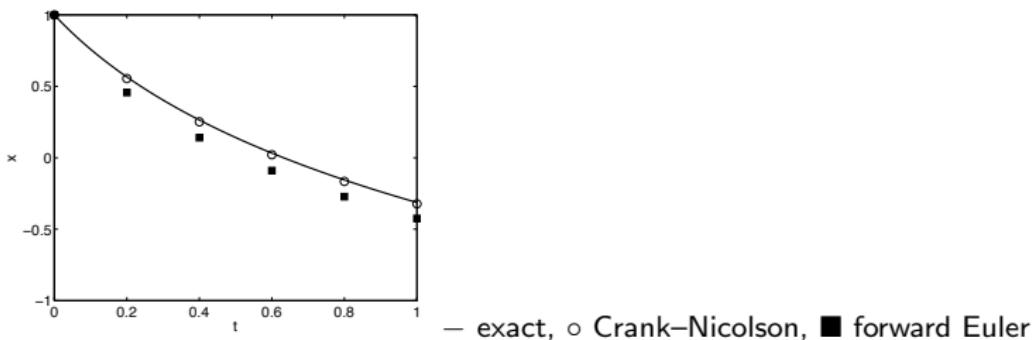
Example - Crank–Nicolson

Use root finding methods, e.g. iterative Newton–Raphson

$$x_{n+1}^{(k+1)} = x_{n+1}^{(k)} - \frac{x_{n+1}^{(k)} + \frac{h}{2}e^{x_{n+1}^{(k)}} - x_n + \frac{h}{2}e^{x_n}}{1 + \frac{h}{2}e^{x_{n+1}^{(k)}}}$$

where $x_{n+1}^{(k)}$ is k 'th guess for the value of x_{n+1} .

May use any value for $x_{n+1}^{(k)}$, but logical to use $x_{n+1}^{(0)} = x_n$.



Crank–Nicolson solution more accurate than Euler ($h = 0.2$).

Runge–Kutta Methods

Most popular method in solving initial value problems.

Start again with $\frac{dx}{dt} = f(t, x)$.

In general, the Runge–Kutta schemes can be written as

$$x_{n+1} = x_n + \phi(x_n, t_n, h)h \quad (9)$$

where ϕ is known as incremental function and can be interpreted as slope used to predict new value of x .

Runge–Kutta Methods

In general, ϕ can be written as

$$\phi = a_1 k_1 + a_2 k_2 + a_3 k_3 + a_4 k_4 + \cdots a_N k_N \quad (10)$$

where

$$k_1 = f(t_n, x_n)$$

$$k_2 = f(t_n + p_1 h, x_n + q_{11} k_1 h)$$

$$k_3 = f(t_n + p_2 h, x_n + q_{21} k_1 h + q_{22} k_2 h)$$

$$k_4 = f(t_n + p_3 h, x_n + q_{31} k_1 h + q_{32} k_2 h + q_{33} k_3 h)$$

$$\begin{matrix} \vdots & \vdots & \vdots \end{matrix}$$

$$k_N = f(t_n + p_{N-1} h, x_i + q_{N-1,1} k_1 h + q_{N-1,2} k_2 h + \cdots + q_{N-1,N-1} k_{N-1} h)$$

Runge–Kutta Methods

In general, ϕ can be written as

$$\phi = a_1 k_1 + a_2 k_2 + a_3 k_3 + a_4 k_4 + \cdots a_N k_N \quad (10)$$

where

$$k_1 = f(t_n, x_n)$$

$$k_2 = f(t_n + p_1 h, x_n + q_{11} k_1 h)$$

$$k_3 = f(t_n + p_2 h, x_n + q_{21} k_1 h + q_{22} k_2 h)$$

$$k_4 = f(t_n + p_3 h, x_n + q_{31} k_1 h + q_{32} k_2 h + q_{33} k_3 h)$$

$$\begin{matrix} \vdots & \vdots & \vdots \end{matrix}$$

$$k_N = f(t_n + p_{N-1} h, x_i + q_{N-1,1} k_1 h + q_{N-1,2} k_2 h + \cdots + q_{N-1,N-1} k_{N-1} h)$$

For $N = 1$, we get the first order Runge–Kutta scheme.

Runge–Kutta Methods

In general, ϕ can be written as

$$\phi = a_1 k_1 + a_2 k_2 + a_3 k_3 + a_4 k_4 + \cdots a_N k_N \quad (10)$$

where

$$k_1 = f(t_n, x_n)$$

$$k_2 = f(t_n + p_1 h, x_n + q_{11} k_1 h)$$

$$k_3 = f(t_n + p_2 h, x_n + q_{21} k_1 h + q_{22} k_2 h)$$

$$k_4 = f(t_n + p_3 h, x_n + q_{31} k_1 h + q_{32} k_2 h + q_{33} k_3 h)$$

$$\begin{matrix} \vdots & \vdots & \vdots \end{matrix}$$

$$k_N = f(t_n + p_{N-1} h, x_i + q_{N-1,1} k_1 h + q_{N-1,2} k_2 h + \cdots + q_{N-1,N-1} k_{N-1} h)$$

For $N = 1$, we get the first order Runge–Kutta scheme.

Just the same as Euler integration scheme.

Runge–Kutta Methods

For $N = 2$ we get $\phi = a_1 k_1 + a_2 k_2$. Substituting gives

$$\begin{aligned}x_{n+1} &= x_n + (a_1 k_1 + a_2 k_2) h \\&= x_n + a_1 f(t_n, x_n)h + a_2 f(t_n + p_1 h, x_n + q_{11} k_1 h)h\end{aligned}\tag{11}$$

Runge–Kutta Methods

For $N = 2$ we get $\phi = a_1 k_1 + a_2 k_2$. Substituting gives

$$\begin{aligned}x_{n+1} &= x_n + (a_1 k_1 + a_2 k_2) h \\&= x_n + a_1 f(t_n, x_n) h + a_2 \color{red}{f(t_n + p_1 h, x_n + q_{11} k_1 h)} h\end{aligned}\tag{11}$$

Last term can be approximated by Taylor-series approximation

$$f(t + h, x + \Delta) = f(t, x) + h \frac{\partial f}{\partial t} + \Delta \frac{\partial f}{\partial x}$$

Runge–Kutta Methods

For $N = 2$ we get $\phi = a_1 k_1 + a_2 k_2$. Substituting gives

$$\begin{aligned}x_{n+1} &= x_n + (a_1 k_1 + a_2 k_2) h \\&= x_n + a_1 f(t_n, x_n) h + a_2 \color{red}{f(t_n + p_1 h, x_n + q_{11} k_1 h)} h\end{aligned}\tag{11}$$

Last term can be approximated by Taylor-series approximation

$$f(t + h, x + \Delta) = f(t, x) + h \frac{\partial f}{\partial t} + \Delta \frac{\partial f}{\partial x}$$

Using this relationship,

$$f(t_n + p_1 h, x_n + q_{11} k_1 h) = f(t_n, x_n) + p_1 h \frac{\partial f}{\partial t} + q_{11} k_1 h \frac{\partial f}{\partial x} \tag{12}$$

Runge–Kutta Methods

We know that $k_1 = f(t_n, x_n)$

$$f(t_n + p_1 h, x_n + q_{11} k_1 h) = f(t_n, x_n) + p_1 h \frac{\partial f}{\partial t} + q_{11} f(t_n, x_n) h \frac{\partial f}{\partial x}$$

