Numerical Methods for Chemical Engineers

Study guide for 6E5X0, 2018-2019

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Numerical Methods

"Simulation and mathematical modeling will power the twenty-first century the way steam powered the nineteenth."

— W.H. Press*

Ptolemy and the almagest

~150 AD. Development of numerical approximations to describe the mo-

Numerical Methods

- Numerical analysis is concerned with obtaining approximate solutions to problems while maintaining reasonable bounds of error...
- ...because it is often impossible to obtain exact answers ...
- Numerical analysis makes use of algorithms to approximate solutions

Relevance

- Important to the world!
- E.g. in astronomy, construction, agriculture, architecture, ...
- And of course in Engineering!

...Chemical Engineering...

- Description of reactors and separators (dynamic and steady state)
- Computational fluid dynamics
- Thermodynamic equations of state
- Optimizing process performance
- Design and synthesis of processes
- Regression of data, e.g. isotherms, kinetics, ...

Course Schedule

Lecture	Date	Topic	Teacher
1	12/11/2017	Programming and algorithms (1+2)	IR
2	15/11/2017	Numerical errors	MSA
3	19/11/2017	Linear eqns: direct methods	IR
4	22/11/2017	Linear eqns: iterative methods	IR
5	26/11/2017	Non-linear equations (1)	MSA
6	29/11/2017	Non-linear equations (2)	MSA
7	03/12/2017	No lecture	
8	06/12/2017	No lecture	
9	10/12/2017	Interpolation + integration	IR
10	13/12/2017	ODEs (1)	MSA
11	17/12/2017	ODEs (2)	MSA
12	20/12/2017	PDEs	MSA
13	07/01/2018	Q&A session	IR
14	10/01/2018	Regression and optimization	IR

Course Objectives

- Gain experience with programming basics and algorithm design
- Acquire knowledge of and experience with different techniques for the numerical solution of systems of linear and non-linear algebraic and differential equations, as well as data analysis and optimization.
- Being able to solve various numerical problems using Matlab or Excel.

Prerequisites

The following courses should give you enough background knowledge to follow this course comfortably:

- Calculus
- Linear Algebra
- Some basic MATLAB experience
  - We will shortly cover some aspects on MATLAB programming in the first lectures. Detailed documents and courses are provided on Canvas, for your own reference.

You will definitely need a laptop with Matlab and Excel installed!

Course Materials

- Lecture slides
- MATLAB scripts
- Additional articles
- There are some useful books:
  - Numerical recipes, W.H. Press et al.
  - Numerical methods for chemical engineering, K.J. Beers
  - Numerical methods for chemical engineers, A. Constantinides
  - Essential matlab-for engineers ,B.D. Hahn
  - Introduction to Numerical Methods and Matlab Programming for Engineers, T. Young and M.J. Mohlenkamp

Look on Canvas for the slides, exercises, scripts, assignments and additional documentation on MATLAB.

Assessment

5 assignments

- Each 20% of the final result
- Done in groups of 2 persons
  - Form groups via Canvas
  - Make sure that you have similar intentions!
- Short report (template provided, Canvas)

Assessment

5 assignments

- Each 20% of the final result
- Done in groups of 2 persons
  - Form groups via Canvas
  - Make sure that you have similar intentions!
- Short report (template provided, Canvas)

About the 5th assignment

- Short assignment + oral exam (in groups)
- Oral exam covers *all assignments and topics*
- Individual knowledge is assessed
- Grade needs to be at least a 5.0

Assignment grading

We will use rubrics to grade your reports. The following categories will be looked at.

Assignment grading

We will use rubrics to grade your reports. The following categories will be looked at.

- Use of numerical methods: e.g. built-in solvers vs. show implementation numerical methods

Assignment grading

We will use rubrics to grade your reports. The following categories will be looked at.

- Use of numerical methods: e.g. built-in solvers vs. show implementation numerical methods
- Analysis of results: just the number is provided vs. high detail analysis and interpretation

Assignment grading

We will use rubrics to grade your reports. The following categories will be looked at.

- Use of numerical methods: e.g. built-in solvers vs. show implementation numerical methods
- Analysis of results: just the number is provided vs. high detail analysis and interpretation
- Programming skills: unstructured code, difficult to change vs. readable code with comments and UI

Assignment grading

We will use rubrics to grade your reports. The following categories will be looked at.

- Use of numerical methods: e.g. built-in solvers vs. show implementation numerical methods
- Analysis of results: just the number is provided vs. high detail analysis and interpretation
- Programming skills: unstructured code, difficult to change vs. readable code with comments and UI
- Visualisation: unreadable graphs with no axes labels or legend vs. publication quality graphs, consistency between datasets

Assignment handout and deadlines

Hand-in your assignments via Canvas

- Deadlines are given on Canvas as well
- Deliver the report in PDF format
- Send along the scripts + necessities in a .zip

When delivering your final assignment, suggest a timeslot for the oral exam (e.g. via Canvas/assignment comment section or as Canvas message).

Contact information

Ivo Roghair

- E-mail: i.roghair@tue.nl
- Office: STW 0.37

Martin van Sint Annaland

- E-mail: m.v.sintannaland@tue.nl
- Office: STW 0.39

Dario Balice, Francesco Sabatino, Jasper van Kampen

For help with the exercises, Dario, Francesco and Jasper will help out during the lectures.

Some last remarks

- Tell us if something is not clear.
- We try to make the lectures interactive, working on examples and creating scripts as we go. It is advised that you work along with us to get the most out of this course!
- The exercises are meant to provide a jump start towards the assignments.
- We will always answer questions on the exercises. We may didactically answer questions on the assignments.
- During the lectures/tutorials we first and foremost work on the exercises. If they are done, you can work on the assignment if you want.

Some Acknowledgements

Some Real Acknowledgements

- To Roel Verstappen of Groningen University
- To Johan Hult of Cambridge University
- To Edwin Zondervan, now at Universität Bremen

Matlab and Programming 1

Programming basics and algorithms

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions

Why should you learn something about programming?

- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods

Why should you learn something about programming?

- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks

Why should you learn something about programming?

- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code

Why should you learn something about programming?

- Scientific techniques depend in an increasing fashion upon computer programs and simulation methods
- Knowledge of programming allows you to automate routine tasks
- Ability to understand algorithms by inspection of the code
- Learn to think by dissecting a problem into smaller bits

Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

```
>> 2 + 3          % Some simple calculations
>> 2*3
>> 2*3^2          % Powers are done with ^
```

Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

```
>> 2 + 3          % Some simple calculations
>> 2*3
>> 2*3^2          % Powers are done with ^
>> a = 2          % Storing values into the workspace
>> b = 3
>> c = (2*3)^2    % Parentheses set priority
>> 8/a-b
```

Getting started

Start Matlab, and enter the following commands on the command line. Evaluate the output.

```
>> 2 + 3          % Some simple calculations
>> 2*3
>> 2*3^2          % Powers are done with ^
>> a = 2          % Storing values into the workspace
>> b = 3
>> c = (2*3)^2    % Parentheses set priority
>> 8/a-b
>> sin(a)         % Mathematical functions can be used
>> sin(0.5*pi)    % pi is an internal Matlab variable
>> 1/0            % Infinity is a thing ...
>> sqrt(-1)       % ... as are imaginary numbers
```

Introduction to programming

What is a program?

A program is a sequence of instructions that is written to perform a certain task on a computer.

- The computation might be something mathematical, a symbolic operation, image analysis, etc.

Program layout

1. Input (Get the radius of a circle)
2. Operations (Compute and store the area of the circle)
3. Output (Print the area to the screen)

Versatility of Matlab

Versatility of Matlab: ODE solver

Editor - /mnt/data/BittorrentSync/Werk/Documents/Onderwijs/Pra...

solveTankSeries.m | tankSeries.m

```
p.F0 = 1.0;
p.K = 0.8;
p.N = 25;
p.A = 1.0;

init = zeros(p.N,1);
time = [0 25];

[t,h] = ode45(@tankSeries,time,init,[],p);

plot(t,h);
xlabel('Time [s]');
ylabel('Height [m]');
```

Figures - Figure 1

Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```

Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```

Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```

Versatility of Matlab: Image analysis

```
I = imread('bubbles.png');
BW = rgb2gray(I);
E = edge(BW, 'canny');
F = imfill(E, 'holes');
result = regionprops(F);
```

Versatility of Matlab: Curve fitting

Matlab help

- Matlab documentation: `doc` or `help` function

Matlab help

- Matlab documentation: `doc` or `help` function
- Canvas page

Matlab help

- Matlab documentation: `doc` or `help` function
- Canvas page
- Introduction to Numerical Methods and Matlab Programming for Engineers. T. Young and M.J. Mohlenkamp (2015). GNU-licensed document, online

Matlab help

- Matlab documentation: `doc` or `help` function
- Canvas page
- Introduction to Numerical Methods and Matlab Programming for Engineers. T. Young and M.J. Mohlenkamp (2015). GNU-licensed document, online
- Search the web!

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions

Terminology

Variable Piece of data stored in the computer memory, to be referenced and/or manipulated

Function Piece of code that performs a certain operation/sequence of operations on given input

Operators Mathematical operators (e.g. `+` `-` `*` or `/`), relational (e.g. `<` `>`or `==`, and logical operators (`&&`, `||`)

Script Piece of code that performs a certain sequence of operations without specified input/output

Expression A command that combines variables, functions, operators and/or values to produce a result.

Variables in Matlab

- Matlab stores variables in the *workspace*

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Matlab also defines a number of variables by default, e.g. `eps`, `pi` or `i`.

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Matlab also defines a number of variables by default, e.g. `eps`, `pi` or `i`.
- You can assign a variable by the `=` sign:

```
>> x = 4*3
x =
    12
```

Variables in Matlab

- Matlab stores variables in the *workspace*
- You should recognize the difference between the *identifier* of a variable (its name, e.g. `x`, `setpoint_p`), and the data that it actually stores (e.g. 0.5)
- Matlab also defines a number of variables by default, e.g. `eps`, `pi` or `i`.
- You can assign a variable by the `=` sign:

```
>> x = 4*3
x =
    12
```

- If you don't assign a variable, it will be stored in `ans`
- Clearing the workspace is done with `clear`.

Datatypes and variables

Matlab uses different types of variables:

Datatype	Example
`string`	`'Wednesday'`
`integer`	`15`
`float`	`0.15`
`vector`	`[0.0; 0.1; 0.2]`
`matrix`	`[0.0 0.1 0.2; 0.3 0.4 0.5]`
`struct`	`sct.name = 'MyDataName'`
	`sct.number = 13`
`logical`	`0` (false)
	`1` (true)

About variables

- Matlab variables can change their type as the program proceeds (this is not common for other programming languages!):

```
>> s = 'This is a string'
s =
This is a string
>> s = 10
s =
    10
```

- Vectors and matrices are essentially *arrays* of another data type. A vector of `struct` is therefore possible.
- Variables are *local* to a function (more on this later).

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Manipulate the value of that entry:

```
>> u(2)=47
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Manipulate the value of that entry:

```
>> u(2)=47
```

Get a slice of a vector:

```
>> u([2 3 4]) % With colon operator: u(2:4)
```

Vectors in Matlab (1)

A row vector:

```
>> v = [0 1 2 3]
```

A column vector by separating elements with semi-colons:

```
>> u = [9; 10; 11; 12; 13; 14; 15]
```

Access (i.e. read) an entry in a vector:

```
>> u(2)
```

Manipulate the value of that entry:

```
>> u(2)=47
```

Get a slice of a vector:

```
>> u([2 3 4]) % With colon operator: u(2:4)
```

Transposing vectors:

```
>> w = v'
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10        % Default stride is 1
>> x = -1:.1:1     % start:stride:stop specifies list
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10        % Default stride is 1
>> x = -1:.1:1     % start:stride:stop specifies list
```

Or, when you prefer to set the *number of elements* instead of the step size:

```
>> y = linspace(0,10,11)
>> p = logspace(2,6,5)
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10        % Default stride is 1
>> x = -1:.1:1     % start:stride:stop specifies list
```

Or, when you prefer to set the *number of elements* instead of the step size:

```
>> y = linspace(0,10,11)
>> p = logspace(2,6,5)
```

Manipulating multiple components:

```
>> y([1 4:7]) = 1
```

Vectors in Matlab (2)

Manual definition may be cumbersome. A colon (:) generates a list:

```
>> a = 1:10        % Default stride is 1
>> x = -1:.1:1     % start:stride:stop specifies list
```

Or, when you prefer to set the *number of elements* instead of the step size:

```
>> y = linspace(0,10,11)
>> p = logspace(2,6,5)
```

Manipulating multiple components:

```
>> y([1 4:7]) = 1
```

Or (by supplying a vector instead of a scalar):

```
>> y([1 4:7]) = 16:20 % equivalent to y([1 4 5 6 7]) = [16 17 18 19 20]
```

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Find a way to define the vector without typing all individual elements

Practice

Given a vector

$$x = [2\ 4\ 6\ 8\ 10\ 12\ 14\ 16\ 18\ 20\ 30\ 40\ 50\ 60\ 70\ 80]$$

- Find a way to define the vector without typing all individual elements
- Investigate the meaning of the following commands:

```
>> y = x(5:end)
>> y(4)
>> y(4) = []
>> sum(x)
>> mean(x)
>> std(x)
>> max(x)
>> fliplr(x)
>> diff(x)
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
>> h = e^2
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
>> h = e^2
```

... wait ... what's that?

```
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.
```

Operations on vectors (1)

```
>> e = 1:5
>> f = 2*e
>> g = 4*f + 20
>> h = e^2
```

... wait ... what's that?

```
Error using  ^
Inputs must be a scalar and a square matrix.
To compute elementwise POWER, use POWER (.^) instead.
```

Matlab uses matrix operations by default, we should use a dot operator to make operations element-wise for `*`, `/` and `^`.

```
>> e.^2
```

Operations on vectors (2)

To demonstrate the matrix product:

```
>> p = [1; 1; 1]
>> q = [1 2 3]
>> p*q   % which is not equal to q*p
```

Operations on vectors (2)

To demonstrate the matrix product:

```
>> p = [1; 1; 1]
>> q = [1 2 3]
>> p*q   % which is not equal to q*p
```

All kinds of mathematical functions on vectors typically operate on elements:

```
>> x = linspace(0,2*pi,100);
>> s = sin(x)
>> e = exp(x)
```

Building blocks: Mathematics and number manipulation

Programming languages usually support the use of various mathematical functions (sometimes via a specialized library). Some examples of the most elementary functions in Matlab:

Command	Explanation
`cos(x), sin(x), tan(x)`	Cosine, sine or tangens of x
`mean(x), std(x)`	Mean, st. deviation of vector x
`exp(x)`	Value of the exponential function e^x
`log10(x), log(x)`	Base-10/Natural logarithm of x
`floor(x)`	Largest integer smaller than x
`ceil(x)`	Smallest integer that exceeds x
`abs(x)`	Absolute value of x
`size(x)`	Size of a vector x
`length(x)`	Number of elements in a vector x
`rem(x,y)`	Remainder of division of x by y

Printing results

You can prevent displaying the outcome of a command by adding a semi-colon at the end of a line:

```
>> c = linspace(0,10,11);
>> length(c)
>> c
>> size(c)
```

Printing results

You can prevent displaying the outcome of a command by adding a semi-colon at the end of a line:

```
>> c = linspace(0,10,11);
>> length(c)
>> c
>> size(c)
```

Altering the display format can be done using the `format` command:

```
>> format compact % loose
>> format long    % short
```

Simple plotting

Make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

Simple plotting

Make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu)
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Simple plotting

Make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu,'*')
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Simple plotting

Make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu,'r--')
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Simple plotting

Make a plot of the following table

T (°C)	5	20	30	50	55
μ (Pa·s)	0.08	0.015	0.009	0.006	0.0055

```
>> T = [ 5 20 30 50 55 ]
>> mu = [ 0.08 0.015 0.009 0.006 0.0055]
```

```
>> plot(T,mu,'ko-','LineWidth',2)
```

```
>> xlabel('Temperature [^\circC]')
>> ylabel('Viscosity [Pa s]')
>> title('Experiment 1')
```

Practice

Create plots of the following functions in a single figure for $x \in \{0, 2\pi\}$:

$$y_1 = \cos x$$

$$y_2 = \arctan x$$

$$y_3 = \frac{\sin x}{x}$$

Practice

Create plots of the following functions in a single figure for $x \in \{0, 2\pi\}$:

$$y_1 = \cos x$$

$$y_2 = \arctan x$$

$$y_3 = \frac{\sin x}{x}$$

Strategies to draw multiple graphs in 1 figure:

```
>> plot(x,y1,x,y2,x,y3)
```

```
>> plot(x,y1)
>> hold on; % Maintain drawn plots in current figure
>> plot(x,y2)
>> plot(x,y3) % The 'hold-property' was already set
```

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

In Matlab:

```
>> A = [ 8 1 6; 3 5 7; 4 9 2]
```

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

In Matlab:

```
>> A = [ 8 1 6; 3 5 7; 4 9 2]
```

Elements can be accessed/manipulated by the following syntax:

```
>> A(3,1) % Third row, first column, also A(3)
>> A(3,:) = [2 4 8] % Set entire third row
>> A(:,3) % Print third column
>> A(A>5) = 2 % Set elements by condition
```

Matrices in Matlab

Matrix A is defined as:

$$A = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

In Matlab:

```
>> A = [ 8 1 6; 3 5 7; 4 9 2]
```

Elements can be accessed/manipulated by the following syntax:

```
>> A(3,1) % Third row, first column, also A(3)
>> A(3,:) = [2 4 8] % Set entire third row
>> A(:,3) % Print third column
>> A(A>5) = 2 % Set elements by condition
```

There are a few functions that help creating matrices:

```
>> A = zeros(4)  % A 4x4 matrix with zeros
>> A = ones(4,1) % A 4-element vector with ones
>> A = eye(3)    % Identity matrix of 3x3
>> A = rand(3,4) % A 3x4 matrix with random numbers
```

Practice

- Find a *short* Matlab expression to create the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 9 & 7 & 5 & 3 & 1 & -1 & -3 \\ 4 & 8 & 16 & 32 & 64 & 128 & 256 \end{bmatrix}$$

Practice

- Find a *short* Matlab expression to create the following matrix:

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 9 & 7 & 5 & 3 & 1 & -1 & -3 \\ 4 & 8 & 16 & 32 & 64 & 128 & 256 \end{bmatrix}$$

- Investigate the command `max(A)`. What does it give?

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met.

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met.

```
num = floor (10 * rand + 1);
guess = input ('Your guess please : ');
if ( guess ~= num )
  disp (['Wrong, it was ',num2str(num),'. Kbye.']);
else
    disp ('Correct !') ;
end
```

Building blocks: conditional statements

`if`-statement: Check whether a (set of) condition(s) is met.

```
num = floor (10 * rand + 1);
guess = input ('Your guess please : ');
if ( guess ~= num )
  disp (['Wrong, it was ',num2str(num),'. Kbye.']);
else
    disp ('Correct !') ;
end
```

Other relational operators

`==`	is equal to
`<=`	is less than or equal to
`>=`	is greater than or equal to
`<`	is less than
`>`	is greater than

Combining conditional statements

`&&`	and
`||`	or
`xor`	exclusive or

Building blocks: loops

`for`-loop: Performs a block of code a certain number of times.

Building blocks: loops

`for`-loop: Performs a block of code a certain number of times.

```
>> p(1) = 1;
>> p(2) = 1;
>> for i = 2:10
p(i+1) = p(i)+p(i-1);
end
>> p
p =
   1   1   2   3   5   8  13  21  34  55  89
```

Building blocks: indeterminate repetition

`while`-loop: Performs and repeats a block of code until a certain condition.

Building blocks: indeterminate repetition

`while`-loop: Performs and repeats a block of code until a certain condition.

```
num = floor (10* rand +1) ;
guess = input ('Your guess please : ');

while ( guess ~= num )
    guess = input ('That is wrong. Try again ... ');
end

if (isempty(guess))
    disp('No number supplied - exit');
else
    disp ('Correct!');
end
```

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Naive approach

```
Z = 1;
Z = Z*2;
Z = Z*3;
Z = Z*4;
... etc ...
```

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Naive approach

```
Z = 1;
Z = Z*2;
Z = Z*3;
Z = Z*4;
... etc ...
```

For-loop

```
Z = 1;
for i = 1:N
    Z = Z*i;
end
```

Example algorithm

Compute the factorial of N: $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

How to deal with this?

Naive approach

```
Z = 1;
Z = Z*2;
Z = Z*3;
Z = Z*4;
... etc ...
```

For-loop

```
Z = 1;
for i = 1:N
    Z = Z*i;
end
```

While-loop

```
Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end
```

Note: `N` must be set beforehand!
Note: Pay attention to the relational operators!

Building blocks: case selection

`switch`-statement: Selects and runs a block of code.

Building blocks: case selection

`switch`-statement: Selects and runs a block of code.

```
[dnum,dnam] = weekday(now);
switch dnum
    case {1,7}
        disp('Yay! It is weekend!');
    case 6
        disp('Hooray! It is Friday!');
    case {2,3,4,5}
        disp(['Today is ' dnam]);
    otherwise
        disp('Today is not a good day...');
end
```

Input and output

Many programs require some input to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")

Input and output

Many programs require some input to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")
- Input may be entered via the keyboard

```
>> a = input('Please enter the number ');
```

Input and output

Many programs require some input to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")
- Input may be entered via the keyboard

```
>> a = input('Please enter the number ');
```

- Input may be read from a file, e.g.

```
>> data = getfield(importdata('myData.txt', ' ', 4), 'data');
>> numdata = xlsread('myExcelDataFile.xls');
```

Input and output

Many programs require some input to function correctly. A combination of the following is common:

- Input may be given in a parameters file ("hard-coded")
- Input may be entered via the keyboard

```
>> a = input('Please enter the number ');
```

- Input may be read from a file, e.g.