Runge–Kutta Methods

We know that $k_1 = f(t_n, x_n)$

$$f(t_n + p_1 h, x_n + q_{11} k_1 h) = f(t_n, x_n) + p_1 h \frac{\partial f}{\partial t} + q_{11} f(t_n, x_n) h \frac{\partial f}{\partial x}$$

$$x_{n+1} = x_n + (a_1 + a_2) f(t_n, x_n) h + a_2 p_1 \frac{\partial f}{\partial t} h^2 + a_2 q_{11} f(t_n, x_n) h \frac{\partial f}{\partial x} h^2$$

Runge–Kutta Methods

We know that $k_1 = f(t_n, x_n)$

$$f(t_n + p_1 h, x_n + q_{11} k_1 h) = f(t_n, x_n) + p_1 h \frac{\partial f}{\partial t} + q_{11} f(t_n, x_n) h \frac{\partial f}{\partial x}$$

$$x_{n+1} = x_n + (a_1 + a_2)f(t_n, x_n)h + a_2 p_1 \frac{\partial f}{\partial t} h^2 + a_2 q_{11} f(t_n, x_n) \frac{\partial f}{\partial x} h^2$$

Can also write Taylor series expansion for x in terms of t as

$$x(t_{n+1}) = x(t_n) + \frac{dx}{dt}(t_n) h + \frac{d^2x}{dt^2}(t_n) \frac{h^2}{2!}$$

with higher order terms neglected.

Runge–Kutta Methods

$$x_{n+1} = x_n + \frac{dx}{dt}h + \frac{d^2x}{dt^2}\frac{h^2}{2!}$$

Using $\frac{dx}{dt} = f(t, x)$

$$x_{n+1} = x_n + f(t_n, x_n)h + \frac{df(t_n, x_n)}{dt}\frac{h^2}{2}$$

Runge–Kutta Methods

$$x_{n+1} = x_n + \frac{dx}{dt}h + \frac{d^2x}{dt^2}\frac{h^2}{2!}$$

Using $\frac{dx}{dt} = f(t, x)$

$$x_{n+1} = x_n + f(t_n, x_n)h + \frac{df(t_n, x_n)}{dt}\frac{h^2}{2}$$

$$\begin{aligned} df &= \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial x}dx \\ \frac{df}{dt} &= \frac{\partial f}{\partial t} + \frac{\partial f}{\partial x}\frac{dx}{dt} \end{aligned} \tag{13}$$

Runge–Kutta Methods

$$x_{n+1} = x_n + f(t_n, x_n)h + \frac{1}{2} \frac{\partial f}{\partial t} h^2 + \frac{1}{2} \frac{\partial f}{\partial x} f(t_n, x_n)h^2$$

Runge–Kutta Methods

$$x_{n+1} = x_n + f(t_n, x_n)h + \frac{1}{2} \frac{\partial f}{\partial t} h^2 + \frac{1}{2} \frac{\partial f}{\partial x} f(t_n, x_n) h^2$$

Compare with

$$x_{n+1} = x_n + (a_1 + a_2)f(t_n, x_n)h + a_2 p_1 \frac{\partial f}{\partial t} h^2 + a_2 q_{11} f(t_n, x_n) \frac{\partial f}{\partial x} h^2$$

gives following three equations

$$a_1 + a_2 = 1$$

$$a_2 p_1 = \frac{1}{2}$$

$$a_2 q_{11} = \frac{1}{2}$$

Runge–Kutta Methods

$$x_{n+1} = x_n + f(t_n, x_n)h + \frac{1}{2} \frac{\partial f}{\partial t} h^2 + \frac{1}{2} \frac{\partial f}{\partial x} f(t_n, x_n) h^2$$

Compare with

$$x_{n+1} = x_n + (a_1 + a_2)f(t_n, x_n)h + a_2 p_1 \frac{\partial f}{\partial t} h^2 + a_2 q_{11} f(t_n, x_n) \frac{\partial f}{\partial x} h^2$$

gives following three equations

$$a_1 + a_2 = 1$$

$$a_2 p_1 = \frac{1}{2}$$

$$a_2 q_{11} = \frac{1}{2}$$

Four unknowns (a_1, a_2, p_1 and q_{11}).

Runge–Kutta Methods

$$x_{n+1} = x_n + f(t_n, x_n)h + \frac{1}{2} \frac{\partial f}{\partial t} h^2 + \frac{1}{2} \frac{\partial f}{\partial x} f(t_n, x_n) h^2$$

Compare with

$$x_{n+1} = x_n + (a_1 + a_2)f(t_n, x_n)h + a_2 p_1 \frac{\partial f}{\partial t} h^2 + a_2 q_{11} f(t_n, x_n) \frac{\partial f}{\partial x} h^2$$

gives following three equations

$$a_1 + a_2 = 1$$

$$a_2 p_1 = \frac{1}{2}$$

$$a_2 q_{11} = \frac{1}{2}$$

Four unknowns (a_1, a_2, p_1 and q_{11}).

Only three equations \Rightarrow there cannot be a unique solution.

Runge–Kutta Methods

One possible solution is to set $a_2 = \frac{1}{2}$:

$$\begin{aligned}a_1 &= \frac{1}{2} \\p_1 &= 1 \\q_{11} &= 1\end{aligned}$$

Hence, one possible second order Runge-Kutta (RK-2) time stepping scheme is

$$x_{n+1} = x_n + \left(\frac{1}{2}k_1 + \frac{1}{2}k_2 \right) h \quad (14)$$

where

$$\begin{aligned}k_1 &= f(t_n, x_n) \\k_2 &= f(t_n + h, x_n + hk_1)\end{aligned}$$

Runge–Kutta Methods

In practice, implementation looks like:

$$\begin{aligned}x_1 &= x_n + hf(x_n) \\x_{n+1} &= x_n + \frac{h}{2} [f(x_n) + f(x_1)] .\end{aligned}\tag{15}$$

Runge–Kutta Methods - 4th order scheme

The 4th order Runge Kutta scheme is by far most popular numerical method for solving ODEs.

The formula for this scheme can be written as

$$x_{n+1} = x_n + \left(\frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4 \right) h \quad (16)$$

where

$$k_1 = f(t_n, x_n)$$

$$k_2 = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2\right)$$

$$k_4 = f(t_n + h, x_n + hk_3)$$

Runge–Kutta Methods

In practice, implementation looks like:

$$\begin{aligned}x_1 &= x_n + \frac{h}{2} f(x_n) \\x_2 &= x_n + \frac{h}{2} f(x_1) \\x_3 &= x_n + h f(x_2) \\x_{n+1} &= x_n + \frac{h}{6} [f(x_n) + 2f(x_1) + 2f(x_2) + f(x_3)] ,\end{aligned}\tag{17}$$

Three intermediate levels of variables (x_1, x_2, x_3) needed.
⇒ implication for memory.

Runge–Kutta Methods

Algorithm can be manipulated to memory-conserving form (only two intermediate variables):

$$\begin{aligned}x_1 &= x_n + \frac{h}{2} f(x_n) \\x_2 &= x_n + \frac{h}{2} f(x_1) \\x_1 &= x_1 + 2x_2 \\x_2 &= x_n + h f(x_2) \\x_1 &= \frac{1}{3} (-x_n + x_1 + x_2) \\x_{n+1} &= x_1 + \frac{h}{6} f(x_2).\end{aligned}\tag{18}$$

Stability and Error Analysis

Consider the model problem

$$\frac{dx}{dt} = \lambda x \quad (19)$$

where λ is a constant (can be complex number).

Stability and Error Analysis

Consider the model problem

$$\frac{dx}{dt} = \lambda x \quad (19)$$

where λ is a constant (can be complex number).

For most engineering problems, real part of λ is negative.

⇒ solution will decay over time.

Stability and Error Analysis

Consider the model problem

$$\frac{dx}{dt} = \lambda x \quad (19)$$

where λ is a constant (can be complex number).

For most engineering problems, real part of λ is negative.

⇒ solution will decay over time.

Applying Euler method (with timestep $\Delta t = h$)

$$x_{n+1} = x_n + \lambda h x_n \quad (20)$$

Stability and Error Analysis

Consider the model problem

$$\frac{dx}{dt} = \lambda x \quad (19)$$

where λ is a constant (can be complex number).

For most engineering problems, real part of λ is negative.

⇒ solution will decay over time.

Applying Euler method (with timestep $\Delta t = h$)

$$x_{n+1} = x_n + \lambda h x_n \quad (20)$$

or $x_{n+1} = (1 + \lambda h) x_n$

Stability and Error Analysis

Thus error at any time step n can be written as

$$x_n = x_0 (1 + \lambda h)^n \quad (21)$$

With $\lambda = \lambda_R + i\lambda_I$, get

$$x_n = x_0 (1 + h\lambda_R + ih\lambda_I)^n = x_0 \sigma^n \quad (22)$$

Stability and Error Analysis

Thus error at any time step n can be written as

$$x_n = x_0 (1 + \lambda h)^n \quad (21)$$

With $\lambda = \lambda_R + i\lambda_I$, get

$$x_n = x_0 (1 + h\lambda_R + ih\lambda_I)^n = x_0 \sigma^n \quad (22)$$

σ is amplification factor.

Stability and Error Analysis

Thus error at any time step n can be written as

$$x_n = x_0 (1 + \lambda h)^n \quad (21)$$

With $\lambda = \lambda_R + i\lambda_I$, get

$$x_n = x_0 (1 + h\lambda_R + ih\lambda_I)^n = x_0 \sigma^n \quad (22)$$

σ is **amplification factor**.

Requirement for **stability of scheme**:

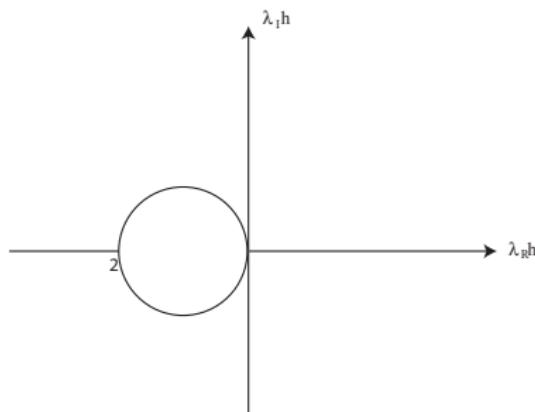
$$\begin{aligned} |\sigma| &\leq 1 \\ |\sigma|^2 &= (1 + h\lambda_R)^2 + (h\lambda_I)^2 \leq 1 \end{aligned} \quad (23)$$

Stability and Error Analysis

$$|\sigma|^2 = (1 + h\lambda_R)^2 + (h\lambda_I)^2 \leq 1$$

This is just circle of radius 1 centred on $(-1, 0)$.

This plot is called **stability diagram**



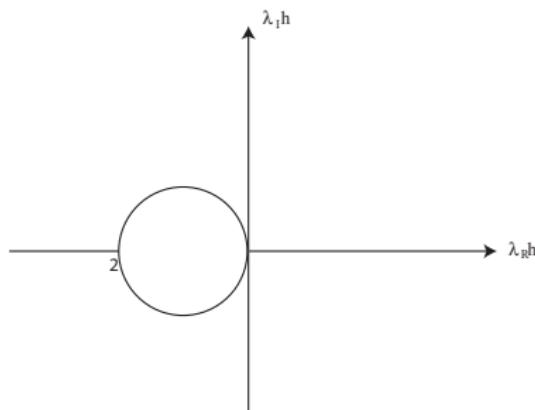
Stability diagram of Euler method.

Stability and Error Analysis

$$|\sigma|^2 = (1 + h\lambda_R)^2 + (h\lambda_I)^2 \leq 1$$

This is just circle of radius 1 centred on $(-1, 0)$.

This plot is called **stability diagram**



If λ real and negative, for numerical method to be stable,

$$h \leq \frac{2}{|\lambda|} \quad (24)$$

Stability diagram of Euler method.

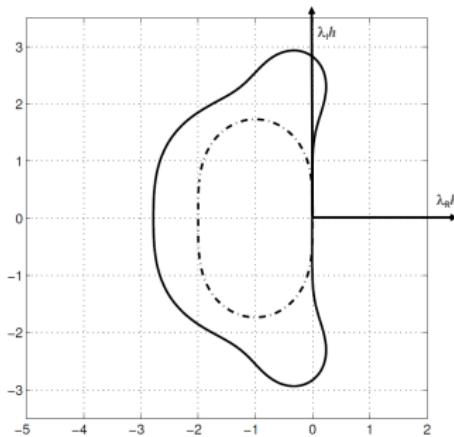
Stability and Error Analysis

For the second-order R-K method, we get

$$1 + \lambda h + \frac{\lambda^2 h^2}{2} - e^{i\theta} = 0$$

4th order R-K method, stability region obtained from

$$\lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \frac{\lambda^4 h^4}{24} + 1 - e^{i\theta} = 0$$



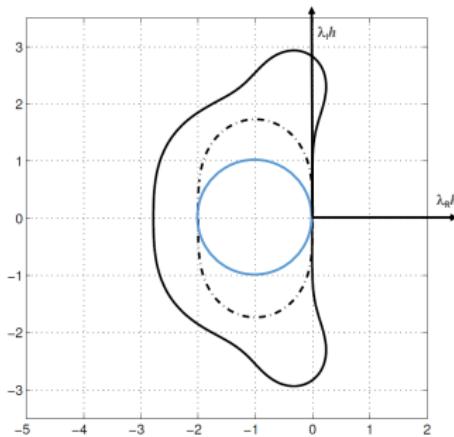
Stability and Error Analysis

For the second-order R-K method, we get

$$1 + \lambda h + \frac{\lambda^2 h^2}{2} - e^{i\theta} = 0$$

4th order R-K method, stability region obtained from

$$\lambda h + \frac{\lambda^2 h^2}{2} + \frac{\lambda^3 h^3}{6} + \frac{\lambda^4 h^4}{24} + 1 - e^{i\theta} = 0$$



Next

Regression



THE UNIVERSITY OF

MELBOURNE



ENGR30003 Numerical Programming for Engineers

Semester 2, 2020

Coordinator and lecturer:

Dr Aman G. Kidanemariam
aman.kidanemariam@unimelb.edu.au

Copywrite © The University of Melbourne
Slides developed by Prof Richard Sandberg

This week

LECTURE 22

Regression

Announcement

Final Exam timetable now available:

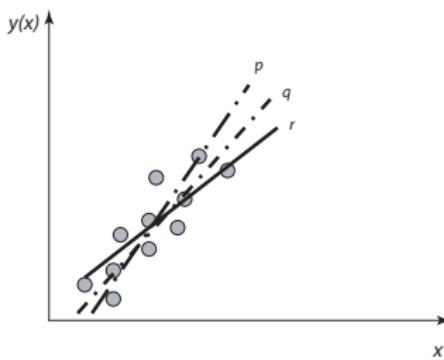
ENGR30003 scheduled for:

Thursday 19/11/2020, 3:00pm

120 min + time for reading/uploading

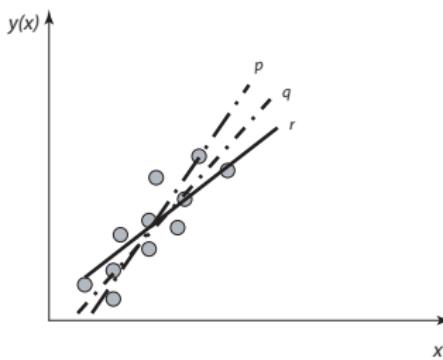
Least Squares Approximations: lecture 15 recap

You have N lots of (x_i, y_i) values from an experiment and you want a straight line fit, do I choose p , q or r ?