```
>> data = getfield(importdata('myData.txt', ' ', 4), 'data');
>> numdata = xlsread('myExcelDataFile.xls');
```

- There are many more advanced functions, e.g. `fread`, `fgets`, ...

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, not .jpg or .pcx)

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, not .jpg or .pcx)
  - Save variables (right click, save as)

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, not .jpg or .pcx)
  - Save variables (right click, save as)
- Save variables automatically (scripted):

```
>> savefile = 'test.mat';
>> p = rand(1,10);
>> q = ones(10);
>> save(savefile,'p','q')
```

Input and output

Output of results to screen, storing arrays to a file or exporting a graphic are the most common ways of getting data out of Matlab:

- Results of each expression are automatically shown on screen as long as the line is not ended with a semi-colon;
- Output may be stored via the GUI:
  - Use the 'Export Setup' function
  - Save figure (use .fig, .eps or .png, not .jpg or .pcx)
  - Save variables (right click, save as)
- Save variables automatically (scripted):

```
>> savefile = 'test.mat';
>> p = rand(1,10);
>> q = ones(10);
>> save(savefile,'p','q')
```

- More advanced functions can be found in e.g. `fwrite`, `fprintf`, ...

Today's outline

- Introduction
- Data structures
- Plotting
- Creating algorithms
- Functions

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

Functions - general

A function in a programming language is a program fragment that performs a certain task. Creating functions keeps your code clean, re-usable and structured.

- You can use functions supplied by the programming language, and define functions yourself
- Functions take one or more input parameters (*arguments*), and *return* an output (result).
  - If functions do not return a result, it is called a procedure
- In Matlab, functions are defined as follows (2 output variables and 3 input arguments):

```
function [out1, out2] = myFunction(in1, in2, in3)
```

Functions - locality and arguments

- You are supplying arguments to a function because it does not have acces to previously defined variables. This is called *locality*.
  - This does not include global variables - but they're evil!
  - Local variables created in a function are not accessible to other functions unless they are returned or supplied as an argument!

Functions - locality and arguments

- You are supplying arguments to a function because it does not have acces to previously defined variables. This is called *locality*.
  - This does not include global variables - but they're evil!
  - Local variables created in a function are not accessible to other functions unless they are returned or supplied as an argument!

Example: write a function that takes 3 variables, and returns the average:

Functions - locality and arguments

- You are supplying arguments to a function because it does not have acces to previously defined variables. This is called *locality*.
  - This does not include global variables - but they're evil!
  - Local variables created in a function are not accessible to other functions unless they are returned or supplied as an argument!

Example: write a function that takes 3 variables, and returns the average:

Approach 1

```
function res = avg1(a,b,c)
    mySum = a + b + c;
    res = mySum / 3;
end
```

Approach 2

```
function res = avg2(a,b,c)
    data = [a; b; c];
    res = mean(data);
end
```

Exercise: create a function

Compute $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

Create a function of our while-loop approach with N the argument:

Original script

```
Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end
```

Exercise: create a function

Compute $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$

Create a function of our while-loop approach with N the argument:

Original script

```
Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end
```

Function

```
function Z = fact_while(N)

Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end

end
```

Functions - checking input

The function we created computes the factorial correctly!

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

- In this case, we should check the user input to prevent an infinite loop:

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

- In this case, we should check the user input to prevent an infinite loop:

Functions - checking input

The function we created computes the factorial correctly!

- When the supplied argument is positive and
- When the supplied argument is a natural number...

- In this case, we should check the user input to prevent an infinite loop:

```
if (fix(N)~=N) | (N<0)
    disp 'Provide a positive integer number!'
    return;
end
```

- If no check can be done before a while-loop, you may want to stop after x loops

Functions - checking input

The whole factorial function, including comments:

```
function Z = fact_while(N)
%% This function computes a factorial of input value N
% Usage   : fact_while(N)
% N       : value of which the factorial is computed
% returns: factorial of N

% Catch non-integer case
if (fix(N)~=N) | (N<0)
    disp 'Provide a positive integer number!'
    return;
end

Z = 1;
i = 1;
while (i<=N)
    Z = Z*i;
    i = i+1;
end

end
```

Recursion

- In order to understand recursion, one must first understand recursion

Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function is called by itself (a function within a function)

Recursion

- In order to understand recursion, one must first understand recursion
- A recursive function is called by itself (a function within a function)
  - This could lead to infinite calls;
  - A base case is required so that recursion is stopped;
  - Base case does not call itself, simply returns.

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

- What does this function do?

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

- What does this function do?
- Can you spot the error?

Recursion: example

```
function out = mystery(a,b)
if (b == 1)
    % Base case
    out = a;
else
    % Recursive function call
    out = a + mystery(a,b-1);
end
```

- What does this function do?
- Can you spot the error?
- How deep can you go? Which values of b don't work anymore?

Recursion: exercise

Create a function computing the factorial of N, based on recursion.

Recursion: exercise

Create a function computing the factorial of N, based on recursion.

```
function res = fact_recursive(x)

% Catch non-integer case
if (fix(x)~=x) | (x<0)
    disp 'You should provide a positive integer number only'
    return;
end

if (x > 1)
    res = x*fact_recursive(x-1);
else
    res = 1;
end

end
```

Matlab and Programming 2

Advanced programming techniques

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Coding in style
- Error management
- Visualisation
- Functions: the sequel
- Conclusions

Today's outline

- Coding in style
- Error management
- Visualisation
- Functions: the sequel
- Conclusions

If anything sticks today, let it be this

Your code will not be understood by anyone

If anything sticks today, let it be this

Your code will not be understood by anyone

That includes future-you

Code organization

- Optimization of a code is time-consuming and complicated
- The more you optimize your code, the less readable it becomes
- But... You can write it in a such way that it will be flexible and easy to maintain
- Especially important in team work
- Any person has its own handwriting. Any programmer has its own coding style.

The coding style ≡ handwriting...

Interpret the following code

```
s=checksc();
if(s==true)
a=cb();
b=cfrsp();
if(a<5)
if(b>5)
a=gtbs();
end
if(a>b)
ubx();
end
end
else
brn();
gtbs();
end
```

WAT

Let's change that a bit... Indentation

Let's change that a bit... Indentation

Shown here with 2 spaces of indentation, Matlab uses 4 by default!

```
s=checksc();
if(s==true)
a=cb();
b=cfrsp();
if(a<5)
if(b>5)
a=gtbs();
end
if(a>b)
ubx();
end
end
else
brn();
gtbs();
end
```

```
s = checksc();
if (s == true)
  a = cb();
  b = cfrsp();
  if (a < 5)
    if (b > 5)
      a = gtbs();
    end
    if (a > b)
      ubx();
    end
  end
else
  brn();
  gtbs();
end
```

Readable variables and function names

```
s = checksc();
if (s == true)
  a = cb();
  b = cfrsp();
  if (a < 5)
    if (b > 5)
      a = gtbs();
    end
    if (a > b)
      ubx();
    end
  end
else
    brn();
    gtbs();
end
```

```
IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < 5)
    if (shelfSize > 5)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

Get rid of obscure constants in the code

```
IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < 5)
    if (shelfSize > 5)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

```
IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < maxShelfSize)
    if (shelfSize > minBooksNeeded)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

That's more like it!

```
s=checksc();
if(s==true)
a=cb();
b=cfrsp();
if(a<5)
if(b>5)
a=gtbs();
end
if(a>b)
ubx();
end
end
else
brn();
gtbs();
end
```

```
IAmFree = checkSchedule();
if (IAmFree == true)
  books = countBooks();
  shelfSize = countFreeSpaceShelf();
  if (books < maxShelfSize)
    if (shelfSize > minBooksNeeded)
      books = goToBookStore();
    end
    if (books > shelfSize)
      useBox();
    end
  end
else
  burnBooks();
  goToBookStore();
end
```

Writing readable code

Good code reads like a book.

Writing readable code

Good code reads like a book.

- When it doesn't, make sure to use comments. In Matlab, everything following `% is a comment`
- Prevent "smart constructions" in the code
- Re-use working code (i.e. create functions for well-defined tasks).
- Documentation is also useful, but hard to maintain.

Writing readable code

Good code reads like a book.

- When it doesn't, make sure to use comments. In Matlab, everything following `% is a comment`
- Prevent "smart constructions" in the code
- Re-use working code (i.e. create functions for well-defined tasks).
- Documentation is also useful, but hard to maintain. (Matlab comes with a function that generates reports from comments)

How not to comment

- Useless:

```
% Start program
```

How not to comment

- Useless:

```
% Start program
```

- Obvious:

```
if (a > 5)    % Check if a is greater than 5
    ...
end
```

How not to comment

- Useless:

```
% Start program
```

- Obvious:

```
if (a > 5)    % Check if a is greater than 5
    ...
end
```

- Too much about the life:

```
% Well... I do not know how to explain what is going on
% in the snippet below. I tried to code in the night
% with some booze and it worked then, but now I have a
% strong hangover and some parameters still need to be
% worked out...
```

How not to comment

- Useless:

```
% Start program
```

- Obvious:

```
if (a > 5)    % Check if a is greater than 5
    ...
end
```

- Too much about the life:

```
% Well... I do not know how to explain what is going on
% in the snippet below. I tried to code in the night
% with some booze and it worked then, but now I have a
% strong hangover and some parameters still need to be
% worked out...
```

- ...

```
% You may think that this function is obsolete, and doesn't seem to
% do anything. And you would be correct. But when we remove this
% function for some reason the whole program crashes and we can't
% figure out why, so here it will stay.
```

Adding comments to our program

```
IAmFree = checkSchedule();
if (IAmFree == true)
% Count books and amount of free space on a shelf.
% If minimum number of books I need is less than a
% shelf capacity, go shopping and buy additional
% literature. If the amount of books after the
% shopping is too big, use boxes to store them.
  books = countBooks();
  shelfSize = countFreeSpaceShelf();

  ...

else
  burnBooks();
  goToBookStore();
end
```

If anything sticks today, let it be this

Your code will not be understood by anyone

If anything sticks today, let it be this

Your code will not be understood by anyone

That includes future-you

If anything sticks today, let it be this

Your code will not be understood by anyone

That includes future-you

Use comments and code to document design and purpose (functionality), not mechanics (implementation).

If anything sticks today, let it be this

Your code will not be understood by anyone

That includes future-you

Use comments and code to document design and purpose (functionality), not mechanics (implementation).

Use consistent and sensible naming of functions and variables.

Today's outline

- Coding in style
- Error management
- Visualisation
- Functions: the sequel
- Conclusions

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors You did not obey the language rules. These errors prevent running or compilation of the program.

Runtime errors Something goes wrong during the execution of the program resulting in an error message (problem with input, division by zero, loading of non-existent files, memory problems, etc.)

Semantic errors The program does not do what you expect, but does what have told it to do.

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors You did not obey the language rules. These errors prevent running or compilation of the program.

Runtime errors Something goes wrong during the execution of the program resulting in an error message (problem with input, division by zero, loading of non-existent files, memory problems, etc.)

Semantic errors The program does not do what you expect, but does what have told it to do.

Errors in computer programs

The following symptoms can be distinguished:

- Unable to execute the program
- Program crashes, warnings or error messages
- Never-ending loops
- Wrong (unexpected) result

Three error categories:

Syntax errors You did not obey the language rules. These errors prevent running or compilation of the program.

Runtime errors Something goes wrong during the execution of the program resulting in an error message (problem with input, division by zero, loading of non-existent files, memory problems, etc.)

Semantic errors The program does not do what you expect, but does what have told it to do.

A convenient tool: the debugger

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

A convenient tool: the debugger

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

A convenient tool: the debugger

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

A convenient tool: the debugger

- No-one can write a 1000-line code without making errors
  - If you can, please come work for us
- One of the most important skills you will acquire is debugging.
- Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.
- In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

"When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

— A. Conan Doyle, The Sign of Four

A convenient tool: the debugger

The debugger can help you to:

- Pause a program at a certain line: set a *breakpoint*
- Check the values of variables during the program
- Controlled execution of the program:
  - One line at a time
  - Run until a certain line
  - Run until a certain condition is met (conditional breakpoint)
  - Run until the current function exits
- Note: You may end up in the source code of Matlab functions!

A convenient tool: the debugger

The debugger can help you to:

- Pause a program at a certain line: set a *breakpoint*
- Check the values of variables during the program
- Controlled execution of the program:
  - One line at a time
  - Run until a certain line
  - Run until a certain condition is met (conditional breakpoint)
  - Run until the current function exits
- Note: You may end up in the source code of Matlab functions!
- Check Canvas (Matlab Crash Course section) for a movie that demonstrates the debugger.

About testcases (validation)

- Testcases: run the program with parameters such that a known result is (should be) produced.
- Testcases: what happens when unforeseen input is encountered?
  - More or fewer arguments than anticipated? (Matlab uses `varargin` and `nargin` to create a varying number of input arguments, and to check the number of given input arguments
  - Other data types than anticipated? How does the program handle this? Warnings, error messages (crash), NaN or worse (a continuing program)?
- For physical modeling, we typically look for analytical solutions
  - Sometimes somewhat stylized cases
  - Possible solutions include Fourier-series
  - Experimental data

Today's outline

- Coding in style
- Error management
- Visualisation
- Functions: the sequel
- Conclusions

Data visualisation

Modeling can lead to very large data sets, that require appropriate visualisation to convey your results.

- 1D, 2D, 3D visualisation
- Multiple variables at the same time (temperature, concentration, direction of flow)
- Use of colors, contour lines
- Use of stream lines or vector plots
- Animations

Plotting

```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
```

Plotting

```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
subplot(2,1,1); plot(x,y,'-',x,y2,'r.');
xlabel('X'); ylabel('Y'); title('Graph and Scatter');
```

Plotting

```
x = -5:0.1:5;
y = x.^2-4*x+3;
y2 = y + (2-4*rand(size(y)));
subplot(2,1,1); plot(x,y,'-',x,y2,'r.');
xlabel('X'); ylabel('Y'); title('Graph and Scatter');
subplot(2,1,2); plot(x,abs(y-y2),'r-');
xlabel('X'); ylabel('Y'); title('Absolute error');
```

Plotting (2)

Easy plotting of functions can be done using the ezplot function: `ezplot('x-sin(x)', [0 2*pi])`:

Be careful with steep gradients:

Other plotting tools

- Errorbars: `errorbar(x,y,err)`

Other plotting tools

- Errorbars: `errorbar(x,y,err)`
- 3D-plots: `plot3(x,y,z)`

Other plotting tools

- Errorbars: `errorbar(x,y,err)`
- 3D-plots: `plot3(x,y,z)`
- Histograms: `histogram(x,20)`

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot

```
surf(x,y,z);
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot
- Contour plot

```
v=-0.5:0.05:0.5;
contour(x,y,z,v,'ShowText', 'on');
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot
- Contour plot
- Waterfall

```
waterfall(x,y,z);
colormap(winter);
```

Multi-dimensional data

Matlab typically requires the definition of rectangular grid coordinates using `meshgrid`:

```
[x y] = meshgrid(-2:0.1:2, -2:0.1:2);
z = x .* y .* exp(-x.^2 - y.^2);
```

- Surface plot
- Contour plot
- Waterfall
- Ribbons

```
ribbon(z);
```

Vector data

The gradient operator, as expected, is used to obtain the gradient of a scalar field. Colors can be used in the background to simultaneously plot field data:

```
[x y] = meshgrid(-2:0.2:2, -2:0.2:2);
z = x .* y .* exp(-x.^2 - y.^2)
[dx dy] = gradient(z,8,8)

% Background
contourf(x,y,z,30,'LineColor','none');
colormap(hot); colorbar;

axis tight; hold on;

% Vectors
quiver(x,y,dx,dy,'k');
```

Today's outline

- Coding in style
- Error management
- Visualisation
- Functions: the sequel
- Conclusions

Functions: revisited

In MATLAB you can define your own functions to re-use certain functionalities. We now define the mathematical function $f = x^2 + e^x$:

```
function y = f(x)
y = x.^2 + exp(x);
```

Functions: revisited

In MATLAB you can define your own functions to re-use certain functionalities. We now define the mathematical function $f = x^2 + e^x$:

```
function y = f(x)
y = x.^2 + exp(x);
```

Note:

- The first line of the file has to contain the `function` keyword
- The variables used are *local*. They will not be available in your Workspace
- The file needs to be saved with the same name as the function, i.e. "f.m"
- The semi-colon prevents that at each function evaluation output appears on the screen
- If `x` is an array, then `y` becomes an array of function values.

Anonymous functions

If you do not want to create a file, you can create an *anonymous function*:

```
>> f = @(x) (x.^2+exp(x))
```

Anonymous functions

If you do not want to create a file, you can create an *anonymous function*:

```
>> f = @(x) (x.^2+exp(x))
```

- `f`: the name of the function
- `@`: the function handle
- `x`: the input argument
- `x.^2+exp(x)`: the actual function

Anonymous functions

If you do not want to create a file, you can create an *anonymous function*:

```
>> f = @(x) (x.^2+exp(x))
```

- `f`: the name of the function
- `@`: the function handle
- `x`: the input argument
- `x.^2+exp(x)`: the actual function

```
>> f(0:0.1:1)
```

Using function handles

A function handle points to a function. It behaves as a variable

```
>> myFunctionHandle = @exp
>> myFunctionHandle(1)
```

Using function handles

A function handle points to a function. It behaves as a variable

```
>> myFunctionHandle = @exp
>> myFunctionHandle(1)
```

Used a.o. for passing a function to another function, for instance for optimization functions.

$$f(x) = x^3 - x^2 - 3\arctan x + 1$$

Using function handles

A function handle points to a function. It behaves as a variable

```
>> myFunctionHandle = @exp
>> myFunctionHandle(1)
```

Used a.o. for passing a function to another function, for instance for optimization functions.

$$f(x) = x^3 - x^2 - 3\arctan x + 1$$

Matlab offers a function `fzero` that can find the roots of a function in a certain range:

```
>> f = @(x) x.^3 - x.^2 - 3*atan(x) + 1;
>> fzero(f,[-2 2])
>> ezplot(f)
>> f(ans)
```

Practice function handles

Consider the function

$$f(x) = -x^2 - 3x + 3 + e^{x^2}$$

The built-in Matlab function `fminbnd` allows to find the minimum of a function in a certain range. Find the minimum of $f(x)$ on $-2 \leq x \leq 2$. Example usage:

```
x = fminbnd(fun,x1,x2)
```

Practice function handles

Consider the function

$$f(x) = -x^2 - 3x + 3 + e^{x^2}$$

The built-in Matlab function `fminbnd` allows to find the minimum of a function in a certain range. Find the minimum of $f(x)$ on $-2 \leq x \leq 2$. Example usage:

```
x = fminbnd(fun,x1,x2)
```

Answer using an anonymous function:

```
>> f = @(x) -x.^2 - 3*x + 3 + exp(x.^2)
>> ezplot(f,[-2 2])
>> fminbnd(f,-2,2)
>> f(ans)
```

Practice function handles

Consider the function

$$f(x) = -x^2 - 3x + 3 + e^{x^2}$$

The built-in Matlab function `fminbnd` allows to find the minimum of a function in a certain range. Find the minimum of $f(x)$ on $-2 \leq x \leq 2$. Example usage:

```
x = fminbnd(fun,x1,x2)
```

Answer using an anonymous function:

```
>> f = @(x) -x.^2 - 3*x + 3 + exp(x.^2)
>> ezplot(f,[-2 2])
>> fminbnd(f,-2,2)
>> f(ans)
```

Various related functions are `fzero`, `feval`, `fsolve`, `fminsearch`. They will be discussed later in the course.

Today's outline

- Coding in style
- Error management
- Visualisation
- Functions: the sequel
- Conclusions

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations
- For now: exercises 1-4 (basics), 5+6 (advanced).
- Preparation for next lecture: familiarize with the concepts, especially part 1 of these slides.

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations
- For now: exercises 1-4 (basics), 5+6 (advanced).
- Preparation for next lecture: familiarize with the concepts, especially part 1 of these slides.

In conclusion...

- Matlab: A versatile development environment, with excellent vector and matrix computations
- Programming basics: variables, operators and functions, locality of variables, recursive operations
- For now: exercises 1-4 (basics), 5+6 (advanced).
- Preparation for next lecture: familiarize with the concepts, especially part 1 of these slides.

Numerical errors in computer simulations

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Example 1

Start your spreadsheet program (Excel, ...)

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`
A4	`=(A3*10)-0.9`

(repeat until A30)

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`
A4	`=(A3*10)-0.9`

(repeat until A30)

What's happening?

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	`0.1`
A2	`=(A1*10)-0.9`
A3	`=(A2*10)-0.9`
A4	`=(A3*10)-0.9`

(repeat until A30)

What's happening?

Enter:

Cell	Value
A1	2

Example 1

Start your spreadsheet program (Excel, ...)

Enter:

Cell	Value
A1	0.1
A2	=(A1*10)-0.9
A3	=(A2*10)-0.9
A4	=(A3*10)-0.9

(repeat until A30)

What's happening?

Enter:

Cell	Value
A1	2
A2	=(A1*10)-18
A3	=(A2*10)-18
A4	=(A3*10)-18

(repeat until A30)

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)

Verification and validation

Problem

Verification and validation

Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

Verification of the physical model

Verification of the physical model

Verification of the physical model

Verification of the physical model

Verification of the physical model

Verification of the physical model

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)

Verification of the physical model

- The perceived orbit of Mars from Earth shows a zig-zag (in contrast to the Sun, Mercury, Venus)
- Even though they were not 'right', Earth-centered models (Ptolemy) were still valid

Be aware of your uncertainties

Aleatory uncertainty

Uncertainty that arises due to inherent randomness of the system, features that are too complex to measure and take into account

Epistemic uncertainty

Uncertainty that arises due to lack of knowledge of the system, but could in principle be known

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)

Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters

Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

Verification and validation

Verification

Verification is the process of mathematically and computationally assuring that the model computes what you have entered.

Validation

Validation is the process of determining the degree to which a model is an accurate representation of the real world from the perspective of the intended uses of the model

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

Errors in computer simulations

In this course we will outline different numerical errors that may appear in computer simulations, and how these errors can affect the simulation results.

- Errors in the mathematical model (physics)
- Errors in the program (implementation)
- Errors in the entered parameters
- Roundoff- and truncation errors
- Break errors

Significant digits

A numerical result $\tilde{x}$ is an approximation of the real value x.

- Absolute error

$$\delta = |\tilde{x} - x|, x \neq 0$$

- Relative error

$$\frac{\delta}{\tilde{x}} = |\frac{\tilde{x} - x}{\tilde{x}}|$$

- Error margin

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta$$

$$x = \tilde{x} \pm \delta$$

Significant digits

A numerical result $\tilde{x}$ is an approximation of the real value x.

- Absolute error

$$\delta = |\tilde{x} - x|, x \neq 0$$

- Relative error

$$\frac{\delta}{\tilde{x}} = |\frac{\tilde{x} - x}{\tilde{x}}|$$

- Error margin

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta$$

$$x = \tilde{x} \pm \delta$$

Significant digits

A numerical result $\tilde{x}$ is an approximation of the real value x.