Least Squares Approximations: lecture 15 recap

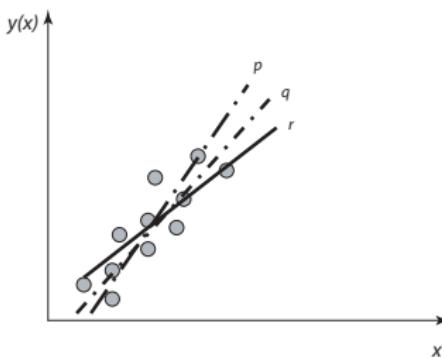
You have N lots of (x_i, y_i) values from an experiment and you want a straight line fit, do I choose p , q or r ?



1. Define your equation: $\hat{y} = ax + b$

Least Squares Approximations: lecture 15 recap

You have N lots of (x_i, y_i) values from an experiment and you want a straight line fit, do I choose p , q or r ?



1. Define your equation: $\hat{y} = ax + b$
2. **AIM:** Minimise: $S(a, b) = \sum_{i=1}^N (y_i - ax_i - b)^2 = \sum_{i=1}^N \epsilon_i^2$, where ϵ_i is the difference between the model and the observed value for i^{th} data point.

Least Squares Approximations: lecture 15 recap

You have N lots of (x_i, y_i) values from an experiment and you want a straight line fit,

1. Define your equation: $\hat{y} = ax + b$
2. **AIM:** Minimise: $S(a, b) = \sum_{i=1}^N (y_i - ax_i - b)^2 = \sum_{i=1}^N \epsilon_i^2$, where ϵ_i is the difference between the model and the observed value for i^{th} data point.
3. S is minimised when $\partial S / \partial a$, $\partial S / \partial b$ are 0.

Least Squares Approximations: lecture 15 recap

You have N lots of (x_i, y_i) values from an experiment and you want a straight line fit,

1. Define your equation: $\hat{y} = ax + b$
2. **AIM:** Minimise: $S(a, b) = \sum_{i=1}^N (y_i - ax_i - b)^2 = \sum_{i=1}^N \epsilon_i^2$, where ϵ_i is the difference between the model and the observed value for i^{th} data point.
3. S is minimised when $\partial S / \partial a, \partial S / \partial b$ are 0.
4. So differentiate, write your equations in matrix form:

$$\begin{bmatrix} \sum_{i=1}^N x_i^2 & \sum_{i=1}^N x_i \\ \sum_{i=1}^N x_i & N \end{bmatrix} \begin{Bmatrix} a \\ b \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^N x_i y_i \\ \sum_{i=1}^N y_i \end{Bmatrix} \quad (1)$$

Least Squares Approximations: lecture 15 recap

You have N lots of (x_i, y_i) values from an experiment and you want a straight line fit,

1. Define your equation: $\hat{y} = ax + b$
2. **AIM:** Minimise: $S(a, b) = \sum_{i=1}^N (y_i - ax_i - b)^2 = \sum_{i=1}^N \epsilon_i^2$, where ϵ_i is the difference between the model and the observed value for i^{th} data point.
3. S is minimised when $\partial S / \partial a, \partial S / \partial b$ are 0.
4. So differentiate, write your equations in matrix form:

$$\begin{bmatrix} \sum_{i=1}^N x_i^2 & \sum_{i=1}^N x_i \\ \sum_{i=1}^N x_i & N \end{bmatrix} \begin{Bmatrix} a \\ b \end{Bmatrix} = \begin{Bmatrix} \sum_{i=1}^N x_i y_i \\ \sum_{i=1}^N y_i \end{Bmatrix} \quad (1)$$

5. Solve using matrix methods learnt earlier in the course.

Least Squares Approximations

Alternative way of solving least squares problem.

Exercise from the assignment

Begin from Eq. 1, show that:

$$b = \bar{y} - a\bar{x}, \quad a = \frac{\sum_{i=1}^N (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^N (x_i - \bar{x})^2}$$

- **Hint:** $\bar{x} = \sum_{i=1}^N x_i / N$ is the mean of x_i .

Least Squares Approximations

The full name is Linear Least Squares **Regression** and is just one method in a huge variety of curve fitting techniques. You have already seen the extension to **polynomial** fitting:

$$\hat{y} = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

We can also extend to **multiple regression** (also called the general linear model):

$$\hat{y} = a_0 + a_1x_1 + a_2x_2 + \cdots + a_nx_n$$

where \hat{y} is now a function of n independent variables

$$\hat{y} = \hat{y}(x_1, \dots, x_n).$$

Multiple Linear Regression

When fitting a model of n variables, define x_{ij} the i^{th} observation of the j^{th} variable:

$$\hat{y}_i = a_0 + a_1 x_{i1} + a_2 x_{i2} + \cdots + a_j x_{ij} + \cdots + a_n x_{in}$$

our matrix becomes (this is left as a fun exercise):

$$\begin{bmatrix} N & \sum x_{i1} & \sum x_{i2} & \dots & \sum x_{in} \\ \sum x_{i1} & \sum x_{i1}x_{i1} & \sum x_{i2}x_{i1} & \dots & \sum x_{in}x_{i1} \\ \sum x_{i2} & \sum x_{i1}x_{i2} & \sum x_{i2}x_{i2} & \dots & \sum x_{in}x_{i2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_{in} & \sum x_{i1}x_{in} & \sum x_{i2}x_{in} & \dots & \sum x_{in}x_{in} \end{bmatrix} \begin{Bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{Bmatrix} = \begin{Bmatrix} \sum y_i \\ \sum x_{i1}y_i \\ \sum x_{i2}y_i \\ \vdots \\ \sum x_{in}y_i \end{Bmatrix}$$

Now we see the purpose of our linear algebra matrix methods!

Assumptions for regression

Regression of this kind assumes a few things:

Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i

Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i
 - This is obvious: we build our model as a linear function:

$$y = az_1 + bz_2 + c$$

Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i
 - This is obvious: we build our model as a linear function:

$$y = az_1 + bz_2 + c$$

- z_1 and z_2 may be non-linear, but y must be linear in z_i :

$$z_1 = x_1^3, \quad z_2 = \sin(x_2)$$

Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i
2. No autocorrelation.

Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i
2. No autocorrelation.
 - Consecutive errors in our model are not correlated.

$$\epsilon_{i+1} \neq f(\epsilon_i)$$

- Also each predictor variable is independent of each other:

$$\epsilon_i \neq f(x_{i1}, \dots, x_{in})$$

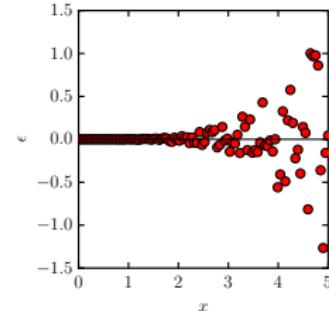
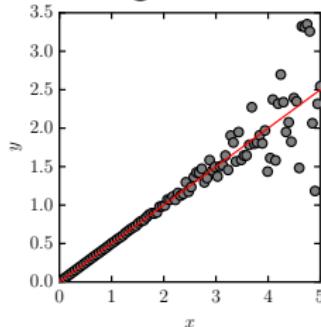
Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i
2. No autocorrelation.
3. Homoscedasticity: constant variation in error

$$\text{Var}(\epsilon_i) = \sigma_i^2 = \sigma^2, \forall i$$

- This could strongly affect our ability to predict. Our error is not constant throughout the data set.



Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i
2. No autocorrelation.
3. Homoscedasticity
4. Normally Distributed Errors: $\epsilon_i \sim N(0, \sigma)$
 - **Think of outliers.** We are minimizing the sum of squares, if one 'square' (ϵ) is very large, then we may not make sensible predictions with our model.

Assumptions for regression

Regression of this kind assumes a few things:

1. linearity: y is a linear function of all z_i
2. No autocorrelation.
3. Homoscedasticity
4. Normally Distributed Errors: $\epsilon_i \sim N(0, \sigma)$

These assumptions are very important!

Goodness of fit

Our linear regression minimises our ‘cost’ function $S = \sum_{i=1}^N \epsilon_i^2$. But how good is our model? One answer is contained in the R^2 value.

Goodness of fit

Our linear regression minimises our ‘cost’ function $S = \sum_{i=1}^N \epsilon_i^2$. But how good is our model? One answer is contained in the R^2 value.

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - \overbrace{ax_i - b}^{\hat{y}_i})^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$

Goodness of fit

Our linear regression minimises our ‘cost’ function $S = \sum_{i=1}^N \epsilon_i^2$. But how good is our model? One answer is contained in the R^2 value.

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - \overbrace{ax_i - b}^{\hat{y}_i})^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$
$$= 1 - \frac{\text{SSE}}{\text{SST}}$$

- Numerator is the **residual sum of squares**
- Denominator is the **total sum of squares**

Goodness of fit

Our linear regression minimises our ‘cost’ function $S = \sum_{i=1}^N \epsilon_i^2$. But how good is our model? One answer is contained in the R^2 value.

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - \overbrace{ax_i - b}^{\hat{y}_i})^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$
$$= 1 - \frac{\text{SSE}}{\text{SST}}$$

- Numerator is the **residual sum of squares**
- Denominator is the **total sum of squares**
- Therefore R^2 is the ratio of variance explained by the model over the total variation in observed values.

Goodness of fit

- R^2 is known as the coefficient of determination

Goodness of fit

- R^2 is known as the coefficient of determination
- $R^2 = r_{y,\hat{y}}^2$, which is called the squared Pearson correlation coefficient. This is defined:

$$r_{y,\hat{y}}^2 = \frac{\text{Cov}(y, \hat{y})}{\text{Var}(y)\text{Var}(\hat{y})}$$

Goodness of fit

- R^2 is known as the coefficient of determination
- $R^2 = r_{y,\hat{y}}^2$, which is called the squared Pearson correlation coefficient. This is defined:

$$r_{y,\hat{y}}^2 = \frac{\text{Cov}(y, \hat{y})}{\text{Var}(y)\text{Var}(\hat{y})}$$

Cov: Co-variance, Var: standard deviation

Takes values from -1 to 1:

- 1: total positive linear correlation
- 0: no linear correlation
- 1: total negative correlation

Goodness of fit

- R^2 is known as the coefficient of determination
- $R^2 = r_{y,\hat{y}}^2$, which is called the squared Pearson correlation coefficient. This is defined:

$$r_{y,\hat{y}}^2 = \frac{\text{Cov}(y, \hat{y})}{\text{Var}(y)\text{Var}(\hat{y})}$$

Proof of this is for a statistics, not a programming course!

Just be aware of the relationship, as you will undoubtedly hear of Pearson's correlation coefficient later in life.

Goodness of fit

- R^2 is known as the coefficient of determination

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - ax_i - b)^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$
$$= 1 - \frac{\text{SSE}}{\text{SST}}$$

- $R^2 = 1$, **all** variation in y_i is accounted for by the model

Goodness of fit

- R^2 is known as the coefficient of determination

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - ax_i - b)^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$
$$= 1 - \frac{\text{SSE}}{\text{SST}}$$

- $R^2 = 1$, **all** variation in y_i is accounted for by the model
- $R^2 = 0$, **no** variation in y_i is accounted for by the model.

Goodness of fit

- R^2 is known as the coefficient of determination

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - ax_i - b)^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$
$$= 1 - \frac{\text{SSE}}{\text{SST}}$$

- $R^2 = 1$, **all** variation in y_i is accounted for by the model
- $R^2 = 0$, **no** variation in y_i is accounted for by the model.
- $R^2 < 0$, model **worse** than using the sample mean as our model $\hat{y} = \bar{y}$.

Goodness of fit

R^2 can be used for other regression types:

Goodness of fit

R^2 can be used for other regression types:

$$R^2 = 1 - \frac{\sum_{i=1}^N (y_i - \hat{y}_i)^2}{\sum_{i=1}^N (y_i - \bar{y})^2}$$

where \hat{y} is a model we choose: polynomial / multiple / a combination of the two.

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

As a **rule of thumb**, 20 data points per predictor variable are required.

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

To understand this, consider if $N = n + 1$. We then have 1 coefficient a_i for each data point! In essence we completely overfit the data.

- $N = 2$, I can fit a straight line through two points (with a and b)

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

To understand this, consider if $N = n + 1$. We then have 1 coefficient a_i for each data point! In essence we completely overfit the data.

- $N = 2$, I can fit a straight line through two points (with a and b)
- $N = 3$, I can fit a plane through three points (with a_0, a_1, a_2)

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

To understand this, consider if $N = n + 1$. We then have 1 coefficient a_i for each data point! In essence we completely overfit the data.

- $N = 2$, I can fit a straight line through two points (with a and b)
- $N = 3$, I can fit a plane through three points (with a_0, a_1, a_2)
- $N = 4$, I can fit a hyperplane through four data points (with a_0, a_1, a_2, a_3)

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

To understand this, consider if $N = n + 1$. We then have 1 coefficient a_i for each data point! In essence we completely overfit the data.

- $N = 2$, I can fit a straight line through two points (with a and b)
- $N = 3$, I can fit a plane through three points (with a_0, a_1, a_2)
- $N = 4$, I can fit a hyperplane through four data points (with a_0, a_1, a_2, a_3)
- turtles all the way down...

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

To account for this, we can use the *adjusted R^2* ,

$$\bar{R}^2 = R^2 - (1 - R^2) \frac{n}{N - n - 1} \quad (2)$$

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

To account for this, we can use the *adjusted* R^2 ,

$$\overline{R}^2 = R^2 - (1 - R^2) \frac{n}{N - n - 1} \quad (2)$$

This penalises models that use more variables. As $n \rightarrow N - 1$,

$$\overline{R}^2 \rightarrow -\infty.$$

Adjusted R^2

R^2 is sensitive to the number of variables n / number of data points N . R^2 can **wrongly** increase if you increase the number of variables.

To account for this, we can use the *adjusted* R^2 ,

$$\boxed{\bar{R}^2 = R^2 - (1 - R^2) \frac{n}{N - n - 1}} \quad (2)$$

This penalises models that use more variables. As $n \rightarrow N - 1$,

$$\bar{R}^2 \rightarrow -\infty.$$

And, as $N/n \rightarrow \infty$,

$$\bar{R}^2 \rightarrow R^2,$$

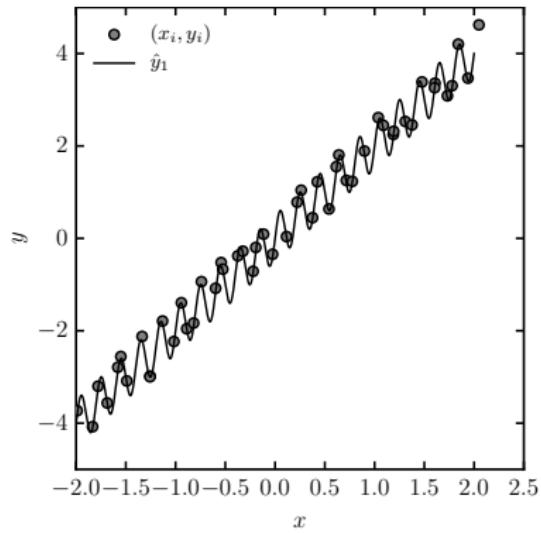
so we recover the original coefficient.