- Absolute error

$$\delta = |\tilde{x} - x|, x \neq 0$$

- Relative error

$$\frac{\delta}{\tilde{x}} = |\frac{\tilde{x} - x}{\tilde{x}}|$$

- Error margin

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta$$

$$x = \tilde{x} \pm \delta$$

Significant digits

- $\tilde{x}$ has m significant digits if the absolute error in x is smaller or equal to 5 at the $(m+1)$-th position:

$$10^{q-1} \leq |\tilde{x}| \leq 10^{q}$$

$$|x - \tilde{x}| = 0.5 \times 10^{q-m}$$

- For example:

$$x = \frac{1}{3}, \tilde{x} = 0.333 \Rightarrow \delta = 0.00033333\ldots$$

3 significant digits

Significant digits

- $\tilde{x}$ has m significant digits if the absolute error in x is smaller or equal to 5 at the $(m+1)$-th position:

$$10^{q-1} \leq |\tilde{x}| \leq 10^{q}$$

$$|x - \tilde{x}| = 0.5 \times 10^{q-m}$$

- For example:

$$x = \frac{1}{3}, \tilde{x} = 0.333 \Rightarrow \delta = 0.00033333\ldots$$

3 significant digits

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

4 5 2 1

- 1 → Number of 1's: $1 \times 10^{(0)}$
- 2 → Number of 10's: $2 \times 10^{(1)}$
- 5 → Number of 100's: $5 \times 10^{(2)}$
- 4 → Number of 1000's: $4 \times 10^{(3)}$

Representation of numbers

- Computers represent a number with a finite number of digits: each number is therefore an approximation due to roundoff and truncation errors.
- In the decimal system, a digit c at position n has a value of $c \times 10^{n-1}$

4 5 2 1

- 1 → Number of 1's: $1 \times 10^{(0)}$
- 2 → Number of 10's: $2 \times 10^{(1)}$
- 5 → Number of 100's: $5 \times 10^{(2)}$
- 4 → Number of 1000's: $4 \times 10^{(3)}$

$$(4521)_{10} = 4 \times 10^3 + 5 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$(4521)_{10} =$

$=($

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} +$$

$$=(1$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} +$$

$$=(10$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} +$$

$$=(100$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} +$$

$$=(1000$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots$$

$$= (10001$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots$$

$$\ldots 1 \times 2^{7} +$$

$$=(100011$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots$$
$$\ldots 1 \times 2^{7} + 0 \times 2^{6} +$$

$$=(1000110$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots$$
$$\ldots 1 \times 2^{7} + 0 \times 2^{6} + 1 \times 2^{5} +$$

$$=(10001101$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1 \times 2^{12} + 0 \times 2^{11} + 0 \times 2^{10} + 0 \times 2^{9} + 1 \times 2^{8} + \ldots$$
$$\ldots 1 \times 2^{7} + 0 \times 2^{6} + 1 \times 2^{5} + 0 \times 2^{4} + \ldots$$

$$= (100011010$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$(4521)_{10} = 1\times 2^{12} + 0\times 2^{11} + 0\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + \ldots$$
$$\ldots 1\times 2^{7} + 0\times 2^{6} + 1\times 2^{5} + 0\times 2^{4} + \ldots$$
$$\ldots 1\times 2^{3} +$$
$$=(1000110101$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = & 1\times 2^{12} + 0\times 2^{11} + 0\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + \ldots \\
& \ldots 1\times 2^{7} + 0\times 2^{6} + 1\times 2^{5} + 0\times 2^{4} + \ldots \\
& \ldots 1\times 2^{3} + 0\times 2^{2} + \\
& =(10001101010
\end{aligned}
$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = & 1\times 2^{12} + 0\times 2^{11} + 0\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + \ldots \\
& \ldots 1\times 2^{7} + 0\times 2^{6} + 1\times 2^{5} + 0\times 2^{4} + \ldots \\
& \ldots 1\times 2^{3} + 0\times 2^{2} + 0\times 2^{1} + \\
& =(100011010100
\end{aligned}
$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = & 1\times 2^{12} + 0\times 2^{11} + 0\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + \ldots \\
& \ldots 1\times 2^{7} + 0\times 2^{6} + 1\times 2^{5} + 0\times 2^{4} + \ldots \\
& \ldots 1\times 2^{3} + 0\times 2^{2} + 0\times 2^{1} + 1\times 2^{0} \\
& = (1000110101001)_2
\end{aligned}
$$

Representation of numbers

- You could use another basis, computers often use the basis 2:

$$
\begin{aligned}
(4521)_{10} = & 1\times 2^{12} + 0\times 2^{11} + 0\times 2^{10} + 0\times 2^{9} + 1\times 2^{8} + \ldots \\
& \ldots 1\times 2^{7} + 0\times 2^{6} + 1\times 2^{5} + 0\times 2^{4} + \ldots \\
& \ldots 1\times 2^{3} + 0\times 2^{2} + 0\times 2^{1} + 1\times 2^{0} \\
& = (1000110101001)_2
\end{aligned}
$$

- In general:

$$(c_m \ldots c_1 c_0)_q = c_0 q^0 + c_1 q^1 + \ldots + c_m q^m, c \in \{0, 1, 2, \ldots, q-1\}$$

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

Representation of numbers

- Numbers are stored in binary in the memory of a computer, in segments of a specific length (called a *word*).
- We distinguish multiple types of numbers:
  - Integers: $-301, -1, 0, 1, 96, 2293, \ldots$
  - Floating points: $-301.01, 0.01, 3.14159265, 14498.2$
- A binary integer representation looks like the following bit sequence:

$$z = \sigma\left(c_0 2^0 + c_1 2^1 + \ldots + c_{\lambda-1} 2^{\lambda-1}\right)$$

  σ is the sign of z (+ or -), and λ is the length of the word
- Endianness: the order of bits stored by a computer

Excercise

- Convert the following decimal number to base-2: 214

Excercise

- Convert the following decimal number to base-2: 214

$$214_{10} = 11010110_2$$

Excercise

- Convert the following decimal number to base-2: 214

$$214_{10} = 11010110_2$$

- Excel:
  - Decimal: `=DEC2BIN(214)`
  - Octal: `=DEC2OCT(214)`
  - Hexadecimal: `=DEC2HEX(214)`

Excercise

- Convert the following decimal number to base-2: 214

$$214_{10} = 11010110_2$$

- Excel:
  - Decimal: `=DEC2BIN(214)`
  - Octal: `=DEC2OCT(214)`
  - Hexadecimal: `=DEC2HEX(214)`
- Matlab:
  - Decimal: `dec2bin(214)`
  - Other base: `dec2base(214,<base>)`

Arithmetic operations with binary numbers

Addition:

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

Arithmetic operations with binary numbers

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

Addition:

$$\begin{array}{rccc} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

Arithmetic operations with binary numbers

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

$$\begin{array}{rccc} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

Addition:

$$\begin{array}{rcccccccc} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

Arithmetic operations with binary numbers

Addition:

$0+0=0$

$0+1=1$

$1+0=1$

$1+1=0$ (carry one)

$$\begin{array}{rccc} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{rcccccccc} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Arithmetic operations with binary numbers

Addition:

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array} \qquad \begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Subtraction:

$0-0=0$
$1-0=1$
$1-1=0$
$0-1=1$ (borrow one)

Arithmetic operations with binary numbers

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

Addition:

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array}$$

$$\begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

$0-0=0$
$1-0=1$
$1-1=0$
$0-1=1$ (borrow one)

Subtraction:

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ - & & 2 & 3 \\ \hline & 1 & 2 & 2 \end{array}$$

Arithmetic operations with binary numbers

Addition:

$0+0=0$
$0+1=1$
$1+0=1$
$1+1=0$ (carry one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array} \qquad \begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Subtraction:

$0-0=0$
$1-0=1$
$1-1=0$
$0-1=1$ (borrow one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ - & & 2 & 3 \\ \hline & 1 & 2 & 2 \end{array} \qquad \begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ - & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

Arithmetic operations with binary numbers

Addition:

$0+0=0$

$0+1=1$

$1+0=1$

$1+1=0$ (carry one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ + & & 2 & 3 \\ \hline & 1 & 6 & 8 \end{array} \qquad \begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ + & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{array}$$

Subtraction:

$0-0=0$

$1-0=1$

$1-1=0$

$0-1=1$ (borrow one)

$$\begin{array}{rrrr} & 1 & 4 & 5 \\ - & & 2 & 3 \\ \hline & 1 & 2 & 2 \end{array} \qquad \begin{array}{rrrrrrrrr} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ - & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ \hline & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{array}$$

- Multiplication and division are more expensive, and more elaborate

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	`-2147483648`

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	`-2147483648`
`intmax`	`2147483647`
`i = int16(intmax)`	`i = 32767`

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308
`f = 0.1`	

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308
`f = 0.1`	
`whos f`	`double` information

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308
`f = 0.1`	
`whos f`	`double` information
`format long e`	

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308
`f = 0.1`	
`whos f`	`double` information
`format long e`	
`realmax`	1.797693134862316e+308

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308
`f = 0.1`	
`whos f`	`double` information
`format long e`	
`realmax`	1.797693134862316e+308
`f`	

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308
`f = 0.1`	
`whos f`	`double` information
`format long e`	
`realmax`	1.797693134862316e+308
`f`	
`fprintf("%0.16f",f)`	0.1000000000000000

Excercise

Try the following commands in Matlab:

Command	Result
`intmin`	-2147483648
`intmax`	2147483647
`i = int16(intmax)`	`i = 32767`
`whos i`	`int16` information
`i = i + 100`	`i = 32767`
`realmax`	1.7977e+308
`f = 0.1`	
`whos f`	`double` information
`format long e`	
`realmax`	1.797693134862316e+308
`f`	
`fprintf("%0.16f",f)`	0.1000000000000000
`fprintf("%0.20f",f)`	0.10000000000000000555

Representation of integer numbers

- In Matlab, integers of the type `int32` are represented by 32-bit words ($\lambda = 31$).

[2] Matlab does not perform actual integer overflows, it just stops at the maximum.

Representation of integer numbers

- In Matlab, integers of the type `int32` are represented by 32-bit words ($\lambda = 31$).
- The set of numbers that an `int32` z can represent is:

$$-2^{31} \leq z \leq 2^{31} - 1 \approx 2 \times 10^9$$

[2] Matlab does not perform actual integer overflows, it just stops at the maximum.

Representation of integer numbers

- In Matlab, integers of the type `int32` are represented by 32-bit words ($\lambda = 31$).
- The set of numbers that an `int32` z can represent is:

$$-2^{31} \leq z \leq 2^{31} - 1 \approx 2 \times 10^9$$

- If, during a calculation, an integer number becomes larger than $2^{\lambda} - 1$, the computer reports an overflow[2]

[2] Matlab does not perform actual integer overflows, it just stops at the maximum.

Representation of integer numbers

- In Matlab, integers of the type `int32` are represented by 32-bit words ($\lambda = 31$).
- The set of numbers that an `int32` z can represent is:

$$-2^{31} \leq z \leq 2^{31} - 1 \approx 2 \times 10^9$$

- If, during a calculation, an integer number becomes larger than $2^{\lambda} - 1$, the computer reports an overflow[2]
- How can a computer identify an overflow?

[2] Matlab does not perform actual integer overflows, it just stops at the maximum.

Representation of real (floating point) numbers

- Formally, a real number is represented by the following bit sequence

$$x = \sigma\left(2^{-1} + c_2 2^{-2} + \ldots + c_m 2^{-m}\right) 2^{e-1023}$$

  Here, σ is the sign of x and e is an integer value.

- A floating point number hence contains sections that contain the sign, the exponent and the mantissa

Image: Wikimedia Commons CC by-SA

Representation of real (floating point) numbers

- Example: $\lambda = 3,\ m = 2,\ x = \frac{2}{3}$

$$x = \pm\left(2^{-1} + c_2 2^{-2}\right)2^e$$

  - $c_0 \in \{0, 1\}$
  - $e = \pm a_0 2^0$
  - $a_0 \in \{0, 1\}$
- Truncation: $fl(x) = 2^{-1} = 0.5$
- Round off: $fl(x) = 2^{-1} + 2^{-2} = 0.75$

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Trigonometric, Logarithmic, and Exponential computations

- Processors can do logic and arithmetic instructions
- Trigonometric, logarithmic and exponential calculations are "higher-level" functions: exp, sin, cos, tan, sec, arcsin, arccos, arctan, log, ln, ...
- Such functions can be performed using these "low level" instructions, for instance using a Taylor series:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Trigonometric, Logarithmic, and Exponential computations

- Processors can do logic and arithmetic instructions
- Trigonometric, logarithmic and exponential calculations are "higher-level" functions: exp, sin, cos, tan, sec, arcsin, arccos, arctan, log, ln, ...
- Such functions can be performed using these "low level" instructions, for instance using a Taylor series:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Trigonometric, Logarithmic, and Exponential computations

- Processors can do logic and arithmetic instructions
- Trigonometric, logarithmic and exponential calculations are "higher-level" functions: exp, sin, cos, tan, sec, arcsin, arccos, arctan, log, ln, ...
- Such functions can be performed using these "low level" instructions, for instance using a Taylor series:

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots$$

Trigonometric, Logarithmic, and Exponential computations

- These operations involve many multiplications and additions, and are therefore *expensive*
- Computations can only take finite time, for infinite series, calculations are interrupted at N

$$\sin(x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^N}{(2N+1)!} x^{2N+1}$$

$$e^x = \sum_{n=0}^{N} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!}$$

- This results in a *break error*

Trigonometric, Logarithmic, and Exponential computations

- These operations involve many multiplications and additions, and are therefore *expensive*
- Computations can only take finite time, for infinite series, calculations are interrupted at N

$$\sin(x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^N}{(2N+1)!} x^{2N+1}$$

$$e^x = \sum_{n=0}^{N} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!}$$

- This results in a *break error*

Trigonometric, Logarithmic, and Exponential computations

- These operations involve many multiplications and additions, and are therefore *expensive*
- Computations can only take finite time, for infinite series, calculations are interrupted at N

$$\sin(x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^N}{(2N+1)!} x^{2N+1}$$

$$e^x = \sum_{n=0}^{N} \frac{x^n}{x!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots + \frac{x^N}{N!}$$

- This results in a *break error*

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry $(0 \leq x \leq \frac{\pi}{2})$

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry $(0 \leq x \leq \frac{\pi}{2})$
3. Use lookup tables for known values

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values
4. Perform taylor expansion

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values
4. Perform taylor expansion

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values
4. Perform taylor expansion

Algorithm for sine-computation

A computer may use a clever algorithm to limit the number of operations required to perform a higher-level function. A (fictional!) example for the computation of $\sin(x)$:

1. Use periodicity so that $0 \leq x \leq 2\pi$
2. Use symmetry ($0 \leq x \leq \frac{\pi}{2}$)
3. Use lookup tables for known values
4. Perform taylor expansion

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Loss of digits

- During operations such as $+$, $-$, $\times$, $\div$, an error can add up
- Consider the summation of x and y

$$\tilde{x} - \delta \leq x \leq \tilde{x} + \delta \quad \text{and} \quad \tilde{y} - \varepsilon \leq y \leq \tilde{y} + \varepsilon$$

$$(\tilde{x} + \tilde{y}) - (\delta + \varepsilon) \leq x + y \leq (\tilde{x} + \tilde{y}) + (\delta + \varepsilon)$$

Loss of digits: Example 1

$$x = \pi, \tilde{x} = 3.1416$$
$$y = 22/7, \tilde{y} = 3.1429$$

Loss of digits: Example 1

$$\left.\begin{array}{l} x = \pi, \tilde{x} = 3.1416 \\ y = 22/7, \tilde{y} = 3.1429 \end{array}\right\} \Rightarrow \left.\begin{array}{l} \delta = \tilde{x} - x = 7.35 \times 10^{-6} \\ \varepsilon = \tilde{y} - y = 4.29 \times 10^{-5} \end{array}\right\}$$

Loss of digits: Example 1

$$\left.\begin{array}{l} x = \pi, \tilde{x} = 3.1416 \\ y = 22/7, \tilde{y} = 3.1429 \end{array}\right\} \Rightarrow \left.\begin{array}{l} \delta = \tilde{x} - x = 7.35 \times 10^{-6} \\ \varepsilon = \tilde{y} - y = 4.29 \times 10^{-5} \end{array}\right\}$$

$$x + y = \tilde{x} + \tilde{y} \pm (\delta + \varepsilon) \approx 6.2845 - 5.025 \times 10^{-5}$$

$$x - y = \tilde{x} - \tilde{y} \pm (\delta + \varepsilon) \approx -0.0013 + 3.55 \times 10^{-5}$$

- The absolute error is small ($\approx 10^{-5}$), but the relative error is much bigger (0.028).
- Adding up the errors results in a loss of significant digits!

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find?
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find?
  Use: `str2double(sprintf('%.4g', term))`
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find? Use: `str2double(sprintf('%.4g', term))`
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Loss of digits: Example 2

- Calculate e^{-5}
  - Use the Taylor series
  - Calculate the first 26 terms ($N = 26$)
- Now repeat the calculation, but use for each calculation only 4 digits. What do you find? Use: `str2double(sprintf('%.4g', term))`
- Without errors you would find: $e^{-5} = 0.006738$
- If you only use 4 digits in the calculations, you'll find 0.00998

Badly (ill) conditioned problems

We consider a system $F(x, y)$ that computes a solution from input data. The input data may have errors:

$$\xrightarrow{x} \boxed{F(x, y) = 0} \xrightarrow{y} \text{(Without errors)}$$

Badly (ill) conditioned problems

We consider a system $F(x, y)$ that computes a solution from input data. The input data may have errors:

$$\xrightarrow{x} \boxed{F(x,y) = 0} \xrightarrow{y} \text{(Without errors)}$$

$$\xrightarrow{x+\delta x} \boxed{F(x+\delta x, y+\delta y) = 0} \xrightarrow{y+\delta y} \text{(With errors)}$$

Badly (ill) conditioned problems

We consider a system $F(x, y)$ that computes a solution from input data. The input data may have errors:

$$\xrightarrow{x} \boxed{F(x,y) = 0} \xrightarrow{y} \text{(Without errors)}$$

$$\xrightarrow{x+\delta x} \boxed{F(x+\delta x, y+\delta y) = 0} \xrightarrow{y+\delta y} \text{(With errors)}$$

$$y(x+\delta x) - y(x) \approx y'(x)\delta x$$

Propagated error on the basis of Taylor expansion

$$C = \max_{\delta x}\left(\left|\frac{\delta y/y}{\delta x/x}\right|\right)$$

Condition criterion, $C < 10$ error development small

Badly (ill) conditioned problems: Example

Solve the following linear system in Matlab using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}$$

Badly (ill) conditioned problems: Example

Solve the following linear system in Matlab using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}, \quad y = \begin{bmatrix} 2.0 \\ 2.0 \end{bmatrix}$$

Badly (ill) conditioned problems: Example

Solve the following linear system in Matlab using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}, \quad y = \begin{bmatrix} 2.0 \\ 2.0 \end{bmatrix}$$

Double precision

```
>> clear;clc;format long e;
>> A = [[1.2969 0.8648]; [0.2161 0.1441]];
>> x = [0.8642; 0.1440];
>> y = A\x
y =
     2.000000002400302e+00
    -2.000000003599621e+00
```

Badly (ill) conditioned problems: Example

Solve the following linear system in Matlab using double and single precision:

$$A = \begin{bmatrix} 1.2969 & 0.8648 \\ 0.2161 & 0.1441 \end{bmatrix}, \quad x = \begin{bmatrix} 0.8642 \\ 0.1440 \end{bmatrix}, \quad y = \begin{bmatrix} 2.0 \\ 2.0 \end{bmatrix}$$

Double precision

```
>> clear;clc;format long e;
>> A = [[1.2969 0.8648]; [0.2161 0.1441]];
>> x = [0.8642; 0.1440];
>> y = A\x
y =
     2.000000002400302e+00
    -2.000000003599621e+00
```

Single precision

```
>> clear;clc;format long e;
>> A = single(
  [[1.2969 0.8648];
  [0.2161 0.1441]] );
>> x = single(
  [0.8642; 0.1440] );
>> y = A\x
y =
   1.3331791e+00
  -1.0000000e+00
```

Badly (ill) conditioned problems: Example

- Matlab already warned us about the bad condition number:

```
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 1.148983e
-08.
```

- The `RCOND` is the reciprocal condition number
- A small error in x results in a big error in y. This is called an ill conditioned problem.

Badly (ill) conditioned problems: Example

- Matlab already warned us about the bad condition number:

  `Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 1.148983e-08.`

- The `RCOND` is the reciprocal condition number
- A small error in x results in a big error in y. This is called an ill conditioned problem.

Badly (ill) conditioned problems: Example

- Matlab already warned us about the bad condition number:

  `Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 1.148983e-08.`

- The `RCOND` is the reciprocal condition number
- A small error in x results in a big error in y. This is called an ill conditioned problem.

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

(Un)stable methods

- The condition criterion does not tell you anything about the quality of a numerical solution method!
- It is very well possible that a certain solution method is more sensitive for one problem than another
- If the method propagates the error, we call it an *unstable method*. Let's look at an example.

The Golden mean

- Let's evaluate the following recurrent relationship:

$$y_{n+1} = y_{n-1} - y_n$$

$$y_0 = 1, \quad y_1 = \frac{2}{1+\sqrt{5}}$$

- You can prove (by substitution) that:

$$y_n = x^{-n}, \quad n = 0, 1, 2, \ldots, \quad x = \frac{1+\sqrt{5}}{2}$$

The Golden mean

- Let's evaluate the following recurrent relationship:

$$y_{n+1} = y_{n-1} - y_n$$

$$y_0 = 1, \quad y_1 = \frac{2}{1+\sqrt{5}}$$

- You can prove (by substitution) that:

$$y_n = x^{-n}, \quad n = 0, 1, 2, \ldots, \quad x = \frac{1+\sqrt{5}}{2}$$

The Golden mean

Recurrent version

```
% initialise
y(1) = 1;
y(2) = 2 / (1 + sqrt(5));

% Perform recurrent approach
for n = 2:39
    y(n+1) = y(n-1)-y(n);
end
```

The Golden mean

Recurrent version

```
% initialise
y(1) = 1;
y(2) = 2 / (1 + sqrt(5));

% Perform recurrent approach
for n = 2:39
    y(n+1) = y(n-1)-y(n);
end
```

Powerlaw version

```
% initialise
x = (1 + sqrt(5))/2;
y2(1) = x^0; % n = 1

% Perform powerlaw apprach
for n = 0:39
    y2(n+1) = x^-n
end
```

The Golden mean

n	Recurrent	Powerlaw
1	1.0000	1.0000
1	0.6180	0.6180
2	0.3820	0.3820
3	0.2361	0.2361
...	...	...
37	$3.080 \cdot 10^{-08}$	$2.995 \cdot 10^{-08}$
38	$1.714 \cdot 10^{-08}$	$1.851 \cdot 10^{-08}$
39	$1.366 \cdot 10^{-08}$	$1.144 \cdot 10^{-08}$
40	$3.485 \cdot 10^{-08}$	$7.071 \cdot 10^{-08}$

- The recurrent approach enlarges errors from earlier calculations!

The Golden mean

n	Recurrent	Powerlaw
1	1.0000	1.0000
1	0.6180	0.6180
2	0.3820	0.3820
3	0.2361	0.2361
...	...	...
37	$3.080 \cdot 10^{-08}$	$2.995 \cdot 10^{-08}$
38	$1.714 \cdot 10^{-08}$	$1.851 \cdot 10^{-08}$
39	$1.366 \cdot 10^{-08}$	$1.144 \cdot 10^{-08}$
40	$3.485 \cdot 10^{-08}$	$7.071 \cdot 10^{-08}$

- The recurrent approach enlarges errors from earlier calculations!

Example 1: Explanation

Recall example 1, where the errors blew up our computation of 0.1, whereas they did not for 2. Why did we see these results?

Example 1: Explanation

Recall example 1, where the errors blew up our computation of 0.1, whereas they did not for 2. Why did we see these results?

- The number 0.1 is not exactly represented in binary
  - A tiny error can accumulate up to catastrophic proportions!
- The number 2 does have an exact binary representation

Example 1: Explanation

Recall example 1, where the errors blew up our computation of 0.1, whereas they did not for 2. Why did we see these results?

- The number 0.1 is not exactly represented in binary
  - A tiny error can accumulate up to catastrophic proportions!
- The number 2 does have an exact binary representation

Example 2

Start your calculation program of choice (Excel, Matlab, ...)

Example 2

Start your calculation program of choice (Excel, Matlab, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi$$

Example 2

Start your calculation program of choice (Excel, Matlab, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi = 19.999099979$$

Example 2

Start your calculation program of choice (Excel, Matlab, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi = 19.999099979 \neq 20$$

Example 2

Start your calculation program of choice (Excel, Matlab, ...)

Calculate the result of y:

$$y = e^{\pi} - \pi = 19.999099979 \neq 20$$

Image: xkcd

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Symbolic math packages

Definition

The use of computers to manipulate mathematical equations and expressions in symbolic form, as opposed to manipulating the numerical quantities represented by those symbols.

- Symbolic integration or differentiation, substitution of one expression into another
- Simplification of an expression, change of subject etc.
- Packages and toolboxes:

Symbolic math packages

Mathematica Well known software package, license available via TU/e

Maple Well known, license available via TU/e

Wolfram|Alpha Web-based interface by Mathematica developer. Less powerful in mathematical respect, but more accessible and has a broad application range (unit conversion, semantic commands).

Sage Open-source alternative to Maple, Mathematica, Magma, and MATLAB.