Limitations of R^2

A high R^2 isn't always better:

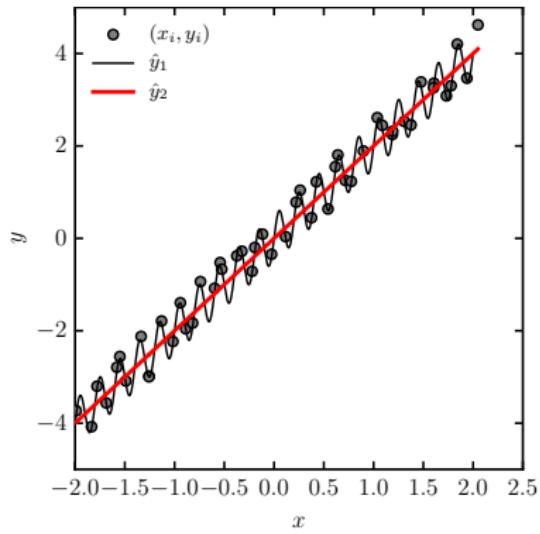
Limitations of R^2

A high R^2 isn't always better:



Limitations of R^2

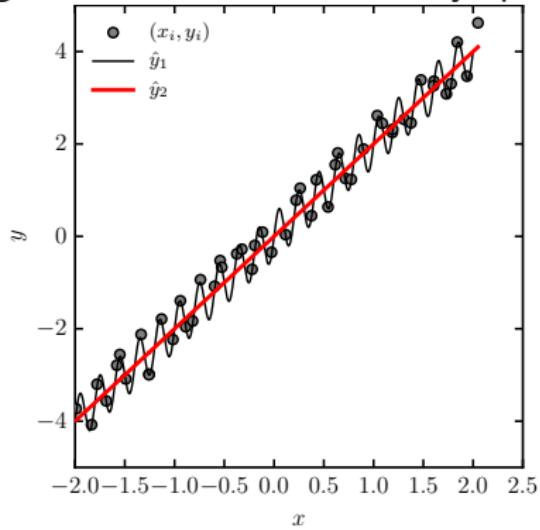
A high R^2 isn't always better:



Limitations of R^2

A high R^2 isn't always better:

- Model 1 has higher R^2 , common sense says pick **Model 2**



Limitations of R^2

A high R^2 isn't always better:

- A low R^2 isn't always bad. If you can show statistical significance of your coefficients, then you can still say my model explains X amount of variance (the interpretation of R^2).

Limitations of R^2

A high R^2 isn't always better:

- A low R^2 isn't always bad. If you can show statistical significance of your coefficients, then you can still say my model explains X amount of variance (the interpretation of R^2).
- Some subjects such as psychology/medicine naturally have more (random) variation.

Assumptions for regression

Recall: Regression of this kind assumes a few things:

1. linearity: y is a linear function of all predictor variables
2. No autocorrelation (for time series data).
3. Homoscedasticity: constant variation in error
4. Normally Distributed Errors: $\epsilon_i \sim N(0, \sigma)$

Breaking assumptions: **linearity**

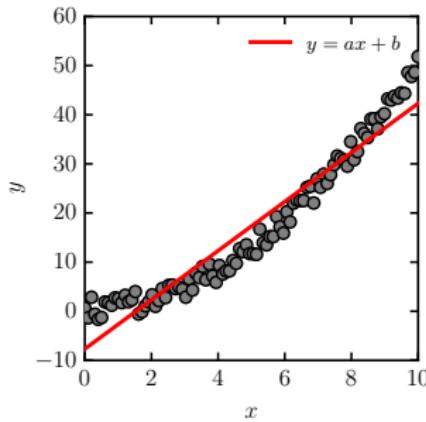
linearity: y is a linear function of all x_i (holding the others fixed)

$$\hat{y} = ax + b, \quad y = \hat{y} + \epsilon$$

- Detect by plotting y_i vs. \hat{y}_i :
 - Points should be roughly symmetric about diagonal line
- and/or ϵ_i vs. \hat{y}_i .
 - Points should be roughly symmetric about horizontal line.
- In these plots look for a pattern in the data. e.g. a curve in the data points.

Breaking assumptions: linearity

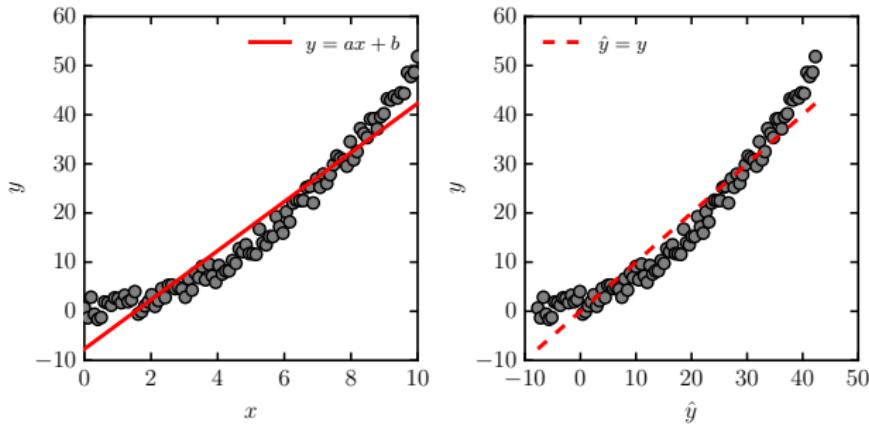
For example, suppose we fit a model through the data:



Breaking assumptions: linearity

For example, suppose we fit a model through the data:

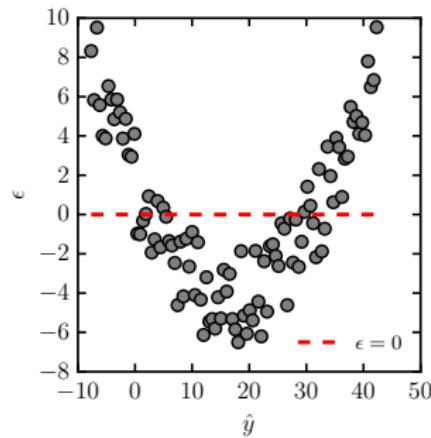
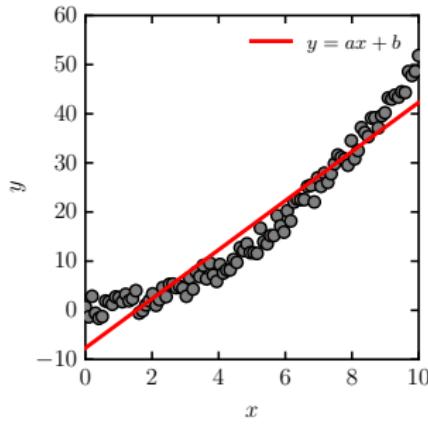
1. check linearity via: y_i vs \hat{y}_i



Breaking assumptions: linearity

For example, suppose we fit a model through the data:

1. check linearity via: y_i vs \hat{y}_i
2. check linearity via: ϵ_i vs \hat{y}_i



Breaking assumptions: **linearity**

linearity: y is a linear function of all x_i (holding the others fixed)