Matlab Symbolic math toolbox

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

```
>> syms x
>> f = (x - 1)*(x + 1)*(x^2 + 1) + 1
```

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

```
>> syms x
>> f = (x - 1)*(x + 1)*(x^2 + 1) + 1

f =

(x^2 + 1)*(x - 1)*(x + 1) + 1
```

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

```
>> syms x
>> f = (x - 1)*(x + 1)*(x^2 + 1) + 1

f =

(x^2 + 1)*(x - 1)*(x + 1) + 1

>> f2 = simplify(f)
```

Symbolic math: simplify

$$f(x) = (x-1)(x+1)(x^2+1)+1$$

```
>> syms x
>> f = (x - 1)*(x + 1)*(x^2 + 1) + 1

f =

(x^2 + 1)*(x - 1)*(x + 1) + 1

>> f2 = simplify(f)

f2 =

x^4
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```
>> syms x
>> f = 1/(x^3+1);
>> my_f_int = int(f)
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```
>> syms x
>> f = 1/(x^3+1);
>> my_f_int = int(f)

my_f_int = log(x + 1)/3 - log((x - 1/2)^2 + 3/4)/6 + (3^(1/2)*atan((2*3^(1/2)*(x - 1/2))
     /3))/3
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```
>> syms x
>> f = 1/(x^3+1);
>> my_f_int = int(f)

my_f_int = log(x + 1)/3 - log((x - 1/2)^2 + 3/4)/6 + (3^(1/2)*atan((2*3^(1/2)*(x - 1/2))
     /3))/3

>> my_f_diff = diff(my_f_int)
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```
>> syms x
>> f = 1/(x^3+1);
>> my_f_int = int(f)

my_f_int = log(x + 1)/3 - log((x - 1/2)^2 + 3/4)/6 + (3^(1/2)*atan((2*3^(1/2)*(x - 1/2))
     /3))/3

>> my_f_diff = diff(my_f_int)

my_f_diff = 1/(3*(x + 1)) + 2/(3*((4*(x - 1/2)^2)/3 + 1)) - (2*x - 1)/(6*((x - 1/2)^2 +
     3/4))
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```
>> syms x
>> f = 1/(x^3+1);
>> my_f_int = int(f)

my_f_int = log(x + 1)/3 - log((x - 1/2)^2 + 3/4)/6 + (3^(1/2)*atan((2*3^(1/2)*(x - 1/2))
    /3))/3

>> my_f_diff = diff(my_f_int)

my_f_diff = 1/(3*(x + 1)) + 2/(3*((4*(x - 1/2)^2)/3 + 1)) - (2*x - 1)/(6*((x - 1/2)^2 +
    3/4))

>> simplify(my_f_diff)
```

Symbolic math: integration and differentiation

$$f(x) = \frac{1}{x^3 + 1}$$

```
>> syms x
>> f = 1/(x^3+1);
>> my_f_int = int(f)

my_f_int = log(x + 1)/3 - log((x - 1/2)^2 + 3/4)/6 + (3^(1/2)*atan((2*3^(1/2)*(x - 1/2))
     /3))/3

>> my_f_diff = diff(my_f_int)

my_f_diff = 1/(3*(x + 1)) + 2/(3*((4*(x - 1/2)^2)/3 + 1)) - (2*x - 1)/(6*((x - 1/2)^2 +
     3/4))

>> simplify(my_f_diff)

ans = 1/(x^3 + 1)
```

Symbolic math: exercises

Exercise 1

Simplify the following expression:

$$f(x) = \frac{2\tan x}{(1+\tan^2 x)}$$

Symbolic math: exercises

Exercise 1

Simplify the following expression:

$$f(x) = \frac{2\tan x}{(1+\tan^2 x)} = \sin 2x$$

```
>> simplify(2*tan(x)/(1 + tan(x)^2))
```

Symbolic math: exercises

Exercise 1

Simplify the following expression:

$$f(x) = \frac{2\tan x}{(1+\tan^2 x)} = \sin 2x$$

```
>> simplify(2*tan(x)/(1 + tan(x)^2))
```

Exercise 2

Calculate the *value* of p:

$$p = \int_0^{10} \frac{e^x - e^{-x}}{\sinh x} dx$$

Symbolic math: exercises

Exercise 1

Simplify the following expression:

$$f(x) = \frac{2\tan x}{(1+\tan^2 x)} = \sin 2x$$

```
>> simplify(2*tan(x)/(1 + tan(x)^2))
```

Exercise 2

Calculate the *value* of p:

$$p = \int_0^{10} \frac{e^x - e^{-x}}{\sinh x} dx$$

```
>> f = ((exp(x)- exp(-x))/sinh(x));
>> p = int(f,0,10)
p = 20
```

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Matlab.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Matlab.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

```
>> syms x
>> f = 3 / (x^2 + 3*x) - 2;
>> solve(f)
ans =
   15^(1/2)/2 - 3/2
 - 15^(1/2)/2 - 3/2
```

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Matlab.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

```
>> syms x
>> f = 3 / (x^2 + 3*x) - 2;
>> solve(f)
ans =
   15^(1/2)/2 - 3/2
 - 15^(1/2)/2 - 3/2
```

Function as a string

$$f(x) = x^2 - 4x + 2$$

Symbolic math: root finding

A root finding method searches for the values where a function reaches zero. We will cover the numerical methods later, here we show how to use root finding with symbolic math in Matlab.

Symbolic math function

$$f(x) = \frac{3}{x^2 + 3x} - 2$$

```
>> syms x
>> f = 3 / (x^2 + 3*x) - 2;
>> solve(f)
ans =
   15^(1/2)/2 - 3/2
 - 15^(1/2)/2 - 3/2
```

Function as a string

$$f(x) = x^2 - 4x + 2$$

```
>> solve('x^2 - 4*x + 2')
ans =
 2^(1/2) + 2
 2 - 2^(1/2)
```

Symbolic math toolbox: variable precision arithmetic

Variable precision can be used to specify the number of significant digits.

```
>> p = vpa(1/3,16)
p = 0.3333333333333333
>> p = vpa(1/3,4)
p = 0.3333
>> a = vpa(0.1, 30)
a = 0.1
>> b = vpa(0.1, 5);
b = 0.1
>> a-b

ans = 0.000000000000005684341886080801486968993846751 4
```

Today's outline

- Introduction
- Roundoff and truncation errors
- Break errors
- Loss of digits
- (Un)stable methods
- Symbolic math
- Summary

Summary

- Numerical errors mar arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Summary

- Numerical errors mar arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Summary

- Numerical errors mar arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Summary

- Numerical errors mar arise due to truncation, roundoff and break errors, which may seriously affect the accuracy of your solution
- Errors may propagate and accumulate, leading to smaller accuracy
- Ill-conditioned problems and unstable methods have to be identified so that proper measures can be taken
- Symbolic math computations may be performed to solve certain equations algebraically, bypassing numerical errors, but this is not always possible.

Linear equations 1

Linear algebra basics

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Overview

Goals

- Different ways of looking at a system of linear equations
- Determination of the inverse, determinant and the rank of a matrix
- The existence of a solution to a set of linear equations

Different views of linear systems

- Separate equations:

$$x + y + z = 4$$
$$2x + y + 3z = 7$$
$$3x + y + 6z = 5$$

- Matrix mapping $Mx = b$:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- Linear combination:

$$x \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + y \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + z \begin{bmatrix} 1 \\ 3 \\ 6 \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Inverse of a matrix

- The inverse M^{-1} is defined such that:

$$MM^{-1} = I \quad \text{and} \quad M^{-1}M = I$$

- Use the inverse to solve a set of linear equations:

$$\begin{aligned} M\boldsymbol{x} &= \boldsymbol{b} \\ M^{-1}M\boldsymbol{x} &= M^{-1}\boldsymbol{b} \\ I\boldsymbol{x} &= M^{-1}\boldsymbol{b} \\ \boldsymbol{x} &= M^{-1}\boldsymbol{b} \end{aligned}$$

How to calculate the inverse?

- The inverse of an $N \times N$ matrix can be calculated using the co-factors of each element of the matrix:

$$M^{-1} = \frac{1}{\det|M|} \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix}^T$$

- $\det|M|$ is the *determinant* of matrix M.
- C_{ij} is the *co-factor* of the ij^{th} element in M.

Computing the co-factors

Consider the following example matrix: $M = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}$

Computing the c

Co ov

A

an n question, multiplied by ± 1, depending on the position.

$$\begin{bmatrix} 1 & & \\ \times & 1 & 3 \\ \times & 1 & 6 \end{bmatrix}$$

Computing the c

Co ov

A

an n the position.

$$\begin{bmatrix} 1 & & \\ \times & 1 & 3 \\ \times & 1 & 6 \end{bmatrix} \qquad \begin{bmatrix} + & - & + \\ - & + & - \\ + & - & + \end{bmatrix}$$

Computing the c

Co ov

A

an n

$$\begin{bmatrix} 1 & & \\ \times & 1 & 3 \\ \times & 1 & 6 \end{bmatrix} \qquad \begin{bmatrix} + & - & + \\ - & + & - \\ + & - & + \end{bmatrix}$$

$$C_{11} = +1 \cdot \det \begin{vmatrix} 1 & 3 \\ 1 & 6 \end{vmatrix}$$

$$= 6 \times 1 - 3 \times 1 = 3$$

Computing the co-factors

Back to our example:

$$M^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}^{-1} = \frac{1}{\det|M|} \begin{bmatrix} 3 & -3 & -1 \\ -5 & 3 & 2 \\ 2 & -1 & -1 \end{bmatrix}^{T}$$

Computing the co-factors

Back to our example:

$$M^{-1} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}^{-1} = \frac{1}{\det\left|M\right|} \begin{bmatrix} 3 & -3 & -1 \\ -5 & 3 & 2 \\ 2 & -1 & -1 \end{bmatrix}^{T}$$

- The determinant is very important
- If $\det\left|M\right| = 0$, the inverse does not exist (singular matrix)

Calculatin

Compute cation of each element on a row (or column) by its cofactor a

$$\det\left|\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}\right| = +\det\left|\begin{bmatrix} 1 & 3 \\ 1 & 6 \end{bmatrix}\right| - \det\left|\begin{bmatrix} 2 & 3 \\ 3 & 6 \end{bmatrix}\right| + \det\left|\begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix}\right| = -1$$

Calculatin

Compute cation of each element on a row (or column) by its cofactor a

$$\det\left|\begin{bmatrix} 1 & 1 & \\ 2 & 1 & \\ 3 & 1 & \end{bmatrix}\right| \quad \begin{matrix} 2 & 3 \\ 3 & 6 \end{matrix}\Bigg|\Bigg| + \det\left|\begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix}\right| = -1$$

$$\det\left|\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix}\right| = +\det\left|\begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix}\right| - 3\det\left|\begin{bmatrix} 1 & 1 \\ 3 & 1 \end{bmatrix}\right| + 6\det\left|\begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix}\right| = -1$$

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Solving a linear system

- Our example:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

Solving a linear system

- Our example:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- The solution is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M^{-1} b = \frac{1}{-1} \begin{bmatrix} 3 & -5 & 2 \\ -3 & 3 & -1 \\ -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix} = \frac{1}{-1} \begin{bmatrix} -13 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ -4 \\ -5 \end{bmatrix}$$

Solving a linear system

- Our example:

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 1 & 3 \\ 3 & 1 & 6 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix}$$

- The solution is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = M^{-1} b = \frac{1}{-1} \begin{bmatrix} 3 & -5 & 2 \\ -3 & 3 & -1 \\ -1 & 2 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 7 \\ 5 \end{bmatrix} = \frac{1}{-1} \begin{bmatrix} -13 \\ 4 \\ 5 \end{bmatrix} = \begin{bmatrix} 13 \\ -4 \\ -5 \end{bmatrix}$$

- The inverse exists, because $\det \left| M \right| = -1$.

Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

- Get the matrix inverse:

```
>> Ainv = inv(A);
```

Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

- Get the matrix inverse:

```
>> Ainv = inv(A);
```

- Compute the solution:

```
>> x = Ainv * b
```

Solving a linear system in Matlab using the inverse

- Create the matrix:

```
>> A = [1 1 1; 2 1 3; 3 1 6];
```

- Create solution vector:

```
>> b = [4; 7; 5];
```

- Get the matrix inverse:

```
>> Ainv = inv(A);
```

- Compute the solution:

```
>> x = Ainv * b
```

- Matlab's internal direct solver:

```
>> x = A\b
```

- These are black boxes! We are going over some methods later!

Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes $N \times N$. Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size N.

Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes $N \times N$. Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size N.

```
% Generate random matrices of various sizes 's'.
% Invert the matrices and store the time required
% for the inversion. Plot the times vs 's'
s = [10:10:90 100:100:1000 2000:1000:5000 10000]
for n = 1:length(s)
```

Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes $N \times N$. Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size N.

```
% Generate random matrices of various sizes 's'.
% Invert the matrices and store the time required
% for the inversion. Plot the times vs 's'
s = [10:10:90 100:100:1000 2000:1000:5000 10000]
for n = 1:length(s)
    s(n)
    A = rand(s(n));
```

Exercise: performance of inverse computation

Create a script that generates matrices with random elements of various sizes $N \times N$. Compute the inverse of each matrix, and use `tic` and `toc` to see the computing time for each inversion. Plot the time as a function of the matrix size N.

```
% Generate random matrices of various sizes 's'.
% Invert the matrices and store the time required
% for the inversion. Plot the times vs 's'
s = [10:10:90 100:100:1000 2000:1000:5000 10000]
for n = 1:length(s)
    s(n)
    A = rand(s(n));
    tic;
    Ainv = inv(A);
    t_inv(n) = toc;
end
loglog(s,t_inv)
xlabel('N')
ylabel('Time [s]')
```

Exercise: sample results

Each computer produces slightly different results because of background tasks, different matrices, etc. This is especially noticable for small systems.

The time increases by 3 orders of magnitude, for every magnitude in N. A matrix inversion scales with $\mathcal{O}(N^3)$!

Solving a linear s

$$Ax = b \qquad \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

- Create matrix A in 3×3 cell
- Create right hand side vector b in 3 vertical cells

[3] In Dutch Excel: `INVERSEMAT`

[4] In Dutch Excel: `PRODUCTMAT`. The semicolon may be a comma.

Solving a linear s

$$Ax = b \qquad \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

- Create matrix A i
- Create right hand s
- Compute the invers
  - Select an empty
  - Type `=MINVERSE( B2:D4 )` [3]
  - Close with Ctrl+Shift+Enter

[3] In Dutch Excel: `INVERSEMAT`

[4] In Dutch Excel: `PRODUCTMAT`. The semicolon may be a comma.

Solving a linear s

$$Ax = b \qquad \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

- Create matrix A i
- Create right hand s
- Compute the invers
  - Select an empty
  - Type `=MINVERSE( B2:D4 )` [3]
  - Close with C
- Solution:
  - Select 3 vert
  - Type `=MMULT( H2:J4 ; B6:B8 )` [4]
  - Close with Ctrl+Shift+Enter

[3] In Dutch Excel: `INVERSEMAT`

[4] In Dutch Excel: `PRODUCTMAT`. The semicolon may be a comma.

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Towards larger systems

Computation of determinants and inverses of large matrices in this way is too difficult (slow), so we need other methods to solve large linear systems!

Towards larger systems

- Determinant of upper triangular matrix:

$$\det\left|M_{\text{tri}}\right| = \prod_{i=1}^{n} a_{ii} \qquad M = \begin{bmatrix} 5 & 3 & 2 \\ 0 & 9 & 1 \\ 0 & 0 & 1 \end{bmatrix} \Rightarrow \det\left|M\right| = 5 \times 9 \times 1 = 45$$

- Matrix multiplication:

$$\det\left|AM\right| = \det\left|A\right| \times \det\left|M\right|$$

- When A is an identity matrix ($\det\left|A\right| = 1$):

$$\det\left|AM\right| = \det\left|A\right| \times \det\left|M\right| = 1 \times \det\left|M\right|$$

- With rules like this, we can use row-operations so that we can compute the determinant more cheaply.

Solutions of linear systems

Rank of a matrix: the number of linearly independent colum
expressed as a linear combination of the other columns) of a

$$M = \begin{bmatrix} 5 & 3 & 2 \\ 0 & 9 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

- 3 independent columns
- In Matlab:

```
>> rank(M)
```

-
- col 4 = col 3 − col 1
- 2 independent columns: rank = 2

Solutions of linear systems

The solution of a system of linear equations may or may not exist, and it may or may not be unique. Existence of solutions can be determined by comparing the rank of the Matrix M with the rank of the augmented matrix M_a:

```
>> rank(A)
>> rank([A b])
```

Our system: $Mx = b$

$$M = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}, b = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} M_{11} & M_{12} & M_{13} & b_1 \\ M_{21} & M_{22} & M_{23} & b_2 \\ M_{31} & M_{32} & M_{33} & b_3 \end{bmatrix}$$

Existence of solutions for linear systems

For a matrix M of size $n \times n$, and augmented matrix M_a:

- $\text{Rank}(M) = n$:
  Unique solution

Existence of solutions for linear systems

For a matrix M of size $n \times n$, and augmented matrix M_a:

- $\text{Rank}(M) = n$:
  Unique solution

- $\text{Rank}(M) = \text{Rank}(M_a) < n$:
  Infinite number of solutions

Existence of solutions for linear systems

For a matrix M of size $n \times n$, and augmented matrix M_a:

- $\text{Rank}(M) = n$:
  Unique solution

- $\text{Rank}(M) = \text{Rank}(M_a) < n$:
  Infinite number of solutions

Two examples

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 4 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 2 & 4 \end{bmatrix}$$

$\text{rank}(M) = 3 = n \Rightarrow$ Unique solution

Two examples

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 2 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 4 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 2 & 4 \end{bmatrix}$$

$\text{rank}(M) = 3 = n \Rightarrow$ Unique solution

$$M = \begin{bmatrix} 1 & 1 & 2 \\ 0 & 3 & 1 \\ 0 & 0 & 0 \end{bmatrix} \quad b = \begin{bmatrix} 17 \\ 11 \\ 0 \end{bmatrix} \Rightarrow M_a = \begin{bmatrix} 1 & 1 & 2 & 17 \\ 0 & 3 & 1 & 11 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$\text{rank}(M) = \text{rank}(M_a) = 2 < n \Rightarrow$ Infinite number of solutions

Today's outline

- Introduction
- Matrix inversion
- Solving a linear system
- Towards larger systems
- Summary

Summary

- Linear equations can be written as matrices
- Using the inverse, the solution can be determined
  - Inverse via cofactors
  - Inverse and solution in Matlab
  - Inverse and solution in Excel
- Inversion scales with N^3
- A solution depends on the rank of a matrix

Linear equations 2

Direct methods

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Overview

Goals

Today we are going to write a program, which can solve a set of linear equations

- The first method is called Gaussian elimination
- We will encounter some problems with Gaussian elimination
- Then LU decomposition will be introduced

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Define the linear system

Consider the system:

$$Ax = b$$

In general:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Desired solution:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1' \\ b_2' \\ b_3' \end{bmatrix}$$

Using row operations

the syst
om row 2
t row, an

$$\left[\begin{array}{ccc|c} A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right]$$

Using

El A_{21}/A_{11}

$$\left[\begin{array}{ccc|c} A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right]$$

Using

El A_{21}/A_{11}

$$\left[\begin{array}{ccc|c} A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right]$$

- $d_{21} \to A_{21}/A_{11}$
- $A_{21} \to A_{21} - A_{11}\,d_{21}$
- $A_{22} \to A_{22} - A_{12}\,d_{21}$
- $A_{23} \to A_{23} - A_{13}\,d_{21}$
- $b_2 \to b_2 - b_1\,d_{21}$

Using

El A_{21}/A_{11}

$$\left[\begin{array}{ccc|c} A_{21} & A_{22} & A_{23} & b_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} 0 & A'_{22} & A'_{23} & b'_2 \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right]$$

- $d_{21} \to A_{21}/A_{11}$
- $A_{21} \to A_{21} - A_{11}d_{21}$
- $A_{22} \to A_{22} - A_{12}d_{21}$
- $A_{23} \to A_{23} - A_{13}d_{21}$
- $b_2 \to b_2 - b_1 d_{21}$

```
d21 = A(2,1)/A(1,1);
A(2,1) = A(2,1) - A(1,1)*d21;
A(2,2) = A(2,2) - A(1,2)*d21;
A(2,3) = A(2,3) - A(1,3)*d21;
b(2) = b(2) - b(1)*d21;
```

Using

El A_{31}/A_{11}

$$\left[\begin{array}{ccc|c} & & & \\ & & & \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} & & & \\ & & & \\ 0 & A'_{32} & A'_{33} & b'_3 \end{array}\right]$$

Using

El A_{31}/A_{11}

$$\left[\begin{array}{ccc|c} & & & \\ & & & \\ A_{31} & A_{32} & A_{33} & b_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} & & & \\ & & & \\ 0 & A'_{32} & A'_{33} & b'_3 \end{array}\right]$$

- $d_{31} \rightarrow A_{31}/A_{11}$
- $A_{31} \rightarrow A_{31} - A_{11}d_{31}$
- $A_{32} \rightarrow A_{32} - A_{12}d_{31}$
- $A_{33} \rightarrow A_{33} - A_{13}d_{31}$
- $b_3 \rightarrow b_3 - b_1 d_{31}$

```
d31 = A(3,1)/A(1,1);
A(3,1) = A(3,1) - A(1,1)*d31;
A(3,2) = A(3,2) - A(1,2)*d31;
A(3,3) = A(3,3) - A(1,3)*d31;
b(3) = b(3) - b(1)*d31;
```

Using

Elin A_{32}/A'_{22}

pivo

$$\left[\begin{array}{ccc|c} & & & \\ 0 & A''_{32} & A''_{33} & b''_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} & & & \\ 0 & 0'' & A'''_{33} & b'''_3 \end{array}\right]$$

Using

Elin A_{32}/A'_{22}

pivo

$$\left[\begin{array}{ccc|c} & & & \\ 0 & \bar{A}_{32} & \bar{A}_{33} & \bar{b}_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} & & & \\ 0 & \bar{\bar{0}} & \bar{\bar{A}}''_{33} & \bar{\bar{b}}''_3 \end{array}\right]$$

- $d_{32} \to A_{32}/A'_{22}$
- $A_{32} \to A_{32} - A'_{22} d_{32}$
- $A_{33} \to A_{33} - A'_{23} d_{32}$
- $b_3 \to b_3 - b'_2 d_{32}$

```
d32 = A(3,2)/A(2,2);
A(3,2) = A(3,1) - A(2,2)*d32;
A(3,3) = A(3,2) - A(2,3)*d32;
b(3) = b(3) - b(2)*d32;
```

Using

Elin A_{32}/A'_{22}
pivo

$$\left[\begin{array}{ccc|c} & & & \\ 0 & A''_{32} & A''_{33} & b''_3 \end{array}\right] \longrightarrow \left[\begin{array}{ccc|c} & & & \\ 0 & 0 & A'''_{33} & b'''_3 \end{array}\right]$$

- $d_{32} \to A_{32}/A'_{22}$
- $A_{32} \to A_{32} - A'_{22} d_{32}$
- $A_{33} \to A_{33} - A'_{23} d_{32}$
- $b_3 \to b_3 - b'_2 d_{32}$

```
d32 = A(3,2)/A(2,2);
A(3,2) = A(3,1) - A(2,2)*d32;
A(3,3) = A(3,2) - A(2,3)*d32;
b(3) = b(3) - b(2)*d32;
```

The matrix is now a triangular matrix — the solution can be obtained by back-substitution.

Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

Start at the last row N, and work upward until row 1.

$$x_3 = b''_3 / A''_{33}$$
$$x_2 = (b'_2 - A'_{23} x_3) / A'_{22}$$
$$x_1 = (b_1 - A_{12} x_2 - A_{13} x_3) / A_{11}$$

Backsubstitution

The system now reads:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & 0 & A''_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b'_2 \\ b''_3 \end{bmatrix}$$

Start at the last row N, and work upward until row 1.

$$x_3 = b''_3 / A''_{33}$$
$$x_2 = (b'_2 - A'_{23} x_3) / A'_{22}$$
$$x_1 = (b_1 - A_{12} x_2 - A_{13} x_3) / A_{11}$$

```
x(3) = b(3) / A(3,3)
x(2) = (b(2) - A(2,3)*x(3)) /A(2,2)
x(1) = (b(1) - A(1,2)*x(2) - A(1,3)*x(3)) / A
     (1,1)
```

In general:

$$x_N = \frac{b_N}{A_{NN}} \qquad x_i = \frac{b_i - \sum_{j=i+1}^{N} A_{ij} x_j}{A_{ii}}$$

Writing the program

- Create a function that provides the framework: take matrix A and vector b as an input, and return the solution x as output:

```
function [x,A,b] = GaussianEliminate(A,b)
```

- We will use *for-loops* instead of typing out each command line.
- Useful Matlab shortcuts:
  - `A(1,:)` $= [A_{11}, A_{12}, A_{13}]$
  - `A(:,2)` $= [A_{12}, A_{22}, A_{32}]^T$
  - `A(1,2:end)` $= [A_{12}, A_{13}]$
- A row operation could look like:

```
A(i,:) = A(i,:) - d*A(1,:)
```

The program: elimination

```
function [x,A,b] = GaussianEliminate(A,b)

% Perform elimination to obtain an upper triangular matrix
N = length(b);
for column=1:(N-1) % Select pivot
    for row=(column+1):N % Loop over subsequent rows (below pivot)
        d=A(row,column)/A(column,column);
        A(row,:)=A(row,:)-d*A(column,:);
        b(row)= b(row)-d*b(column);
    end
end
```

The program: Backsubstitution

```
% Assign b to x
x=b;

% Perform backsubstitution
for row=N:-1:1
    x(row) = b(row);
    for i =(row+1):N
        x(row)=x(row)-A(row,i)*x(i);
    end
    x(row)=x(row)/A(row,row);
end
```

$$x_N = \frac{b_N}{A_{NN}} \qquad x_i = \frac{b_i - \sum_{j=i+1}^{N} A_{ij} x_j}{A_{ii}}$$

Exercise: Gaussian Elimination

- The function we just made can be found on Canvas
- Use `help GaussianEliminate` to find out how it works
- Solve the following system of equations:

$$\begin{bmatrix} 9 & 9 & 5 & 2 \\ 6 & 7 & 1 & 3 \\ 6 & 4 & 3 & 5 \\ 2 & 6 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 7 \\ 4 \\ 10 \\ 1 \end{bmatrix}$$

- Compare your solution with `A\b`

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Partial pivoting

- Now try to run the algorithm with the following system:

$$\begin{bmatrix} 0 & 2 & 1 \\ 3 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 10 \end{bmatrix}$$

Partial pivoting

- Now try to run the algorithm with the following system:

$$\begin{bmatrix} 0 & 2 & 1 \\ 3 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 10 \end{bmatrix}$$

- It does not work! Division by zero, due to $A_{11} = 0$.
- Solution: Swap rows to move largest element to the diagonal.

Partial

- Find maximum element row below pivot in current column

```
[dummy,index] = max(abs(A(column:end,column)));
Index = index+column-1;
```

Partial

- elow pivot in current column
- Store current row

```
[dummy,index] = max(abs(A(column:end,column)));
Index = index+column-1;
```

```
temp = A(column,:);
```

Partial

- nt column
- Swap pivot row and desired row in A

```
[dummy,index] = max(abs(A(column:end,column)));
Index = index+column-1;
```

```
temp = A(column,:);
```

```
A(column,:) = A(index,:);
A(index,:) = temp;
```

Partial

- nt column
- Do the same for b: store and swap

```
[dummy,index] = max(abs(A(column:end,column)));
Index = index+column-1;
```

```
temp = A(column,:);
```

```
A(column,:) = A(index,:);
A(index,:) = temp;
```

```
temp = b(column);
b(column) = b(index);
b(index) = temp;
```

Improve the program by using re-usable functions

```
function [x] = GaussianEliminate(A,b)
% GaussianEliminate(A,b): solves x in Ax=b
N = length(b);
for c=1:(N-1)
    [dummy,index]=max(abs(A(c:end,c)));
    index=index+c-1;
    A = SWAP(A,c,index); % Created swap function
    b = SWAP(b,c,index);
    for row=(column+1):N
        d=A(row,column)/A(column,column);
        A(row,:)=A(row,:)-d*A(column,:);
        b(row)= b(row)-d*b(column);
    end
end
x = backwardSub(A,b); % Created BS function
return
```

This function is also provided (named GaussianEliminate_v2 and GaussianEliminate_v3 on Canvas).

Alternatives to this program

- MATLAB can compute the solution to Ax=b with its own solvers (more efficient) `A\b`
- Too many loops. Loops make MATLAB slow.
- There are fundamental problems with Gaussian elimination

Alternatives to this program

- MATLAB can compute the solution to Ax=b with its own solvers (more efficient) `A\b`
- Too many loops. Loops make MATLAB slow.
- There are fundamental problems with Gaussian elimination
  - You can add a counter to the algorithm to see how many subtraction and multiplication operations it performs for a given size of matrix A.
  - The number of operations to perform Gaussian elimination is $\mathcal{O}(2N^3)$ (where N is the number of equations)
  - Exercise: verify this for our script

Alternatives to this program

- MATLAB can compute the solution to Ax=b with its own solvers (more efficient) `A\b`
- Too many loops. Loops make MATLAB slow.
- There are fundamental problems with Gaussian elimination
  - You can add a counter to the algorithm to see how many subtraction and multiplication operations it performs for a given size of matrix A.
  - The number of operations to perform Gaussian elimination is $\mathcal{O}(2N^3)$ (where N is the number of equations)
  - Exercise: verify this for our script
  - LU decomposition takes $\mathcal{O}(2N^3/3)$ flops, 3 times less!
  - Forward and backward substitution each take $\mathcal{O}(N^2)$ flops (both cases)

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

LU Decomposition

Suppose we want to solve the previous set of equations, but with several right hand sides:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ x_1 & x_2 & x_3 \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ b_1 & b_2 & b_3 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

LU Decomposition

Suppose we want to solve the previous set of equations, but with several right hand sides:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ x_1 & x_2 & x_3 \\ \vdots & \vdots & \vdots \end{bmatrix} = \begin{bmatrix} \vdots & \vdots & \vdots \\ b_1 & b_2 & b_3 \\ \vdots & \vdots & \vdots \end{bmatrix}$$

Factor the matrix A into two matrices, L and U, such that $A = LU$:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \times & 1 & 0 \\ \times & \times & 1 \end{bmatrix} \begin{bmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & \times \end{bmatrix}$$

Now we can solve for each right hand side, using only a forward followed by a backward substitution!

Substitutions

- Define a lower and upper matrix L and U so that $A = LU$
- Therefore $LUx = b$
- Define a new vector $y = Ux$ so that $Ly = b$
- Solve for y, use L and forward substitution
- Then we have y, solve for x, use $Ux = y$
- Solve for x, use U and backward substitution
- But how to get L and U?

Decomposing the matrix (1)

When we eliminate the element A_{21} we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to A.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} - d_{21}A_{12} & A_{23} - d_{21}A_{13} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

Decomposing the matrix (2)

When we eliminate the element A_{31} we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to A.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} = A_{22} - d_{21}A_{12} & A'_{23} = A_{23} - d_{21}A_{13} \\ 0 & A'_{32} = A_{32} - d_{31}A_{12} & A'_{33} = A_{33} - d_{31}A_{21} \end{bmatrix}$$

Decomposing the matrix (3)

When we eliminate the element A_{32} we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to A.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & d_{32} & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A''_{33} = A'_{33} - d_{32}A'_{23} \end{bmatrix}$$

Decomposing the matrix (3)

When we eliminate the element A_{32} we can keep multiplying by a matrix that undoes this row operations, so that the product remains equal to A.

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & d_{32} & 1 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A''_{33} = A'_{33} - d_{32}A'_{23} \end{bmatrix}$$

This finishes the LU decomposition!

Pivoting during decomposition

Suppose we have arrived at the situation bel

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & & \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Pivoting during decomposition

Suppose we have arrived at the situation bel

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & & \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Exchange rows 2 and 3 to get the largest value on the main diagonal. Use a permutation matrix to store the swapped rows:

Pivoting during decomposition

Suppose we have arrived at the situation bel

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & & \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Exchange rows 2 and 3 to get the largest value on the main diagonal. Use a permutation matri rows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} & & \\ & & \\ d_{21} & 1 & 0 \end{bmatrix} \begin{bmatrix} & & \\ 0 & A'_{22} & A'_{23} \end{bmatrix}$$

Pivoting during decomposition

Suppose we have arrived at the situation bel

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ d_{21} & 1 & 0 \\ d_{31} & 0 & 1 \end{bmatrix} \begin{bmatrix} A_{11} & & \\ 0 & A'_{22} & A'_{23} \\ 0 & A'_{32} & A'_{33} \end{bmatrix}$$

Exchange rows 2 and 3 to get the largest value on the main diagonal. Use a permutation matri rows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} & & \\ & & \\ d_{21} & 1 & 0 \end{bmatrix} \begin{bmatrix} & & \\ 0 & A'_{22} & A'_{23} \end{bmatrix}$$

Multiplying with a permutation matrix will swap the rows of a matrix. The permutation matrix is just an identity matrix, whose rows have been interchanged.

Recipe for LU decomposition

When decomposing matrix A into $A = LU$, it may be beneficial to swap rows to get the largest values on the diagonal of U (pivoting). A permutation matrix P is used to store row swapping such that:

$$PA = LU$$

- Write down a permutation matrix and the linear system
- Promote the largest value in the column diagonal
- Eliminate all elements below diagonal
- Move on to the next column and move largest elements to diagonal
- Eliminate elements below diagonal
- Repeat 5 and 6
- Write down L,U and P

LU decomposition example (1)

Write down a permutation matrix, which starts as the identity matrix, and the linear system:

$$PA = LU$$

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

LU decomposition example (1)

Write down a permutation matrix, which starts as the identity matrix, and the linear system:

$$PA = LU$$

$$\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$$

Prom [redacted] nto the diag [redacted]

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 2 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix}$$

LU decompo

Eliminate all elements below 3
= row 3 - 0.5 row 1. Record

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} & & \\ & & \\ 0.5 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1.5 & -0.5 \end{bmatrix}$$

LU decompo

Eliminate all elements below
= row 3 - 0.5 row 1. Record

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 1 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 0.5 & 0 & 1 \end{bmatrix}$$

Elimination of column 1 is done. Step t
below/on the diagonal to the diagonal (swap rows 2 and 3). Adjust P and lower triangle of L
accor

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1.5 & -0.5 \\ 0 & 1 & 1 \end{bmatrix}$$

LU decom

Eliminate all elements below the d
row 3 = row 3 - $\frac{2}{3}$row 2. Record t

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & & \\ 0.5 & & \\ 0 & \frac{2}{3} & 1 \end{bmatrix} \begin{bmatrix} 2 & & \\ 0 & & \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

LU decom

Eliminate all elements below the d
row 3 = row 3 - $\frac{2}{3}$row 2. Record t

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 & 1 \\ 2 & 1 & 0 \\ 1 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 1 & & \\ 0.5 & & \\ 0 & \frac{2}{3} & 1 \end{bmatrix}\begin{bmatrix} 2 & & \\ 0 & & \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

We have obtained the matrices from $PA = LU$:

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \quad L = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0 & \frac{2}{3} & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 1 & 1 \\ 0 & 1.5 & -0.5 \\ 0 & 0 & \frac{4}{3} \end{bmatrix}$$

Proceed with solving for x.

Substitutions

$$Ax = b \quad \Rightarrow \quad PAx = Pb \equiv d$$
$$PA = LU \quad \Rightarrow \quad LUx = d$$

- Define a new vector $y = Ux$
  - $Ly = b \quad \Rightarrow \quad Ly = d$
  - Solve for y, forward substitution:

$$y_1 = \frac{d_1}{L_{11}}$$
$$y_i = \frac{d_i - \sum_{j=1}^{i-1} L_{ij} y_j}{L_{ii}}$$

- Then solve $Ux = y$:
  - Solve for x, backward substitution:

$$x_N = \frac{y_N}{U_{NN}}$$
$$x_i = \frac{y_i - \sum_{j=i+1}^{N-1} U_{ij} x_j}{U_{ii}}$$

How to use the solver in Matlab

```
A = rand(5,5);            % Get random matrix
[L, U, P] = lu(A);        % Get L, U and P
b = rand(5,1);            % Random b vector
d = P*b;                  % Permute b vector
y = forwardSub(L,d);      % Can also do y=L\d
x = backwardSub(U,y);     % Can also do x=U\y
rnorm = norm(A*x - b);    % Residual

% Compare results to internal Matlab solver
x = A\b
```

How to use the solver in Matlab

```
A = rand(5,5);              % Get random matrix
[L, U, P] = lu(A);          % Get L, U and P
b = rand(5,1);              % Random b vector
d = P*b;                    % Permute b vector
y = forwardSub(L,d);        % Can also do y=L\d
x = backwardSub(U,y);       % Can also do x=U\y
rnorm = norm(A*x - b);      % Residual

% Compare results to internal Matlab solver
x = A\b
```

- Use this as a basis to create a function that takes A and b, and returns x.
- Use the function to check the performance for various matrix sizes and inspect the performance.

Today's outline

- Introduction
- Gauss elimination
- Partial Pivoting
- LU decomposition
- Summary

Summary

- This lecture covered direct methods using elimination techniques.
- Gaussian elimination can be slow ($\mathcal{O}(N^3)$)
- Back substitution is often faster ($\mathcal{O}(N^2)$)
- LU decomposition means that we don't have to do Gaussian elimination every time (saves time and effort), but the matrix has to stay the same.
- Matlab has build in routines for solving linear equations (backslash operator \) and LU decomposition (`lu`).
- Advanced techniques such as (preconditioned) conjugate gradient or biconjugate gradient solvers are also available.
- Next part covers iterative approaches

Linear equations 3

Iterative methods

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Sparse matrices

- In many engineering cases, we deal with sparse matrices (as opposed to dense matrices)
- A matrix is sparse when it mostly consists of zeros
- Linear systems where equations depend on a limited number of variables (e.g. spatial discretization)
- Storing zeros is not very efficient:

```
>> A = eye(10000);
>> whos A
>> S = sparse(A);
>> whos S
```

- Can you think of a way to achieve this?
- Sparse matrix formats: Yale, CRS, CCS

Sparse matrix storage format

- Example: Yale storage format, storing 3 vectors:
  - `A = [5 8 3 6]`
  - `IA = [0 0 2 3 4]`
  - `JA = [0 1 2 1]`
- `A` stores the non-zero values
- `IA` stores the index in A of the first non-zero in row i
- `JA` stores the column index
- Note: zero-based indices are used here!

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 5 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{bmatrix}$$

Sparse matrix layout examples

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Laplace's equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T =$ Temperature

$\alpha =$ Thermal diffusivity

Laplace's equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T =$ Temperature

$\alpha =$ Thermal diffusivity

Laplace's equation

$$\frac{\partial T}{\partial t} = \alpha \nabla^2 T$$

$T =$ Temperature

$\alpha =$ Thermal diffusivity

In steady state:

$$\nabla^2 T = 0$$

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$$

Discretization of Laplace's equation (I)

- Define a grid of points in x and y

Discretization of Laplace's equation (I)

- Define a grid of points in x and y
- Index of the grid points using 2D coordinates i and j

Discretization of Laplace's equation (I)

- Define a grid of points in x and y
- Index of the grid points using 2D coordinates i and j
- Set up the equations using a 1D index system: $T_{i,j} = T_{i+N_x(j-1)}$

Discretization of Laplace's equation (II)

Estimate the second-order differentials: assume a piece-wise linear profile in the temperature:

Discretization of Laplace's equation (II)

Estimate the second-order differentials: assume a piece-wise linear profile in the temperature:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{\left.\frac{\partial T}{\partial x}\right|_{i+\frac{1}{2}} - \left.\frac{\partial T}{\partial x}\right|_{i-\frac{1}{2}}}{\Delta x}$$

$$\approx \frac{\frac{\left(T_{i+1,j}-T_{i,j}\right)}{\Delta x} - \frac{\left(T_{i,j}-T_{i-1,j}\right)}{\Delta x}}{\Delta x}$$

$$= \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2}$$

Discretization of Laplace's equation (III)

The y-direction is derived analogously, so that the 2D Laplace's equation is discretized as:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

Discretization of Laplace's equation (III)

The y-direction is derived analogously, so that the 2D Laplace's equation is discretized as:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

Use a single index counter $k = i + N_x(j-1)$, so that the equation becomes:

$$\frac{T_{k+1} - 2T_k + T_{k-1}}{(\Delta x)^2} + \frac{T_{k+N_x} - 2T_k + T_{k-N_x}}{(\Delta y)^2} = 0$$

Discretization of Laplace's equation (III)

The y-direction is derived analogously, so that the 2D Laplace's equation is discretized as:

$$\frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta x)^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{(\Delta y)^2} = 0$$

Use a single index counter $k = i + N_x(j-1)$, so that the equation becomes:

$$\frac{T_{k+1} - 2T_k + T_{k-1}}{(\Delta x)^2} + \frac{T_{k+N_x} - 2T_k + T_{k-N_x}}{(\Delta y)^2} = 0$$

For an equal spaced grid $\Delta x = \Delta y = 1$:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

$$\Rightarrow AT = b$$

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Creating the linear system

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

Create a *banded* matrix A: the main diagonal k contains -4, whereas the bands at $k-1$, $k+1$, $k-N_x$ and $k+N_x$ contain a 1. Boundary cells just contain a 1 on the main diagonal so that the temperature is equal to T_b (e.g. $T_1 = 1T_b$).

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \cdots & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \ddots & 0 \\ 0 & \cdots & 1 & \cdots & 1 & -4 & 1 & \cdots & 1 & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_k \\ T_k + 1 \\ \vdots \\ T_{(N_y-1)N_x} \\ T_{(N_y-1)N_x+1} \end{bmatrix} = \begin{bmatrix} T_b \\ T_b \\ \vdots \\ 0 \\ 0 \\ \vdots \\ T_b \\ T_b \end{bmatrix}$$

Creating the linear system

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

Create a *banded* matrix A in Matlab, by setting the coefficients for the internal cells:

```
Nx=5;  %number of points along x direction
Ny=5;  %number of points in the y direction
Nc=Nx*Ny; % Total number of points

e = ones(Nc,1);
A = spdiags([e,e,-4*e,e,e],[-Nx,-1,0,1,Nx],Nc,Nc);
b = zeros(Nc,1);
```

The function `spdiags` uses the following arguments:

- The coefficients that have to be put on the diagonals arranged as columns in a matrix
- The position of the bands with respect to the main diagonal
- Size of the resulting matrix (in our case square $N_xN_y \times N_xN_y$)

Matrix sparsity

- Let's check the matrix layout:

```
>> spy(A)
```

- This command shows the non-zero values of a matrix
- Apart from the main diagonal, there are offset bands!

About boundary conditions

- For the nodes on the boundary, we have a simple equation:

$$T_{k,\text{boundary}} = \text{Some fixed value}$$

- However, we have set all nodes to be a function of their neighbors...
- Find the boundary node indices using $k = i + Nx(j-1)$
  - `i = 1, j = 1:Ny`
  - `i = Nx, j = 1:Ny`
  - `j = 1, i = 1:Nx`
  - `j = Ny, i = 1:Nx`
- Reset the row in A to zeros, set $A_{kk} = 1$
- Set value in rhs: $b_k = T_{k,\text{boundary}}$
- Boundary conditions are often more elaborate to implement! See `setBoundaryConditions.m`.

Partial implementation of the boundary conditions

See `setBoundaryConditions.m`.

```
function [A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny)

% Set boundary conditions over x-direction
for i=1:Nx
    j = 1;
    ind = i + Nx * (j-1);
    A(ind,:) = 0;    % Reset matrix for boudary cells
    A(ind,ind) = 1; % Add a 1 on the diagonal
    b(ind) = Tb(1);
    j = Ny;
    ind = i + Nx * (j-1);
    A(ind,:) = 0;    % Reset matrix for boudary cells
    A(ind,ind) = 1; % Add a 1 on the diagonal
    b(ind) = Tb(2);
end

%% Repeat for y-direction
```

How applying boundary conditions affects the linear system

```
function [A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny)
```

How applying boundary conditions affects the linear system

```
function [A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny)
```

- Make sure that matrix `A` and right hand side vector `b` are in your workspace, as well as `Nx` and `N_y`
- Create a vector that holds the temperature at each boundary:

```
>> T = [10 20 30 40];
```

- Call the function, store A and b in new variables:

```
>> [A2,b2] = setBoundaryConditions(A,b,T,Nx,Ny);
```

- Check the new structure of the matrix and the right hand side:

```
>> subplot(1,2,1); spy(A2);
>> subplot(1,2,2); spy(b2);
```

A full program, including solver

The program and auxiliary functions are on Canvas (`solveLaplaceEq.m`)

```
function [x,y,T,A] = solveLaplaceEq(Nx,Ny)
% Solves the steady-state Laplace equation

Tb = [10 20 30 40]; % Fixed boundary temperatures

% Fill sparse matrix with [1 1 -4 1 1]
e = ones(Nx*Ny,1);
A = spdiags([e,e,-4*e,e,e],[-Nx,-1,0,1,Nx],Nx*Ny,Nx*Ny);
b = zeros(Nx*Ny,1);

[A,b] = setBoundaryConditions(A,b,Tb,Nx,Ny);

T = A\b;  % Solve matrix
Tc = reshape(T,[Nx,Ny]); % Reshape x-vec to mat Nx,Ny
[xc yc] = meshgrid(1:Nx,1:Ny); % Get position arrays
surf(xc,yc,Tc); % Surface plot
```

Sample results

Solved for a 20×20 system with $T_b = [10\ 20\ 30\ 40]$.