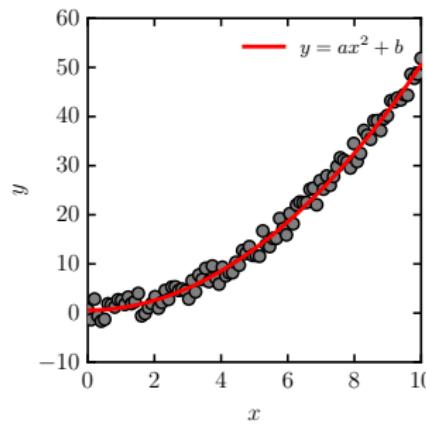
$$\hat{y} = ax + b, \quad y = \hat{y} + \epsilon$$

- Detect by plotting y_i vs. \hat{y}_i :
 - Points should be roughly symmetric about diagonal line
- and/or ϵ_i vs. \hat{y}_i .
 - Points should be roughly symmetric about horizontal line.
- FIX The easiest way is through variable transforms

Breaking assumptions: linearity

Let's fit a model:

$$\hat{y} = az + b, \quad z = x^2$$

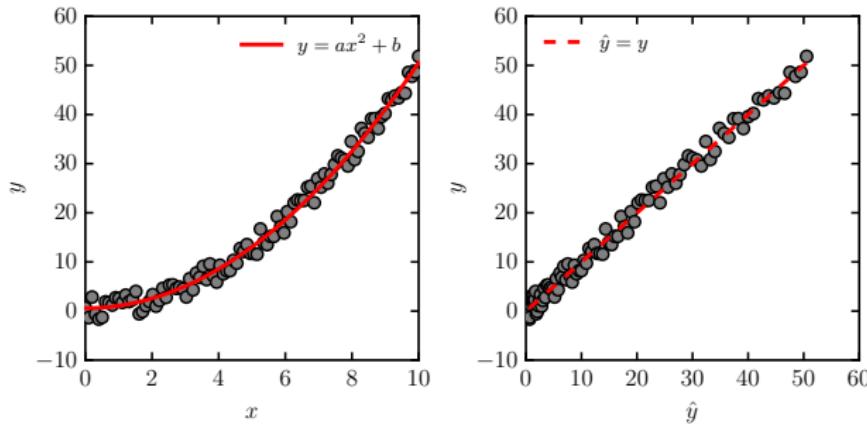


Breaking assumptions: linearity

Let's fit a model:

$$\hat{y} = az + b, \quad z = x^2$$

1. check linearity via: y_i vs \hat{y}_i , well distributed diagonally.

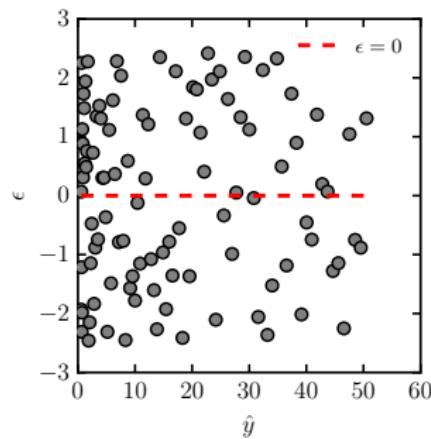
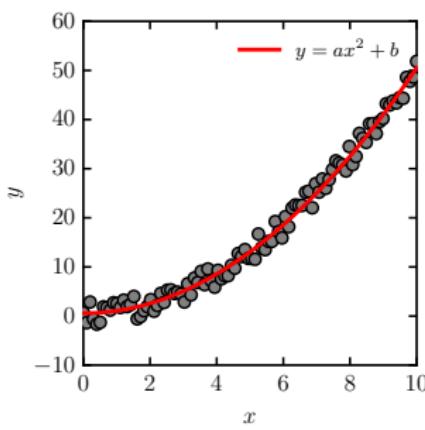


Breaking assumptions: linearity

Let's fit a model:

$$\hat{y} = az + b, \quad z = x^2$$

1. check linearity via: y_i vs \hat{y}_i , well distributed diagonally.
2. check linearity via: ϵ_i vs \hat{y}_i , well distributed horizontally.



Breaking assumptions: **autocorrelation**

There should be **no** autocorrelation of error. I.e:

$$A(\tau) = \sum_{i=1}^N \frac{(\epsilon_i - \bar{\epsilon})(\epsilon_{i+\tau} - \bar{\epsilon})}{\sigma_\epsilon} \approx 0, \quad \forall \tau \in \mathbb{N}$$

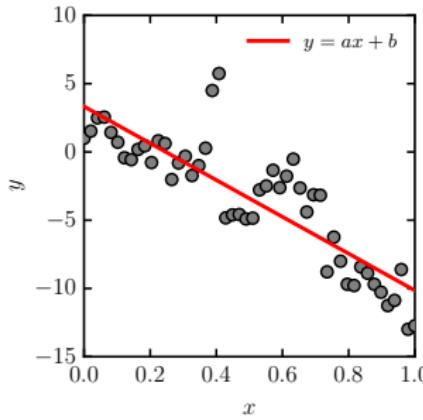
- Detect by plotting $A(\tau)$ vs. τ :

Breaking assumptions: autocorrelation

There should be **no** autocorrelation of error. I.e:

$$A(\epsilon) = \sum_{i=1}^N \frac{(\epsilon_i - \bar{\epsilon})(\epsilon_{i+\tau} - \bar{\epsilon})}{\sigma_\epsilon} \approx 0, \quad \forall \tau \in \mathbb{N}$$

- Detect by plotting $A(\tau)$ vs. τ :

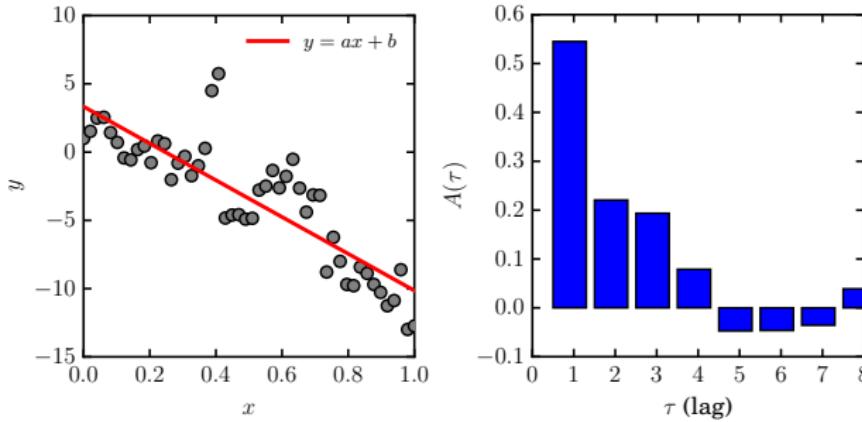


Breaking assumptions: autocorrelation

There should be **no** autocorrelation of error. I.e:

$$A(\epsilon) = \sum_{i=1}^N \frac{(\epsilon_i - \bar{\epsilon})(\epsilon_{i+\tau} - \bar{\epsilon})}{\sigma_\epsilon} \approx 0, \quad \forall \tau \in \mathbb{N}$$

- Detect by plotting $A(\tau)$ vs. τ :



Breaking assumptions: autocorrelation

There should be **no** autocorrelation of error. I.e:

$$A(\epsilon) = \sum_{i=1}^N \frac{(\epsilon_i - \bar{\epsilon})(\epsilon_{i+\tau} - \bar{\epsilon})}{\sigma_\epsilon} \approx 0, \quad \forall \tau \in \mathbb{N}$$

- Detect by plotting $A(\tau)$ vs. τ :
- FIX If $0.2 < A(\tau = j) < 0.5$, then we can add a lag (less than 0.2 is 'OK'):

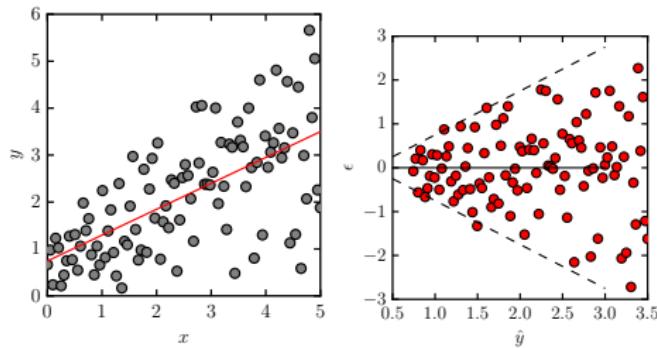
$$\hat{y}_i = ax + b + cx_{i-j}$$

- x_{i-j} is the 'lag', it is a delay of x by j observations. Note, your degrees of freedom reduces by j . Which may affect R^2 .
- For example, you may see a lag every 3 or 12 for quarterly or monthly data.

Breaking assumptions: homoscedasticity

The variance of our model error ϵ should be constant for all predictor variable values.

- Detect by plotting ϵ_i vs \hat{y}
- most common violation is when residuals grow over time:

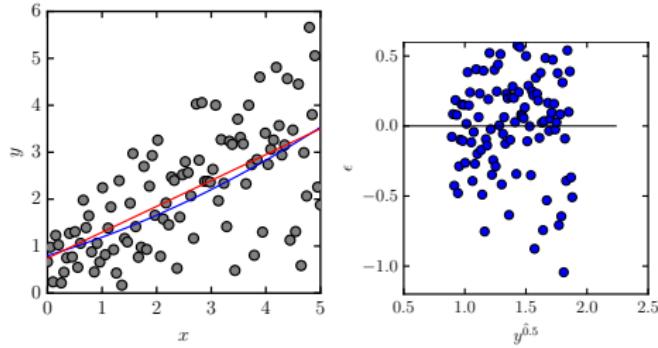


Breaking assumptions: homoscedasticity

The variance of our model error ϵ should be constant for all predictor variable values.

- Detect by plotting ϵ_i vs \hat{y}
- FIX take a log or $\sqrt{\cdot}$ of the predictor variable: for example:

$$\sqrt{y} = y^* = ax_i + b$$



Breaking assumptions: **normality**

Our model errors ϵ_i must come from a normal distribution. I.e. the probability of observing an error $\epsilon_i = \zeta$ is:

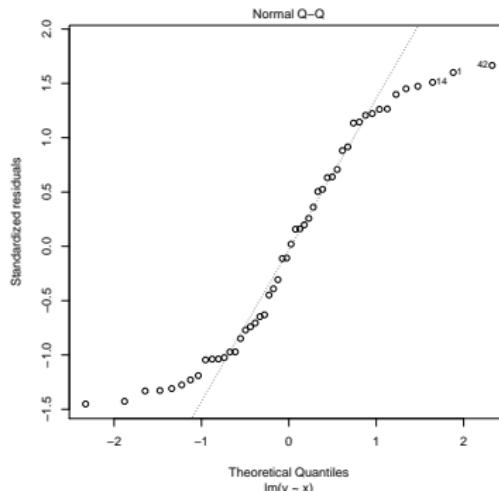
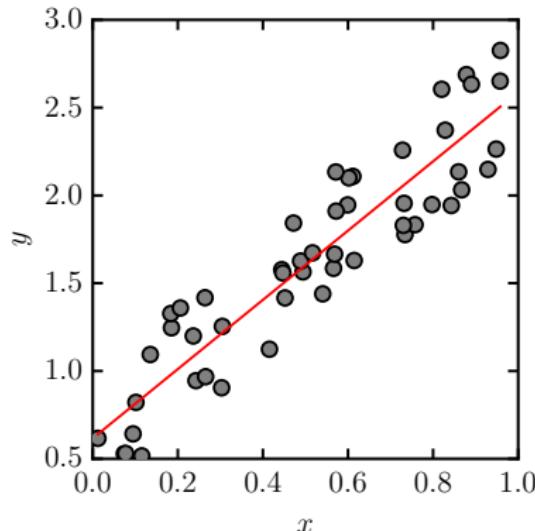
$$P(\epsilon_i = \zeta) = \frac{1}{\sqrt{2\pi\sigma_\epsilon^2}} \exp\left(-\frac{\zeta^2}{2\sigma_\epsilon^2}\right)$$

- **Detect by** plotting the quantiles of standardised residuals against theoretical residuals:
- i.e. calculate the quantiles of $\frac{\epsilon_i}{\sigma_\epsilon}$ and plot against the quantiles of $N(0, 1)$
- P.S quantiles are equal divisions of a frequency distribution. Each contains the same fraction of the total population.

Breaking assumptions: normality

Our model errors ϵ_i must come from a normal distribution.

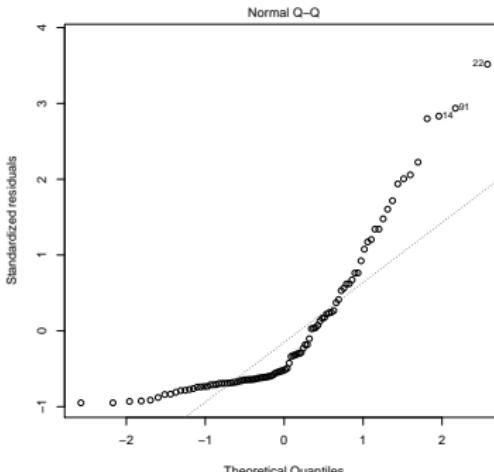
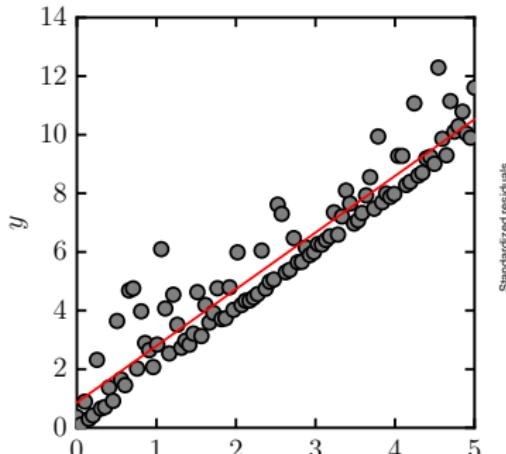
- Detect by plotting the quantiles of standardised residuals against theoretical residuals.
- This kind of plot is a Q-Q plot.
- Example of minor violation at tails of distribution:



Breaking assumptions: normality

Our model errors ϵ_i must come from a normal distribution.

- Detect by plotting the quantiles of standardised residuals against theoretical residuals.
- This kind of plot is a Q-Q plot.
- Example of bad violation of normality. The residuals do not resemble a normal distribution (the straight line):



Breaking assumptions: **normality**

Our model errors ϵ_i must come from a normal distribution.

- **Detect by** plotting the quantiles of standardised residuals against theoretical residuals.
- Examples of minor and bad violation of normality. ‘outliers’ are affecting the fit
- **FIX** often we violate normality because we violate linearity. Therefore transformations of the variables may help.
- **FIX** Another possibility is to get more data (increase N). This is because we may be able to appeal to the **central limit theorem** (that given enough independent random variables, the mean of these variables approaches a normal distribution).



THE UNIVERSITY OF

MELBOURNE