LU decomposition of a sparse matrix

```
>> [L,U,P] = lu(A)
>> subplot(1,2,1)
>> spy(L)
>> subplot(1,2,2)
>> spy(U)
```

LU decomposition of a sparse matrix

- With LU decomposition we produce matrices that are less sparse than the original matrix.
- Sparse storage often required, and also numerical techniques that fully utilizes this!

```
>> [L,U,P] = lu(A)
>> subplot(1,2,1)
>> spy(L)
>> subplot(1,2,2)
>> spy(U)
```

LU decomposition

- LU decomposition and Gaussian elimination on a matrix like A requires more memory (with 3D problems, the offset in the diagonal would even be bigger!)
- In general extra memory allocation will not be a problem for MATLAB
- MATLAB is clever, in that sense that it attempts to reorder equations, to move elements closer to the diagonal)

LU decomposition

- LU decomposition and Gaussian elimination on a matrix like A requires more memory (with 3D problems, the offset in the diagonal would even be bigger!)
- In general extra memory allocation will not be a problem for MATLAB
- MATLAB is clever, in that sense that it attempts to reorder equations, to move elements closer to the diagonal)

Alternatives for elimination methods

- Use iterative methods when systems are large and sparse.
- Often such systems are encountered when we want to solve PDEs of higher dimensions

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Examples of iterative methods

- Jacobi method
- Gauss-Seidel method
- Succesive over relaxation

- `bicg` — Bi-conjugate gradient method
- `pcg` — preconditioned conjugate gradient method
- `gmres` — generalized minimum residuals method
- `bicgstab` — Bi-conjugate gradient method

The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  1. Start with an initial guess for the values of T at each node

The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  1. Start with an initial guess for the values of T at each node
  2. Compute updated values and store a new vector:

$$T_k^{\text{new}} = \frac{T_{k-N_x}^{\text{old}} + T_{k-1}^{\text{old}} + T_{k+1}^{\text{old}} + T_{k+N_x}^{\text{old}}}{4}$$

The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  1. Start with an initial guess for the values of T at each node
  2. Compute updated values and store a new vector:

$$T_k^{\text{new}} = \frac{T_{k-N_x}^{\text{old}} + T_{k-1}^{\text{old}} + T_{k+1}^{\text{old}} + T_{k+N_x}^{\text{old}}}{4}$$

  3. Do this for all nodes

The Jacobi method

- In our example we derived the following equation:

$$T_{k-N_x} + T_{k-1} - 4T_k + T_{k+1} + T_{k+N_x} = 0$$

- Rearranging gives:

$$T_k = \frac{T_{k-N_x} + T_{k-1} + T_{k+1} + T_{k+N_x}}{4}$$

- In the Jacobi scheme the iteration proceeds as follows:
  1. Start with an initial guess for the values of T at each node
  2. Compute updated values and store a new vector:

$$T_k^{\text{new}} = \frac{T_{k-N_x}^{\text{old}} + T_{k-1}^{\text{old}} + T_{k+1}^{\text{old}} + T_{k+N_x}^{\text{old}}}{4}$$

  3. Do this for all nodes
  4. Repeat the procedure until converged

Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
```

Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40;  % Left
T(nx,:) = 60; % Right
T(:,1) = 20;  % Bottom
T(:,ny) = 30; % Top
```

Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40;  % Left
T(nx,:) = 60; % Right
T(:,1) = 20;  % Bottom
T(:,ny) = 30; % Top
Tnew = T;
```

Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40;  % Left
T(nx,:) = 60; % Right
T(:,1) = 20;  % Bottom
T(:,ny) = 30; % Top
Tnew = T;
% For plotting
[x y] = meshgrid(1:nx, 1:ny);
```

Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40;  % Left
T(nx,:) = 60; % Right
T(:,1) = 20;  % Bottom
T(:,ny) = 30; % Top
Tnew = T;
% For plotting
[x y] = meshgrid(1:nx, 1:ny);
for iter = 1:1000
  for i = 2:nx-1
    for j = 2:ny-1
      Tnew(i,j) = (T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))/4.0;
    end
  end
```

Jacobi method for Laplace's equation

See `laplace_jacobi.m` (from Canvas)

```
% Grid size
nx = 40; ny = 40;
% The temperature field + boundaries at old and new times
T = zeros(nx,ny);
T(1,:) = 40;  % Left
T(nx,:) = 60; % Right
T(:,1) = 20;  % Bottom
T(:,ny) = 30; % Top
Tnew = T;
% For plotting
[x y] = meshgrid(1:nx, 1:ny);
for iter = 1:1000
  for i = 2:nx-1
    for j = 2:ny-1
      Tnew(i,j) = (T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))/4.0;
    end
  end
  surf(x,y,Tnew);
  title(['Iteration: ' num2str(iter)]);
  drawnow
  T = Tnew; % Update T
end
```

About the straightforward implementation

- The method as implemented works fine for a simple Laplace equation
- For generic systems of linear equations, the implementation cannot be used.

About the straightforward implementation

- The method as implemented works fine for a simple Laplace equation
- For generic systems of linear equations, the implementation cannot be used.

We will now introduce the Jacobi method so it can be used for generic systems of linear equations.

The Jacobi method with matrices

We can split our (banded) matrix A into a diagonal matrix D and a remainder R:

$$A = D + R$$

$$\begin{bmatrix} \times & \times & & & & & \times & & \\ \times & \times & \times & & & & & \times & \\ & \times & \times & \times & & & & & \times \\ & & \times & \times & \times & & & & \\ & & & \times & \times & \times & & & \\ & & & & \times & \times & \times & & \\ \times & & & & & \times & \times & \times & \\ & \times & & & & & \times & \times & \times \\ & & \times & & & & & \times & \times \end{bmatrix} = \begin{bmatrix} \times & & & & & & & & \\ & \times & & & & & & & \\ & & \times & & & & & & \\ & & & \times & & & & & \\ & & & & \times & & & & \\ & & & & & \times & & & \\ & & & & & & \times & & \\ & & & & & & & \times & \\ & & & & & & & & \times \end{bmatrix} + \begin{bmatrix} & \times & & & & & \times & & \\ \times & & \times & & & & & \times & \\ & \times & & \times & & & & & \times \\ & & \times & & \times & & & & \\ & & & \times & & \times & & & \\ & & & & \times & & \times & & \\ \times & & & & & \times & & \times & \\ & \times & & & & & \times & & \times \\ & & \times & & & & & \times & \end{bmatrix}$$

Jacobi method: solving a system

- We can solve $AT = b$, now written generally as $Ax = b$, by:

$$\begin{aligned} Ax &= b \\ (D+R)x &= b \\ Dx &= b - Rx \\ Dx^{\text{new}} &= b - Rx^{\text{old}} \\ x^{\text{new}} &= D^{-1}(b - Rx^{\text{old}}) \end{aligned}$$

- Using the n and $n+1$ notation for old and new time steps, we find in general:

$$x^{n+1} = D^{-1}(b - Rx^n)$$

$$x_i^{n+1} = \frac{1}{A_{ii}}\left(b_i - \sum_{j \neq i} A_{ij} x_j^n\right)$$

Diagram of the Jacobi method

Set T^{old}
$=$ a guess

Diagram of the Jacobi method

Diagram of the Jacobi method

Diagram of the Jacobi method

Diagram of the Jacobi method

Diagram of the Jacobi method

Diagram of the Jacobi method

Diagram of the Jacobi method

The core of the solver

The full file is on Canvas, `solveJacobi.m`.

```
  while ( xDiff > tol && it_jac < 1000 )
    x_old = x;
    for i=1:N
        s = 0;
        for j = 1:N
            if (j ~= i)
                s = s+A(i,j)*x_old(j);
            end
        end
        x(i) = (b(i)-s)/A(i,i);
    end
    it_jac = it_jac+1;
    xDiff = norm((x-x_old)./x,2);
end
it_jac
```

The core of the solver

The full file is on Canvas, `solveJacobi.m`.

```
  while ( xDiff > tol && it_jac < 1000 )
    x_old = x;
    for i=1:N
        s = 0;
        for j = 1:N
            if (j ~= i)
                s = s+A(i,j)*x_old(j);
            end
        end
        x(i) = (b(i)-s)/A(i,i);
    end
    it_jac = it_jac+1;
    xDiff = norm((x-x_old)./x,2);
end
it_jac
```

Try to call it from the `solveLaplaceEq.m` file, instead of using `\`.

A few details on this algorithm

- The while loop holds two aspects
  - A convergence criterion (`norm((x-x_old)./x,2)> tol`). Some considerations are:
    - L_1-norm (sum)
    - L_2-norm (Euclidian distance)
    - L_∞-norm (max)
  - Protection against infinite loops (no convergence)

A few details on this algorithm

- The while loop holds two aspects
  - A convergence criterion (`norm((x-x_old)./x,2)> tol`). Some considerations are:
    - L_1-norm (sum)
    - L_2-norm (Euclidian distance)
    - L_∞-norm (max)
  - Protection against infinite loops (no convergence)
- Reset the sum for each row, before summing for the new unknown node

A few details on this algorithm

- The while loop holds two aspects
  - A convergence criterion (`norm((x-x_old)./x,2)> tol`). Some considerations are:
    - L_1-norm (sum)
    - L_2-norm (Euclidian distance)
    - L_∞-norm (max)
  - Protection against infinite loops (no convergence)
- Reset the sum for each row, before summing for the new unknown node

- Start vector x is not shown in the example, but should be there!
- It can have huge impact on performance!
- The for-loops also have a large performance penalty!

The solver using array indices

Make a copy of the Jacobian solver, and replace the for-loop by a vector-operation:

```
% While not converged or max_it not reached
while ( xDiff > tol && it_jac < 1000 )
  x_old = x;
  for i=1:N
    % Sum off-diagonal*x_old
    offDiagonalIndex = [1:(i-1) (i+1):N];
    Aij_Xj = A(i,offDiagonalIndex)*x_old(offDiagonalIndex);

    % Compute new x value
    x(i) = (b(i)-Aij_Xj)/A(i,i);
  end
  it_jac = it_jac+1;
  xDiff = norm((x-x_old)./x,2);
end
```

Iterations 1, 2, 3 and 10

Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate x^{new} is returned to the solution x^{old} as soon as it is completed
- For following nodes, the updated solution is used immediately

Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate x^{new} is returned to the solution x^{old} as soon as it is completed
- For following nodes, the updated solution is used immediately
- Our straightforward script (from the Jacobi method) is therefore changed easily:
  - Do not create a `Tnew` array (save memory!)
  - Do not store the solution in `Tnew`, but simply in `T`
  - Do not perform the update step `T=Tnew`
  - See `laplace_gaussseidel.m` for the algorithm.

Gauss-Seidel method

The Gauss-Seidel method is quite similar to Jacobi method

- The only difference is that the new estimate x^{new} is returned to the solution x^{old} as soon as it is completed
- For following nodes, the updated solution is used immediately
- Our straightforward script (from the Jacobi method) is therefore changed easily:
  - Do not create a `Tnew` array (save memory!)
  - Do not store the solution in `Tnew`, but simply in `T`
  - Do not perform the update step `T=Tnew`
  - See `laplace_gaussseidel.m` for the algorithm.
- The straightforward script works well for the current Laplace equation, but we define the generic Gauss-Seidel algorithm on the following slides.

Gauss-Seidel method

- Define a lower and strictly upper triangular matrix, such that $A = L + U$
- Now we can solve AT=b by:

$$
\begin{aligned}
(L+U)T &= b \\
LT &= b - UT \\
LT^{\text{new}} &= b - UT^{\text{old}} \\
T^{\text{new}} &= L^{-1}(b - UT^{\text{old}})
\end{aligned}
$$

- Using the n and $n+1$ notation for old and new time steps, we find in for the general Gauss-Seidel method:

$$x^{n+1} = L^{-1}(b - Ux^n)$$

$$x_i^{n+1} = \frac{1}{A_{ii}}\left(b_i - \sum_{j<i} A_{ij}x_j^{n+1} - \sum_{j>i} A_{ij}x_j^n\right)$$

Today's outline

- Introduction
- Sparse matrices
- Laplace's equation
- Creating a sparse system
- Iterative methods
- Summary

Summary

- Partial differential equations can be written as sparse systems of linear equations
- Sparse systems can be handled with a direct method like Gaussian elimination
- If you have systems of more than 1 dimension, a direct method still can be used, if there are no memory issues, otherwise an iterative method may be attractive.
- The Jacobi method was introduced. Many other methods are based on the Jacobi method (SOR method, for example)

Curve fitting, regression and optimization

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Overview

- We are going to fit measurements to models today
- You will also learn what R^2 actually means
- We get introduced to constrained and unconstrained optimization.
- We will use the simplex method to solve linear programming problems (LP)

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Fitting models to data

How to fit a model to the data?

We would like to fit the following model to the data:

$$\hat{y} = a_1 + a_2 x + a_3 x^2 + a_4 x^3$$

How to fit a model to the data?

We would like to fit the following model to the data:

$$\hat{y} = a_1 + a_2 x + a_3 x^2 + a_4 x^3$$

First step: If we have N data points, we could write the model as the product of a matrix and a vector:

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \vdots \\ \hat{y}_N \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ 1 & x_3 & x_3^2 & x_3^3 \\ \vdots & \vdots & \vdots & \\ 1 & x_N & x_N^2 & x_N^3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix}$$

$$\hat{y} = Xa$$

X is called the design matrix and a are the fit parameters.

Residuals

Second step: work out the residuals for each data point:

$$d_i = (y_i - \hat{y}_i)$$

Residuals

Second step: work out the residuals for each data point:

$$d_i = (y_i - \hat{y}_i)$$

Third step: work out the sum of squares of the residuals:

$$\text{SSE} = \sum_i d_i^2 = \sum_i (y_i - \hat{y}_i)$$

This can be written using the dot-product operation:

$$\text{SSE} = \sum_i d_i^2 = d \cdot d = d^T \cdot d = (y_i - \hat{y}_i)^T \cdot (y_i - \hat{y}_i)$$

Minimizing the sum of squares

Choose the parameter vector such that the sum of squares of the residuals is minimized; the partial derivative with respect to each parameter should be set to zero:

$$\frac{\partial}{\partial a_i}\left[\left(y-(Xa)^T\right)(y-Xa)\right]$$

Minimizing the sum of squares

Choose the parameter vector such that the sum of squares of the residuals is minimized; the partial derivative with respect to each parameter should be set to zero:

$$\frac{\partial}{\partial a_i}\left[\left(y-(Xa)^T\right)(y-Xa)\right]$$

In Matlab, we can solve our linear system $\hat{Y} = Xa$ simply by running `a = X\y`.

Minimizing the sum of squares

Choose the parameter vector such that the sum of squares of the residuals is minimized; the partial derivative with respect to each parameter should be set to zero:

$$\frac{\partial}{\partial a_i}\left[\left(y-(Xa)^T\right)(y-Xa)\right]$$

In Matlab, we can solve our linear system $\hat{Y} = Xa$ simply by running `a = X\y`.

- If there are more data points ($N > 4$), we can write an analogue, but maybe a consistent solution does not exist (the system is over specified).
- However, matlab will find values for the vector a which minimize $\|y - aX\|^2$, so i.e. a least squares fit.

Fitting our problem: Matlab solver

```
N=length(x);
X(:,1) = ones(N,1);
X(:,2) = x;
X(:,3) = x.^2;
X(:,4) = x.^3;
A =X\y;
```

Fitting our problem: Matlab curve-fitting toolbox

- Start the toolbox: `cftool`
- Choose the dataset (x and y data)
- Choose the interpolant type (polynomial, exponential, ..., custom)
- Get the coefficients (save to workspace or write them down)

Fitting our problem: Excel

- Create a column with the independent and dependent data series (x and y)
- Create a column that computes $\hat{y}$, keeping the coefficients as separate cells
- Compute the sum of squares of the residuals (another column, sum the results)
- Use the solver to minimize this sum, modifying the coefficient cells
- Note: regression in Excel + display equation is dangerous if you choose the 'line' plot (use scatter if you can)

How good is the model?

- For a model to make sense the data points should be scattered randomly around the model predictions, the mean of the residuals d should be zero: $d_i = (y_i - \hat{y}_i)$
- Its always good to check if the residuals are not correlated with the measured values, if that is the case, it can indicate that your model is wrong.

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Regression coefficients

- Variance measured in the data (y) is:

$$\sigma_y^2 = \frac{1}{N}\sum_i (y_i - \overline{y})^2$$

- Variance of the residuals is:

$$\sigma_{\text{error}}^2 = \frac{1}{N}\sum_i (d_i)^2$$

- Variance in the model is:

$$\sigma_{\text{model}}^2 = \frac{1}{N}\sum_i \left(\hat{y}_i - \overline{\hat{y}}\right)^2$$

Regression coefficients

Given that the error is uncorrelated we can state that:

$$\sigma_y^2 = \sigma_{\text{error}}^2 + \sigma_{\text{model}}^2$$

$$R^2 = \frac{\sigma_{\text{model}}^2}{\sigma_y^2} = 1 - \frac{\sigma_{\text{error}}^2}{\sigma_y^2}$$

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}$$

- SSE: Sum of errors squared
- SST: Total sum of squares (model)
- SSR: Sum of squares (data)

Regression coefficients

- An uncorrellated error ($\overline{d} \to 0$) means that SSE, SST and SSR will have χ^2 -distributions and the ratios will have an F-distribution. If SSR/SSE is large, the model is good!
- There is a chance that the model is rubbish, but that SSR/SSE will yield a good value, Analysis of Variance (ANOVA) will be a good tool to calculate the probability of such a thing happening!

Back to the example

The statistics:

	Value
N	100
SSE	32.042
SST	896.907
SSR	928.950
R^2	0.964

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Dynamic fitting of non-linear equations: lsqnonlin

You may encounter situations where the model data is slightly more complicated to obtain (e.g. a numerical model based on ODEs where coefficients are unknown), or you want to perform fitting of multiple functions/coefficients, or just want to automate things via scripts. Matlab's Optimization toolbox gives access to a powerful function `lsqnonlin`, least-squares non-linear optimization.

General use of lsqnonlin

```
k = lsqnonlin(fun,k0,lb,ub,options)
```

- `fun` is a function handle to the fit criterium (e.g. `@myFitCrit`). The fit criterium function `myFitCrit` should return the residuals vector, e.g. $d_i = (y_i - \hat{y}_i)$. Here, y_i would again be the measurement data and $\hat{y}$ the solution computed by a model.
- `k0` is the initial guess for the fitting coefficient (or: array of initial guesses when fitting multiple coefficients)
- `lb` and `ub` are the lower and upper boundaries for `k0`. These should both be the size of the `k0`-array.
- `options` are some fitting options, for more fine-grained control on the fit procedure. Use e.g. `options = optimset('TolX',1.0E-6,'MaxFunEvals',1000);` to create an options object, or leave it empty (`options = []`).

Example use of lsqnonlin

We have experimental data stored in the file `tudataset1.mat`, containing T and U data. We want to fit a model with coefficients k_1 and k_2 with the following structure:

$$\frac{du}{dt} = -k_1 u + k_2$$

Example use of lsqnonlin

We have experimental data stored in the file `tudataset1.mat`, containing T and U data. We want to fit a model with coefficients k_1 and k_2 with the following structure:

$$\frac{du}{dt} = -k_1 u + k_2$$

- First, we need to create a function that contains the ODE:

```
function dudt = simpleode(t,u,k);
dudt = -k(1)*u + k(2);
```

Note that we supply a vector `k`, containing both coefficients for fitting

Example use of lsqnonlin

We have experimental data stored in the file `tudataset1.mat`, containing T and U data. We want to fit a model with coefficients k_1 and k_2 with the following structure:

$$\frac{du}{dt} = -k_1 u + k_2$$

- First, we need to create a function that contains the ODE:

```
function dudt = simpleode(t,u,k);
dudt = -k(1)*u + k(2);
```

  Note that we supply a vector `k`, containing both coefficients for fitting

- We create a fit criterium function:

```
function err = fitcrit(ke,T,U,U0)
[t,ue] = ode45(@simpleode,T,U0,[],ke);
err = (ue-U);
```

Example use of lsqnonlin

Now let's make a script that uses `lsqnonlin` to yield k-values fitted to our dataset:

Example use of lsqnonlin

Now let's make a script that uses `lsqnonlin` to yield k-values fitted to our dataset:

```
% initial value
U0 = 1.00;
% initial guesses for model parameters
k0 = [1.00 1.00];
% lower and upper bounds for model parameters
LB = [0.00 0.00];
UB = [Inf Inf ];
% Perform nonlinear least squares fit
options = optimset('TolX',1.0E-6,'MaxFunEvals',1000);
[ke,RESNORM,RESIDUAL,EXITFLAG,OUTPUT,LAMBDA,JACOBIAN] = lsqnonlin(@fitcrit,k0,LB,UB,
    options,T,U,U0);
```

Our fitted coefficients are stored in `ke`. Note that we get a lot more data back that allows to check the fitting results in more detail.

Example use of lsqnonlin

Postprocessing of results

The data returned by `lsqnonlin` can be used to obtain the 95% confidence intervals. Recall the command:

```
[ke,RESNORM,RESIDUAL,EXITFLAG,OUTPUT,LAMBDA,JACOBIAN] = lsqnonlin(@fitcrit,k0,LB,UB,
    options,T,U,U0);
```

Using the residuals and Jacobian we can use `nlparci` to get the confidence bounds:

```
cflim = nlparci(ke, RESIDUAL, JACOBIAN);

clc
disp('model parameters and confidence limits');
T = table;
T.ke = ke';
T.LowerCI = cflim(:,1);
T.UpperCI = cflim(:,2)
```

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

What is optimization?

Optimization is minimization or maximization of an objective function (also called a performance index or goal function) that may be subject to certain constraints.

- $\min f(x)$: Goal function
- $g(x) = 0$: Equality constraints
- $h(x) \geq 0$: Inequality constraints

Optimization Spectrum

Problem	Method	Solvers
LP	Simplex method	Linprog (Matlab)
	Barrier methods	CPLEX (GAMS, AIMMS, AMPL, OPB)
NLP	Lagrange multiplier method	Fminsearch/fmincon (Matlab)
QP	Successive linear programming	MINOS (GAMS, AMPL)
	Quadratic programming	CONOPT (GAMS)
MIP	Branch and bound	
MILP	Dynamic programming	Bintprog (Matlab)
MINLP	Generalized Benders decomposition	DICOPT (GAMS)
MIQP	Outer approximation method	BARON (GAMS)
	Disjunctive programming	

Factors of concern

- Continuity of the functions
- Convexity of the functions
- Global versus local optima
- Constrained versus unconstrained optima

Today's outline

- Introduction
- Curve fitting
- Regression
- Fitting numerical models
- Optimization
- Linear programming
- Summary

Linear programming

In linear programming the objective function and the constraints are linear functions.

Linear programming

In linear programming the objective function and the constraints are linear functions.

For example:

$\max z = f(x_1, x_2) = 40x_1 + 88x_2$
s.t. (subject to)
$2x_1 + 8x_2 \leq 60$
$5x_1 + 2x_2 \leq 60$
$x_1 \geq 0$
$x_2 \geq 0$

If the constraints are satisfied, but the objective function is not maximized/minimized we speak of a feasible solution.

If also the objective function is maximized/minimized, we speak of an optimal solution!

Plotting the constraints

Plotting the constraints

Normal form of an LP problem

$$\max z = f(x_1, x_2) = 40x_1 + 88x_2$$

s.t.

$$2x_1 + 8x_2 \leq 60$$
$$5x_1 + 2x_2 \leq 60$$
$$x_1 \geq 0$$
$$x_2 \geq 0$$

Normal form of an LP problem

$$\max z = f(x_1, x_2) = 40x_1 + 88x_2$$
s.t.
$$2x_1 + 8x_2 \leq 60$$
$$5x_1 + 2x_2 \leq 60$$
$$x_1 \geq 0$$
$$x_2 \geq 0$$

$$\max z = f(x) = 40x_1 + 88x_2$$
s.t.
$$2x_1 + 8x_2 + x_3 = 60$$
$$5x_1 + 2x_2 + x_4 = 60$$
$$x_i \geq 0 \qquad i \in 1, 2, 3, 4$$

x_3 and x_4 are called slack variables, they are non auxiliary variables introduced for the purpose of converting inequalities in to equalities

The simplex method

We can formulate our earlier example to the normal form and consider it as the following augmented matrix with $T_0 = [z \quad x_1 \quad x_2 \quad x_3 \quad x_4 \quad b]$:

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

This matrix is called the (initial) simplex table. Each simplex table has two kinds of variables, the basic variables (columns having only one nonzero entry) and the nonbasic variables

The simplex method

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

Every simplex table has a feasible solution. It can be obtained by setting the nonbasic variables to zero: $x_1 = 0$, $x_2 = 0$, $x_3 = 60/1$, $x_4 = 60/1$, $z = 0$.

The optimal solution?

- The optimal solution is now obtained stepwise by pivoting in such way that z reaches a maximum.
- The big question is, how to choose your pivot equation ...

Step 1: Selection of the pivot column

Select as the column of the pivot, the first column with a negative entry in Row 1. In our example, thats column 2 (-40)

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

Step 2: Selection of the pivot row

Divide the right sides by the corresponding column entries of the selected pivot column. In our example that is $60/2 = 30$ and $60/5 = 12$.

$$T_0 = \begin{bmatrix} 1 & -40 & -88 & 0 & 0 & 0 \\ 0 & 2 & 8 & 1 & 0 & 60 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

Take as the pivot equation the equation that gives the smallest quotient, so 60/5.

Step 3: Elimination by row operations

- Row 1 = Row 1 + 8 * Row 3
- Row 2 = Row 2 - 0.4 * Row 3

$$T_1 = \begin{bmatrix} 1 & 0 & -72 & 0 & 8 & 480 \\ 0 & 0 & 7.2 & 1 & -0.4 & 36 \\ 0 & 5 & 2 & 0 & 1 & 60 \end{bmatrix}$$

The basic variables are now x_1, x_3 and the nonbasic variables are x_2, x_4. Setting the nonbasic variables to zero will give a new feasible solution: $x_1 = 60/5$, $x_2 = 0$, $x_3 = 36/1$, $x_4 = 0$, $z = 480$.

The simplex method

- We moved from $z = 0$ to $z = 480$. The reason for the increase is because we eliminated a negative term from the eqation, so: elimination should only be applied to negative entries in Row 1, but no others.
- Although we found a feasible solution, we did not find the optimal solution yet (the entry of -72 in our simplex table) $\longrightarrow$ repeat step 1 to 3.

The simplex method

Another iteration is required:

- Step 1: Select column 3
- Step 2: $36/7.2 = 5$ and $60/2 = 30 \longrightarrow$ select 7.2 as the pivot
- Elimination by row operations:
  - Row 1 = Row 1 + 10*Row 2
  - Row 3 = Row 3 - (2/7.2)*Row 2

$$T_2 = \begin{bmatrix} 1 & 0 & 0 & 10 & 4 & 840 \\ 0 & 0 & 7.2 & 1 & -0.4 & 36 \\ 0 & 5 & 0 & -1/36 & 1/0.9 & 50 \end{bmatrix}$$

- The basic feasible solution: $x_1 = 50/5$, $x_2 = 36/7.2$, $x_3 = 0$, $x_4 = 0$, $z = 840$ (no more negative entries: so this solution is also the optimal solution)

Using Matlab for LP problems

We are going to solve the following LP problem:

$\min f(x) = -5x_1 - 4x_2 - 6x_3$

s.t.

$x_1 - x_2 + x_3 \leq 20$

$3x_1 + 2x_2 + 4x_3 \leq 42$

$3x_1 + 2x_2 \leq 30$

$x_1 \geq 0$

$x_2 \geq 0$

$x_3 \geq 0$

Using the function `linprog`:

```
f = [-5; -4; -6]
A = [1 -1 1; 3 2 4; 3 2 0];
b = [20; 42; 30];
lb = zeros(3,1);
[x,fval,exitflag,output,lambda]
= linprog(f,A,b,[],[],lb);
```

Gives:

```
x = 0.00 15.00 3.00
lambda.ineqlin = 0 1.50 0.50
lambda.lower = 1.00 0 0
```

Summary

- Curve fitting: Manual procedures for polynomial fitting in Matlab and Excel
- Curve fitting: Matlab's curve-fitting toolbox
- Curve fitting: Matlab's non-linear least-squares solver `lsqnonlin`
- Optimization: An introduction to the Simplex method
- Optimization: Use of the `linprog` solver

Numerical interpolation

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Part X

Numerical interpolation

Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines

Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines

Interpolation problem

Definition

Given a set of points x_k, $k = 0, \ldots, n$, $x_i \neq x_j$ with associated function values f_k, $k = 0, \ldots, n$, or simply: $\{x_k, f_k\}_{k=0}^{n}$. The interpolation problem is defined as: find a polynomial p_n such that this interpolates the values of f_k on the points x_k:

$$p_n(x_k) = f_k, \quad k = 0, \ldots, n$$

Interpolation problem

Definition

Given a set of points x_k, $k = 0, \ldots, n$, $x_i \neq x_j$ with associated function values f_k, $k = 0, \ldots, n$, or simply: $\{x_k, f_k\}_{k=0}^n$. The interpolation problem is defined as: find a polynomial p_n such that this interpolates the values of f_k on the points x_k:

$$p_n(x_k) = f_k, \quad k = 0, \ldots, n$$

Theorem

The interpolation problem for $\{x_k, f_k\}_{k=0}^n$ has a unique solution when $x_i \neq x_j$ for $i \neq j$. Note that we cannot allow multiple function values f_k for the same value of x_k.

What is interpolation?

Interpolation means constructing additional data points within the range of, and using, a discrete set of known data points.

It is typically performed on a uniformly spread data set, but this is not strictly necessary for all methods

Is interpolation the same as curve fitting?

Is interpolation the same as curve fitting?

NO

Is interpolation the same as curve fitting?

NO

- Curve-fitting requires additionally some way of computing the error between function (curve) and data
- Curve-fitting does not strictly enforce the function to match the data exactly
- Curve-fitting may be done on multiple datapoints at one position
- Curve-fitting is much more expensive to do, requires optimisation

Is interpolation the same as curve fitting?

NO

- Curve-fitting requires additionally some way of computing the error between function (curve) and data
- Curve-fitting does not strictly enforce the function to match the data exactly
- Curve-fitting may be done on multiple datapoints at one position
- Curve-fitting is much more expensive to do, requires optimisation

Is interpolation the same as curve fitting?

NO

- Curve-fitting requires additionally some way of computing the error between function (curve) and data
- Curve-fitting does not strictly enforce the function to match the data exactly
- Curve-fitting may be done on multiple datapoints at one position
- Curve-fitting is much more expensive to do, requires optimisation

Is interpolation the same as curve fitting?

NO

- Curve-fitting requires additionally some way of computing the error between function (curve) and data
- Curve-fitting does not strictly enforce the function to match the data exactly
- Curve-fitting may be done on multiple datapoints at one position
- Curve-fitting is much more expensive to do, requires optimisation

Why do chemical engineers need interpolation?

- Comparison of two data sets which are given at different positions
  - An experimental data set may have been recorded at a constant rate, but the numerical solution is computed at irregular intervals
- Reconstruction of field values distant of computing nodes
  - A CFD simulation on a regular grid containing structures that are not grid-conformant requires interpolation to the structures
- Calculation of a physical property at a condition between those of a lookup table
  - The viscosity of a substance may have been measured at 20°C and 30°C, but not at the desired 28.5°C

Why do chemical engineers need interpolation?

- Comparison of two data sets which are given at different positions
  - An experimental data set may have been recorded at a constant rate, but the numerical solution is computed at irregular intervals
- Reconstruction of field values distant of computing nodes
  - A CFD simulation on a regular grid containing structures that are not grid-conformant requires interpolation to the structures
- Calculation of a physical property at a condition between those of a lookup table
  - The viscosity of a substance may have been measured at 20°C and 30°C, but not at the desired 28.5°C

Why do chemical engineers need interpolation?

- Comparison of two data sets which are given at different positions
  - An experimental data set may have been recorded at a constant rate, but the numerical solution is computed at irregular intervals
- Reconstruction of field values distant of computing nodes
  - A CFD simulation on a regular grid containing structures that are not grid-conformant requires interpolation to the structures
- Calculation of a physical property at a condition between those of a lookup table
  - The viscosity of a substance may have been measured at 20°C and 30°C, but not at the desired 28.5°C

General

Several important numerical interpolation methods are discussed today:

- Piecewise constant interpolation
- Linear interpolation
  - Bilinear interpolation
- Polynomial interpolation (Newton's method)
- Spline interpolation

Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines

Today's data set

Download the datafile `interpolation-dataset.mat`, which contains multiple data sets.

Today's data set

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

Download the datafile `interpolation-dataset.mat`, which contains multiple data sets.

We start with `x1` and `y1`:

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$
3	1.00
4	$\frac{5}{3} = 1.67$
5	$\frac{23}{3} = 7.67$

Piecewise constant interpolation

- Nearest-neighbor interpolation in the continuous range $x \in [0,5]$
- How to treat the point halfway (e.g. at $x = 2.5$)?

$$
\begin{aligned}
x \in &[0, 0.5] && \rightarrow f(x) = f(0) \\
x \in &]0.5, 1.5] && \rightarrow f(x) = f(1) \\
x \in &]1.5, 2.5] && \rightarrow f(x) = f(2) \\
x \in &]2.5, 3.5] && \rightarrow f(x) = f(3) \\
x \in &]3.5, 4.5] && \rightarrow f(x) = f(4)
\end{aligned}
$$

- Not often used for simple problems, but e.g. for 2D (Voronoi)

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

Piecewise constant interpolation

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

- Nearest-neighbor interpolation in the continuous range $x \in [0, 5]$
- How to treat the point halfway (e.g. at $x = 2.5$)?

$x \in [0, 0.5] \rightarrow f(x) = f(0)$

$x \in]0.5, 1.5] \rightarrow f(x) = f(1)$

$x \in]1.5, 2.5] \rightarrow f(x) = f(2)$

$x \in]2.5, 3.5] \rightarrow f(x) = f(3)$

$x \in]3.5, 4.5] \rightarrow f(x) = f(4)$

- Not often used for simple problems, but e.g. for 2D (Voronoi)

Piecewise constant interpolation

- Nearest-neighbor interpolation in the continuous range $x \in [0,5]$
- How to treat the point halfway (e.g. at $x = 2.5$)?

$x \in [0, 0.5] \rightarrow f(x) = f(0)$

$x \in]0.5, 1.5] \rightarrow f(x) = f(1)$

$x \in]1.5, 2.5] \rightarrow f(x) = f(2)$

$x \in]2.5, 3.5] \rightarrow f(x) = f(3)$

$x \in]3.5, 4.5] \rightarrow f(x) = f(4)$

- Not often used for simple problems, but e.g. for 2D (Voronoi)

Data set $f_n(x_n)$ represented by o at discrete intervals $x_n \in \{0, 5\}$

Piecewise constant interpolation

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

- Nearest-neighbor interpolation in the continuous range $x \in [0, 5]$
- How to treat the point halfway (e.g. at $x = 2.5$)?

$$
\begin{aligned}
x &\in [0, 0.5] && \rightarrow f(x) = f(0) \\
x &\in]0.5, 1.5] && \rightarrow f(x) = f(1) \\
x &\in]1.5, 2.5] && \rightarrow f(x) = f(2) \\
x &\in]2.5, 3.5] && \rightarrow f(x) = f(3) \\
x &\in]3.5, 4.5] && \rightarrow f(x) = f(4)
\end{aligned}
$$

- Not often used for simple problems, but e.g. for 2D (Voronoi)

Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines

Linear interpolation

- Linear interpolation to (x, y) between 2 data points (x_2, y_2) and (x_3, y_3):

$$\frac{y - y_2}{x - x_2} = \frac{y_3 - y_2}{x_3 - x_2}$$

- Reordered, and more formally:

$$y = y_n + (y_{n+1} - y_n)\frac{x - x_n}{x_{n+1} - x_n}$$

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

Linear interpolation

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

- Linear interpolation to (x, y) between 2 data points (x_2, y_2) and (x_3, y_3):

$$\frac{y - y_2}{x - x_2} = \frac{y_3 - y_2}{x_3 - x_2}$$

- Reordered, and more formally:

$$y = y_n + (y_{n+1} - y_n)\frac{x - x_n}{x_{n+1} - x_n}$$

Linear interpolation

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

- Linear interpolation to (x, y) between 2 data points (x_2, y_2) and (x_3, y_3):

$$\frac{y - y_2}{x - x_2} = \frac{y_3 - y_2}{x_3 - x_2}$$

- Reordered, and more formally:

$$y = y_n + (y_{n+1} - y_n)\frac{x - x_n}{x_{n+1} - x_n}$$

Linear interpolation

Data set $f_n(x_n)$ represented by ○ at discrete intervals $x_n \in \{0, 5\}$

- Linear interpolation to (x, y) between 2 data points (x_2, y_2) and (x_3, y_3):

$$\frac{y - y_2}{x - x_2} = \frac{y_3 - y_2}{x_3 - x_2}$$

- Reordered, and more formally:

$$y = y_n + (y_{n+1} - y_n)\frac{x - x_n}{x_{n+1} - x_n}$$

Linear interpolation

- While linear interpolation is fast, and relatively easy to program, it is not very accurate
- At the nodes, the derivatives are discontinuous i.e. not differentiable
- Error is proportional to the square of the distance between nodes

Linear interpolation

- While linear interpolation is fast, and relatively easy to program, it is not very accurate
- At the nodes, the derivatives are discontinuous i.e. not differentiable
- Error is proportional to the square of the distance between nodes

Linear interpolation

- While linear interpolation is fast, and relatively easy to program, it is not very accurate
- At the nodes, the derivatives are discontinuous i.e. not differentiable
- Error is proportional to the square of the distance between nodes

Example: Linear interpolation in Matlab

Consider the data set in `sim_exp_dataset.mat`, containing a normalized concentration and time vector for an experiment and a simulation. The simulation was performed with adaptive node distance to save computation time, thus the concentration is not known at the same times. We are not able to compare yet.

Example: Linear interpolation in Matlab

Consider the data set in `sim_exp_dataset.mat`, containing a normalized concentration and time vector for an experiment and a simulation. The simulation was performed with adaptive node distance to save computation time, thus the concentration is not known at the same times. We are not able to compare yet.

```
% Linear interpolation
c_sim_new = interp1(t_sim,c_sim,t_exp,'linear');
diff = abs(c_exp-c_sim_new);
% Plot the solution
subplot(2,1,1);
plot(t_exp,c_exp,'b-x',t_exp,c_sim_new,'r-o');
subplot(2,1,2);
stem(t_exp,diff);
% Compute the L2-norm
norm(diff)
```

Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain $p(x, y)$ using 4 field values f_{00}, f_{10}, f_{01} and f_{11}.

$$g_1 = f_{01}\frac{x_1 - x}{x_1 - x_0} + f_{11}\frac{x - x_0}{x_1 - x_0}$$
$$= f_{01}\frac{x_1 - x}{\Delta x} + f_{11}\frac{x - x_0}{\Delta x}$$

$$g_2 = f_{00}\frac{x_1 - x}{\Delta x} + f_{10}\frac{x - x_0}{\Delta x}$$

$$p = g_2\frac{y_1 - y}{\Delta y} + g_1\frac{y - y_0}{\Delta y}$$

Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain $p(x, y)$ using 4 field values f_{00}, f_{10}, f_{01} and f_{11}.

$$g_1 = f_{01}\frac{x_1 - x}{x_1 - x_0} + f_{11}\frac{x - x_0}{x_1 - x_0}$$
$$= f_{01}\frac{x_1 - x}{\Delta x} + f_{11}\frac{x - x_0}{\Delta x}$$

$$g_2 = f_{00}\frac{x_1 - x}{\Delta x} + f_{10}\frac{x - x_0}{\Delta x}$$

$$p = g_2\frac{y_1 - y}{\Delta y} + g_1\frac{y - y_0}{\Delta y}$$

Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain $p(x, y)$ using 4 field values f_{00}, f_{10}, f_{01} and f_{11}.

$$g_1 = f_{01}\frac{x_1 - x}{x_1 - x_0} + f_{11}\frac{x - x_0}{x_1 - x_0}$$
$$= f_{01}\frac{x_1 - x}{\Delta x} + f_{11}\frac{x - x_0}{\Delta x}$$

$$g_2 = f_{00}\frac{x_1 - x}{\Delta x} + f_{10}\frac{x - x_0}{\Delta x}$$

$$p = g_2\frac{y_1 - y}{\Delta y} + g_1\frac{y - y_0}{\Delta y}$$

Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain $p(x, y)$ using 4 field values f_{00}, f_{10}, f_{01} and f_{11}.

$$g_1 = f_{01}\frac{x_1 - x}{x_1 - x_0} + f_{11}\frac{x - x_0}{x_1 - x_0}$$
$$= f_{01}\frac{x_1 - x}{\Delta x} + f_{11}\frac{x - x_0}{\Delta x}$$

$$g_2 = f_{00}\frac{x_1 - x}{\Delta x} + f_{10}\frac{x - x_0}{\Delta x}$$

$$p = g_2\frac{y_1 - y}{\Delta y} + g_1\frac{y - y_0}{\Delta y}$$

Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain $p(x, y)$ using 4 field values f_{00}, f_{10}, f_{01} and f_{11}.

$$g_1 = f_{01}\frac{x_1 - x}{x_1 - x_0} + f_{11}\frac{x - x_0}{x_1 - x_0}$$
$$= f_{01}\frac{x_1 - x}{\Delta x} + f_{11}\frac{x - x_0}{\Delta x}$$

$$g_2 = f_{00}\frac{x_1 - x}{\Delta x} + f_{10}\frac{x - x_0}{\Delta x}$$

$$p = g_2\frac{y_1 - y}{\Delta y} + g_1\frac{y - y_0}{\Delta y}$$

Bi-linear interpolation

When a 2D field of some quantity is known, we can interpolate the solution to an arbitrary position in the 2D domain $p(x, y)$ using 4 field values f_{00}, f_{10}, f_{01} and f_{11}.

$$g_1 = f_{01}\frac{x_1 - x}{x_1 - x_0} + f_{11}\frac{x - x_0}{x_1 - x_0}$$
$$= f_{01}\frac{x_1 - x}{\Delta x} + f_{11}\frac{x - x_0}{\Delta x}$$

$$g_2 = f_{00}\frac{x_1 - x}{\Delta x} + f_{10}\frac{x - x_0}{\Delta x}$$

$$p = g_2\frac{y_1 - y}{\Delta y} + g_1\frac{y - y_0}{\Delta y}$$

- The order of interpolation (x or y direction first) does not matter; the results are equal

Higher-dimensional field interpolation in Matlab

2D or higher-dimensional fields of data can be interpolated in Matlab using the `interp2`, `interp3` or even `interpn` functions, the method can be adjusted:

- Similar to 1D linear interpolation, the derivatives are discontinuous on the grid nodes
- Also consider tri-linear interpolation (for 3D fields), or bicubic interpolation (2D, but third order)

A practical example

Field interpolation is used in e.g. CFD simulations, e.g. a fluidized bed simulation using a *discrete particle model*, where particles are found in between the grid nodes used for velocity

Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines

Polynomial interpolation

The examples that we have seen, are simplified forms of *Newton polynomials*. We can interpolate our data with a polynomial of degree n:

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

Polynomial interpolation via Vandermonde matrix

Consider the data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, the Vandermonde matrix V, coefficient vector a and function value vector y:

$$V_{m,n} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix} \quad a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

The coefficients of a polynomial through the data points can be obtained by solving the linear system $Va = y$.

Polynomial interpolation via Vandermonde matrix

Consider the data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, the Vandermonde matrix V, coefficient vector a and function value vector y:

$$V_{m,n} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix} \quad a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

The coefficients of a polynomial through the data points can be obtained by solving the linear system $Va = y$.

```
>> x = [0 1 2];
>> y = [1.0000; 3.6667; 2.6667];
>> V = vander(x);
>> a = V\y
a =
   -1.8333
    4.5000
    1.0000
```

Polynomial interpolation via Vandermonde matrix

Consider the data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, the Vandermonde matrix V, coefficient vector a and function value vector y:

$$V_{m,n} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix} \quad a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

The coefficients of a polynomial through the data points can be obtained by solving the linear system $Va = y$.

```
>> x = [0 1 2];
>> y = [1.0000; 3.6667; 2.6667];
>> V = vander(x);
>> a = V\y
a =
   -1.8333
    4.5000
    1.0000
```

So we found the equation:

$$p_2(x) = -1.8333x^2 + 4.5x - 1$$

Polynomial interpolation via Vandermonde matrix

Consider the data points $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$, the Vandermonde matrix V, coefficient vector a and function value vector y:

$$V_{m,n} = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{n-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \cdots & x_m^{n-1} \end{pmatrix} \quad a = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{pmatrix}$$

The coefficients of a polynomial through the data points can be obtained by solving the linear system $Va = y$.

```
>> x = [0 1 2];
>> y = [1.0000; 3.6667; 2.6667];
>> V = vander(x);
>> a = V\y
a =
   -1.8333
    4.5000
    1.0000
```

So we found the equation:

$$p_2(x) = -1.8333x^2 + 4.5x - 1$$

These Vandermonde-systems are often *ill-conditioned*, so we need another, more stable, method!

Construction of Newton polynomials

Formally, the polynomials $p_n(x)$ are described using prefactors $f[x_0,\ldots,x_k]$ and polynomial terms $w_m(x)$:

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k] w_k(x)$$

Construction of Newton polynomials

Formally, the polynomials $p_n(x)$ are described using prefactors $f[x_0,\dots,x_k]$ and polynomial terms $w_m(x)$:

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\dots,x_k] w_k(x)$$

The polynomial terms are computed via:

$$w_0(x) = 1,\ w_1(x) = (x - x_0),\ w_2(x) = (x - x_0)\cdot(x - x_1),$$
$$w_m(x) = (x - x_0)\cdot(x - x_1)\cdots(x - x_{m-1}) = w_{m-1}\cdot(x - x_{m-1})$$
$$w_m(x) = \prod_{j=0}^{m-1}(x - x_j), \qquad m = 0,\dots,n$$

Construction of Newton polynomials

Formally, the polynomials $p_n(x)$ are described using prefactors $f[x_0,\ldots,x_k]$ and polynomial terms $w_m(x)$:

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k] w_k(x)$$

The polynomial terms are computed via:

$$w_0(x) = 1,\ w_1(x) = (x - x_0),\ w_2(x) = (x - x_0)\cdot(x - x_1),$$
$$w_m(x) = (x - x_0)\cdot(x - x_1)\cdots(x - x_{m-1}) = w_{m-1}\cdot(x - x_{m-1})$$
$$w_m(x) = \prod_{j=0}^{m-1}(x - x_j), \qquad m = 0,\ldots,n$$

The prefactors are *forward divided differences*, which can be computed as:

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k
x_0	$f[x_0] = f_0$

x_k	f_k
0	1

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k
x_0	$f[x_0] = f_0$
x_1	$f[x_1] = f_1 \qquad f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_0}$

x_k	f_k
0	1
1	3.67 $\qquad \frac{\frac{11}{3} - 1}{1 - 0} = \frac{8}{3}$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1}(x - x_j)$$

x_k	f_k		
x_0	$f[x_0] = f_0$		
x_1	$f[x_1] = f_1$	$f[x_0,x_1] = \frac{f_1 - f_0}{x_1 - x_0}$	
x_2	$f[x_2] = f_2$	$f[x_1,x_2] = \frac{f_2 - f_1}{x_2 - x_1}$	$f[x_0,x_1,x_2] = \frac{f[x_1,x_2] - f[x_0,x_1]}{x_2 - x_0}$

x_k	f_k		
0	1		
1	3.67	$\frac{\frac{11}{3} - 1}{1 - 0} = \frac{8}{3}$	
2	2.67	$\frac{\frac{8}{3} - \frac{11}{3}}{2 - 1} = \frac{-1}{1} = -1$	$\frac{(-1) - \frac{8}{3}}{2 - 0} = -\frac{11}{6}$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k		
0	1		
1	3.67	$\frac{\frac{11}{3}-1}{1-0} = \frac{8}{3}$	
2	2.67	$\frac{\frac{8}{3}-\frac{11}{3}}{2-1} = \frac{-1}{1} = -1$	$\frac{(-1)-\frac{8}{3}}{2-0} = -\frac{11}{6}$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k		
0	1		
1	3.67	$\frac{\frac{11}{3}-1}{1-0} = \frac{8}{3}$	
2	2.67	$\frac{\frac{8}{3}-\frac{11}{3}}{2-1} = \frac{-1}{1} = -1$	$\frac{(-1)-\frac{8}{3}}{2-0} = -\frac{11}{6}$

$$p_2(x) = 1 \cdot w_m(0) + \frac{8}{3} \cdot w_m(1) + \left(-\frac{11}{6}\right) \cdot w_m(2)$$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k		
0	1		
1	3.67	$\frac{\frac{11}{3}-1}{1-0} = \frac{8}{3}$	
2	2.67	$\frac{\frac{8}{3}-\frac{11}{3}}{2-1} = \frac{-1}{1} = -1$	$\frac{(-1)-\frac{8}{3}}{2-0} = -\frac{11}{6}$

$$p_2(x) = 1 \cdot w_m(0) + \frac{8}{3} \cdot w_m(1) + \left(-\frac{11}{6}\right) \cdot w_m(2)$$

$$= 1 \cdot 1 + \frac{8}{3} \cdot (x - 0) + \left(-\frac{11}{6}\right) \cdot (x - 0)(x - 1)$$

Construction of Newton polynomials: example

Sample data

x_k	f_k
0	1.00
1	$\frac{11}{3} = 3.67$
2	$\frac{8}{3} = 2.67$

$$p_n(x) = \sum_{k=0}^{n} f[x_0,\ldots,x_k]\, w_k(x)$$

$$f[x_{x-k},\ldots,x_r] \equiv \frac{f[x_{r-k+1},\ldots,x_r] - f[x_{r-k},\ldots,x_{r-1}]}{x_r - x_{r-k}}$$

$$w_m(x) = \prod_{j=0}^{m-1} (x - x_j)$$

x_k	f_k		
0	1		
1	3.67	$\frac{\frac{11}{3}-1}{1-0} = \frac{8}{3}$	
2	2.67	$\frac{\frac{8}{3}-\frac{11}{3}}{2-1} = \frac{-1}{1} = -1$	$\frac{(-1)-\frac{8}{3}}{2-0} = -\frac{11}{6}$

$$p_2(x) = 1 \cdot w_m(0) + \frac{8}{3} \cdot w_m(1) + \left(-\frac{11}{6}\right) \cdot w_m(2)$$

$$= 1 \cdot 1 + \frac{8}{3} \cdot (x - 0) + \left(-\frac{11}{6}\right) \cdot (x-0)(x-1) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

Construction of Newton polynomials: example

For each three points, a new polynomial interpolant can be derived:

$$p_2(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$$

$$p_2(x) = 4 - \frac{x^2}{3}$$

$$p_2(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$$

$$p_2(x) = \frac{8}{3}x^2 - 18x + 31$$

$$f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$$

Polynomial fitting in Matlab: example

Develop the $p_2(x)$, $p_3(x)$ and $p_4(x)$ from the following data set (example data `x2` and `y2`):

x_k	y_k
−1.0	2.8677
−0.5	7.7530
0.0	22.0000
0.5	65.7863
1.0	208.6744

Polynomial fitting in Matlab: example

Develop the $p_2(x)$, $p_3(x)$ and $p_4(x)$ from the following data set (example data `x2` and `y2`):

x_k	y_k
−1.0	2.8677
−0.5	7.7530
0.0	22.0000
0.5	65.7863
1.0	208.6744

We use the built-in `polyfit(x,y,n)` and `polyval(p,x)` functions:

Polynomial fitting in Matlab: example

Develop the $p_2(x)$, $p_3(x)$ and $p_4(x)$ from the following data set (example data `x2` and `y2`):

x_k	y_k
−1.0	2.8677
−0.5	7.7530
0.0	22.0000
0.5	65.7863
1.0	208.6744

We use the built-in `polyfit(x,y,n)` and `polyval(p,x)` functions:

```
x_cont = linspace(-1,1,1001);
p2 = polyfit(x2,y2,2);
p3 = polyfit(x2,y2,3);
p4 = polyfit(x2,y2,4);
```

Polynomial fitting in Matlab: example

Develop the $p_2(x)$, $p_3(x)$ and $p_4(x)$ from the following data set (example data `x2` and `y2`):

x_k	y_k
−1.0	2.8677
−0.5	7.7530
0.0	22.0000
0.5	65.7863
1.0	208.6744

We use the built-in `polyfit(x,y,n)` and `polyval(p,x)` functions:

```
x_cont = linspace(-1,1,1001);
p2 = polyfit(x2,y2,2);
p3 = polyfit(x2,y2,3);
p4 = polyfit(x2,y2,4);
```

```
y_cont2 = polyval(p2,x_cont);
y_cont3 = polyval(p3,x_cont);
y_cont4 = polyval(p4,x_cont);
plot(x2,y2,'o',x_cont,y_cont2,x_cont,y_cont3,x_cont,y_cont4)
```

Polynomial fitting in Matlab: example

Develop the $p_2(x)$, $p_3(x)$ and $p_4(x)$ from the following data set (example data `x2` and `y2`):

x_k	y_k
−1.0	2.8677
−0.5	7.7530
0.0	22.0000
0.5	65.7863
1.0	208.6744

We use the built-in `polyfit(x,y,n)` and `polyval(p,x)` functions:

```
x_cont = linspace(-1,1,1001);
p2 = polyfit(x2,y2,2);
p3 = polyfit(x2,y2,3);
p4 = polyfit(x2,y2,4);
```

```
y_cont2 = polyval(p2,x_cont);
y_cont3 = polyval(p3,x_cont);
y_cont4 = polyval(p4,x_cont);
plot(x2,y2,'o',x_cont,y_cont2,x_cont,y_cont3,x_cont,y_cont4)
```

Polynomial fitting in Matlab: example

Develop the $p_2(x)$, $p_3(x)$ and $p_4(x)$ from the following data set (example data `x2` and `y2`):

x_k	y_k
−1.0	2.8677
−0.5	7.7530
0.0	22.0000
0.5	65.7863
1.0	208.6744

We use the built-in `polyfit(x,y,n)` and `polyval(p,x)` functions:

```
x_cont = linspace(-1,1,1001);
p2 = polyfit(x2,y2,2);
p3 = polyfit(x2,y2,3);
p4 = polyfit(x2,y2,4);
```

```
y_cont2 = polyval(p2,x_cont);
y_cont3 = polyval(p3,x_cont);
y_cont4 = polyval(p4,x_cont);
plot(x2,y2,'o',x_cont,y_cont2,x_cont,y_cont3,x_cont,y_cont4)
```

Polynomial fitting in Matlab: example

Develop the $p_2(x)$, $p_3(x)$ and $p_4(x)$ from the following data set (example data `x2` and `y2`):

x_k	y_k
−1.0	2.8677
−0.5	7.7530
0.0	22.0000
0.5	65.7863
1.0	208.6744

We use the built-in `polyfit(x,y,n)` and `polyval(p,x)` functions:

```
x_cont = linspace(-1,1,1001);
p2 = polyfit(x2,y2,2);
p3 = polyfit(x2,y2,3);
p4 = polyfit(x2,y2,4);
```

```
y_cont2 = polyval(p2,x_cont);
y_cont3 = polyval(p3,x_cont);
y_cont4 = polyval(p4,x_cont);
plot(x2,y2,'o',x_cont,y_cont2,x_cont,y_cont3,x_cont,y_cont4)
```

Exercise

Develop the $p_4(x)$ and $p_{10}(x)$ interpolants from the following data sets:

$$f(x) = \frac{1}{x^2 + \frac{1}{25}} \qquad x \in [-1, 1]$$

```
x3a = linspace(-1 , 1 , 5);
x3b = linspace(-1 , 1 , 11);
y3a = 1 ./ (x3a.^2 + (1/25));
y3b = 1 ./ (x3b.^2 + (1/25));
```

Exercise

Develop the $p_4(x)$ and $p_{10}(x)$ interpolants from the following data sets:

$$f(x) = \frac{1}{x^2 + \frac{1}{25}} \qquad x \in [-1, 1]$$

```
x3a = linspace(-1 , 1 , 5);
x3b = linspace(-1 , 1 , 11);
y3a = 1 ./ (x3a.^2 + (1/25));
y3b = 1 ./ (x3b.^2 + (1/25));
```

```
x_cont = linspace(-1,1,1001);
p4 = polyfit(x3a,y3a,4);
p10 = polyfit(x3b,y3b,10);
y_cont4 = polyval(p4,x_cont);
y_cont10 = polyval(p10,x_cont);
ezplot('1./(x.^2+(1/25))',[-1 1]); hold on;
plot(x3a,y3a,'o',x3b,y3b,'x',x_cont,y_cont4,x_cont,y_cont10);
```

Exercise

Develop the $p_4(x)$ and $p_{10}(x)$ interpolants from the following data sets:

$$f(x) = \frac{1}{x^2 + \frac{1}{25}} \qquad x \in [-1, 1]$$

```
x3a = linspace(-1 , 1 , 5);
x3b = linspace(-1 , 1 , 11);
y3a = 1 ./ (x3a.^2 + (1/25));
y3b = 1 ./ (x3b.^2 + (1/25));
```

```
x_cont = linspace(-1,1,1001);
p4 = polyfit(x3a,y3a,4);
p10 = polyfit(x3b,y3b,10);
y_cont4 = polyval(p4,x_cont);
y_cont10 = polyval(p10,x_cont);
ezplot('1./(x.^2+(1/25))',[-1 1]); hold on;
plot(x3a,y3a,'o',x3b,y3b,'x',x_cont,y_cont4,x_cont,y_cont10);
```

Exercise

Develop the $p_4(x)$ and $p_{10}(x)$ interpolants from the following data sets:

$$f(x) = \frac{1}{x^2 + \frac{1}{25}} \qquad x \in [-1, 1]$$

```
x3a = linspace(-1 , 1 , 5);
x3b = linspace(-1 , 1 , 11);
y3a = 1 ./ (x3a.^2 + (1/25));
y3b = 1 ./ (x3b.^2 + (1/25));
```

```
x_cont = linspace(-1,1,1001);
p4 = polyfit(x3a,y3a,4);
p10 = polyfit(x3b,y3b,10);
y_cont4 = polyval(p4,x_cont);
y_cont10 = polyval(p10,x_cont);
ezplot('1./(x.^2+(1/25))',[-1 1]); hold on;
plot(x3a,y3a,'o',x3b,y3b,'x',x_cont,y_cont4,x_cont,y_cont10);
```

Exercise

Develop the $p_4(x)$ and $p_{10}(x)$ interpolants from the following data sets:

$$f(x) = \frac{1}{x^2 + \frac{1}{25}} \qquad x \in [-1, 1]$$

```
x3a = linspace(-1 , 1 , 5);
x3b = linspace(-1 , 1 , 11);
y3a = 1 ./ (x3a.^2 + (1/25));
y3b = 1 ./ (x3b.^2 + (1/25));
```

```
x_cont = linspace(-1,1,1001);
p4 = polyfit(x3a,y3a,4);
p10 = polyfit(x3b,y3b,10);
y_cont4 = polyval(p4,x_cont);
y_cont10 = polyval(p10,x_cont);
ezplot('1./(x.^2+(1/25))',[-1 1]); hold on;
plot(x3a,y3a,'o',x3b,y3b,'x',x_cont,y_cont4,x_cont,y_cont10);
```

Final thoughts on polynomial interpolation

- An polynomial interpolant of order n requires $n+1$ data points
  - More data points: interpolant does *not always* cross the points
  - Fewer data points: interpolant is not unique
- Higher-degree polynomials at equidistant points may cause strong oscillatory behaviour (Runge's phenomenon)
  - Mitigation of the problem on Chebyshev (i.e. non uniform grid)...
  - ... or by performing piecewise interpolation (next topic)
- Matlab functions `polyfit(x,y,n)` and `polyval(p,x_new)` were demonstrated.

Final thoughts on polynomial interpolation

- An polynomial interpolant of order n requires $n+1$ data points
  - More data points: interpolant does *not always* cross the points
  - Fewer data points: interpolant is not unique
- Higher-degree polynomials at equidistant points may cause strong oscillatory behaviour (Runge's phenomenon)
  - Mitigation of the problem on Chebyshev (i.e. non uniform grid)...
  - ... or by performing piecewise interpolation (next topic)
- Matlab functions `polyfit(x,y,n)` and `polyval(p,x_new)` were demonstrated.

Final thoughts on polynomial interpolation

- An polynomial interpolant of order n requires $n+1$ data points
  - More data points: interpolant does *not always* cross the points
  - Fewer data points: interpolant is not unique
- Higher-degree polynomials at equidistant points may cause strong oscillatory behaviour (Runge's phenomenon)
  - Mitigation of the problem on Chebyshev (i.e. non uniform grid)...
  - ... or by performing piecewise interpolation (next topic)
- Matlab functions `polyfit(x,y,n)` and `polyval(p,x_new)` were demonstrated.

Today's outline

- Introduction
- Piecewise constant
- Linear
- Polynomial
- Splines

Spline interpolation

A spline is a numerical function that represents a smooth, higher order, piecewise polynomial interpolants of a data set.

Spline interpolation

A spline is a numerical function that represents a smooth, higher order, piecewise polynomial interpolants of a data set.

- Smooth: the interpolant is continuous in the first and second derivatives
- Higher order: The most common type of splines uses third-order polynomials (cubic splines)
- Piecewise polynomial: The interpolant is constructed between each two consecutive tabulated points

Spline interpolation

A spline is a numerical function that represents a smooth, higher order, piecewise polynomial interpolants of a data set.

- Smooth: the interpolant is continuous in the first and second derivatives
- Higher order: The most common type of splines uses third-order polynomials (cubic splines)
- Piecewise polynomial: The interpolant is constructed between each two consecutive tabulated points

Spline interpolation

A spline is a numerical function that represents a smooth, higher order, piecewise polynomial interpolants of a data set.

- Smooth: the interpolant is continuous in the first and second derivatives
- Higher order: The most common type of splines uses third-order polynomials (cubic splines)
- Piecewise polynomial: The interpolant is constructed between each two consecutive tabulated points

Splines: comparison to other interpolation techniques

Interpolation of $f(x) = \dfrac{\sin x}{1 + x^2}$

Splines: comparison to other interpolation techniques

Interpolation of $f(x) = \dfrac{\sin x}{1+x^2}$

Splines: comparison to other interpolation techniques

Interpolation of $f(x) = \dfrac{\sin x}{1 + x^2}$

Splines: comparison to other interpolation techniques

Interpolation of $f(x) = \dfrac{\sin x}{1 + x^2}$

Splines: comparison to other interpolation techniques

Interpolation of $f(x) = \dfrac{\sin x}{1 + x^2}$

Spline interpolation in Matlab

We can generate a random data set, and interpolate using `interp1`:

Spline interpolation in Matlab

We can generate a random data set, and interpolate using `interp1`:

```
% Generate random data set
x=0:25;
y = rand(size(x));
% Interpolant on a fine mesh
xc = linspace(0,25,1001);
yc = interp1(x,y,xc,'spline');
plot(x,y,'o',xc,yc,'-r')
```

Summary

- Interpolation is used to obtain data between existing data points
  - (Bi-)Linear, polynomial and spline interpolation methods
  - Construction of Newton polynomials
  - Oscillations of high-order polynomials

Numerical integration

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion

What is numerical integration?

To determine the integral $I(x)$ of an integrand $f(x)$, which can be used to compute the area underneath the integrand between $x = a$ and $x = b$.

$$I(x) = \int_a^b f(x)dx$$

What is numerical integration?

To determine the integral $I(x)$ of an integrand $f(x)$, which can be used to compute the area underneath the integrand between $x = a$ and $x = b$.

$$I(x) = \int_a^b f(x)dx$$

Today we will outline different numerical integration methods.

- Riemann integrals
- Trapezoidal rule
- Simpson's rule

Why do chemical engineers need integration?

- Obtaining the cumulative particle size distribution from a particle size distribution
- The concentration outflow over time may be integrated to yield the residence time distribution
- Integration of a varying product outflow yields the total product outflow
- Quantitative analysis of mixture components via e.g. GC/MS
- Not all function have an explicit antiderivative, e.g. $\int e^{x^2} dx$ or $\int \frac{1}{\ln x} dx$

Why do chemical engineers need integration?

- Obtaining the cumulative particle size distribution from a particle size distribution
- The concentration outflow over time may be integrated to yield the residence time distribution
- Integration of a varying product outflow yields the total product outflow
- Quantitative analysis of mixture components via e.g. GC/MS
- Not all function have an explicit antiderivative, e.g. $\int e^{x^2} dx$ or $\int \frac{1}{\ln x} dx$

Why do chemical engineers need integration?

- Obtaining the cumulative particle size distribution from a particle size distribution
- The concentration outflow over time may be integrated to yield the residence time distribution
- Integration of a varying product outflow yields the total product outflow
- Quantitative analysis of mixture components via e.g. GC/MS
- Not all function have an explicit antiderivative, e.g. $\int e^{x^2} dx$ or $\int \frac{1}{\ln x} dx$

Why do chemical engineers need integration?

- Obtaining the cumulative particle size distribution from a particle size distribution
- The concentration outflow over time may be integrated to yield the residence time distribution
- Integration of a varying product outflow yields the total product outflow
- Quantitative analysis of mixture components via e.g. GC/MS
- Not all function have an explicit antiderivative, e.g. $\int e^{x^2} dx$ or $\int \frac{1}{\ln x} dx$

Why do chemical engineers need integration?

- Obtaining the cumulative particle size distribution from a particle size distribution
- The concentration outflow over time may be integrated to yield the residence time distribution
- Integration of a varying product outflow yields the total product outflow
- Quantitative analysis of mixture components via e.g. GC/MS
- Not all function have an explicit antiderivative, e.g. $\int e^{x^2} dx$ or $\int \frac{1}{\ln x} dx$

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion

Riemann integrals

Basic idea: Subdivide the interval $[a, b]$ into n subintervals of equal length $\Delta x = \frac{b-a}{n}$ and use the sum of area to approximate the integral.

Riemann integrals

Basic idea: Subdivide the interval $[a, b]$ into n subintervals of equal length $\Delta x = \frac{b-a}{n}$ and use the sum of area to approximate the integral.

Left endpoint rule

$$L_n = \sum_{i=1}^{n} f(x_{i-1})\Delta x_i$$

Riemann integrals

Basic idea: Subdivide the interval $[a, b]$ into n subintervals of equal length $\Delta x = \frac{b-a}{n}$ and use the sum of area to approximate the integral.

Left endpoint rule

$$L_n = \sum_{i=1}^{n} f(x_{i-1})\Delta x_i$$

Right endpoint rule

$$R_n = \sum_{i=1}^{n} f(x_i)\Delta x_i$$

Riemann integrals

Basic idea: Subdivide the interval $[a, b]$ into n subintervals of equal length $\Delta x = \frac{b-a}{n}$ and use the sum of area to approximate the integral.

Left endpoint rule

$$L_n = \sum_{i=1}^{n} f(x_{i-1})\Delta x_i$$

Right endpoint rule

$$R_n = \sum_{i=1}^{n} f(x_i)\Delta x_i$$

Midpoint rule

$$M_n = \sum_{i=1}^{n} f(\bar{x}_i)\Delta x_i$$

with $\bar{x}_i = \frac{x_{i-1}+x_i}{2}$

Errors in Riemann integrals

We define the exact integral as $I = \int_a^b f(x)dx$, and L_n, R_n and M_n represent the left, right and midpoint rule approximations of I based on n intervals.

Errors in Riemann integrals

We define the exact integral as $I = \int_a^b f(x)dx$, and L_n, R_n and M_n represent the left, right and midpoint rule approximations of I based on n intervals.

Writing $f_{\max}^{(k)}$ for the maximum value of the k-th derivative, the upper-bounds of the errors by Riemann integrals are:

- $|I - L_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - R_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - M_n| \leq \dfrac{f_{\max}^{(2)}(b-a)^3}{24n^2}$

Errors in Riemann integrals

We define the exact integral as $I = \int_a^b f(x)dx$, and L_n, R_n and M_n represent the left, right and midpoint rule approximations of I based on n intervals.

Writing $f_{\max}^{(k)}$ for the maximum value of the k-th derivative, the upper-bounds of the errors by Riemann integrals are:

- $|I - L_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - R_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - M_n| \leq \dfrac{f_{\max}^{(2)}(b-a)^3}{24n^2}$

Errors in Riemann integrals

We define the exact integral as $I = \int_a^b f(x)dx$, and L_n, R_n and M_n represent the left, right and midpoint rule approximations of I based on n intervals.

Writing $f_{\max}^{(k)}$ for the maximum value of the k-th derivative, the upper-bounds of the errors by Riemann integrals are:

- $|I - L_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - R_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - M_n| \leq \dfrac{f_{\max}^{(2)}(b-a)^3}{24n^2}$

Errors in Riemann integrals

We define the exact integral as $I = \int_a^b f(x)dx$, and L_n, R_n and M_n represent the left, right and midpoint rule approximations of I based on n intervals.

Writing $f_{\max}^{(k)}$ for the maximum value of the k-th derivative, the upper-bounds of the errors by Riemann integrals are:

- $|I - L_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - R_n| \leq \dfrac{f_{\max}^{(1)}(b-a)^2}{2n}$
- $|I - M_n| \leq \dfrac{f_{\max}^{(2)}(b-a)^3}{24n^2}$

Note that while $|I - L_n|$ and $|I - R_n|$ give the same *upper-bounds* of the error, this does not mean the same error. Rather, the error is of opposite sign!

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion

Trapezoid rule

Since the sign of the approximation error of the left and right endpoint rules is opposite, we can take the average of these approximations:

$$T_n = \frac{L_n + R_n}{2}$$

Trapezoid rule

Since the sign of the approximation error of the left and right endpoint rules is opposite, we can take the average of these approximations:

$$T_n = \frac{L_n + R_n}{2}$$

The total area is obtained by geometric reconstruction of trapezoids:

$$T_n = \sum_{i=1}^{n} \frac{f(x_{i+1}) + f(x_i)}{2} \Delta x_i$$

Trapezoid rule

Since the sign of the approximation error of the left and right endpoint rules is opposite, we can take the average of these approximations:

$$T_n = \frac{L_n + R_n}{2}$$

The total area is obtained by geometric reconstruction of trapezoids:

$$T_n = \sum_{i=1}^{n} \frac{f(x_{i+1}) + f(x_i)}{2} \Delta x_i$$

Note that this can be rewritten for equidistant intervals:

$$T_n = \frac{b-a}{2n} \left(f(x_0) + 2f(x_1) + \ldots + 2f(x_{n-1}) + f(x_n) \right)$$

Error in trapezoid integration

The trapezoid rule result over n intervals T_n approximates the exact integral $I = \int_a^b f(x)dx$. The upper-bounds of the error is given as:

$$|I - T_n| \leq \frac{f_{\max}^{(2)}(b-a)^3}{12n^2}$$

Error in trapezoid integration

The trapezoid rule result over n intervals T_n approximates the exact integral $I = \int_a^b f(x)dx$. The upper-bounds of the error is given as:

$$|I - T_n| \leq \frac{f^{(2)}_{\max}(b-a)^3}{12n^2}$$

Recall that the midpoint rule approximates with an upper-bound error of

$$|I - M_n| \leq \frac{f^{(2)}_{\max}(b-a)^3}{24n^2}$$

Error in trapezoid integration

The trapezoid rule result over n intervals T_n approximates the exact integral $I = \int_a^b f(x)dx$. The upper-bounds of the error is given as:

$$|I - T_n| \leq \frac{f^{(2)}_{\max}(b-a)^3}{12n^2}$$

Recall that the midpoint rule approximates with an upper-bound error of

$$|I - M_n| \leq \frac{f^{(2)}_{\max}(b-a)^3}{24n^2}$$

The midpoint rule approximation has lower error bounds than the trapezoid rule. A linear function is, however, better approximated by the trapezoid rule.

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion

Towards higher-order integration

Compare how the midpoint and trapezoid functions behave on convex and concave parts of a graph.

Midpoint rule

Trapezoid rule

Towards higher-order integration

Compare how the midpoint and trapezoid functions behave on convex and concave parts of a graph.

Midpoint rule

Trapezoid rule

In convex parts (bending down), the midpoint rule tends to overestimate the integral (trapezoid underestimates). In concave parts (bending up), the midpoint rule tends to underestimate the integral (trapezoid overestimates).

Towards higher-order integration

The errors of the midpoint rule and trapezoid rule behave in a similar way, but have opposite signs.

- Midpoint: $|I - M_n| \leq \dfrac{f_{\max}^{(2)}(b-a)^3}{24n^2}$
- Trapezoid: $|I - T_n| \leq \dfrac{f_{\max}^{(2)}(b-a)^3}{12n^2}$

Towards higher-order integration

The errors of the midpoint rule and trapezoid rule behave in a similar way, but have opposite signs.

- Midpoint: $|I - M_n| \leq \dfrac{f^{(2)}_{\max}(b-a)^3}{24n^2}$
- Trapezoid: $|I - T_n| \leq \dfrac{f^{(2)}_{\max}(b-a)^3}{12n^2}$

For a quadratic function, the errors relate as:

$$|I - M_n| = \frac{1}{2}|I - T_n|$$

Towards higher-order integration

The errors of the midpoint rule and trapezoid rule behave in a similar way, but have opposite signs.

- Midpoint: $|I - M_n| \leq \dfrac{f^{(2)}_{\max}(b-a)^3}{24n^2}$
- Trapezoid: $|I - T_n| \leq \dfrac{f^{(2)}_{\max}(b-a)^3}{12n^2}$

For a quadratic function, the errors relate as:

$$|I - M_n| = \frac{1}{2}|I - T_n|$$

Taking the weighted average of these two yields the Simpson's rule:

$$S_{2n} = \frac{2}{3}M_n + \frac{1}{3}T_n$$

The $2n$ means we have $2n$ subintervals: the n trapezoid intervals are subdivided by the midpoint rule.

Simpson's rule

Consider the interval $i \in [x_0, x_2]$, subdivided in three equidistant interpolation points: x_0, x_1, x_2.

- Midpoint: $M_i = f(\frac{x_0 + x_2}{2})2\Delta x = f(x_1)2\Delta x$
- Trapezoid: $T_i = \frac{f(x_0) + f(x_2)}{2}2\Delta x$
- Simpson: $S_i = \frac{2}{3}M_i + \frac{1}{3}T_i$

Note that M_i and T_i were computed on interval $x_2 - x_0 = 2\Delta x$.

Simpson's rule

Consider the interval $i \in [x_0, x_2]$, subdivided in three equidistant interpolation points: x_0, x_1, x_2.

- Midpoint: $M_i = f(\frac{x_0 + x_2}{2})2\Delta x = f(x_1)2\Delta x$
- Trapezoid: $T_i = \frac{f(x_0) + f(x_2)}{2}2\Delta x$
- Simpson: $S_i = \frac{2}{3}M_i + \frac{1}{3}T_i$

Note that M_i and T_i were computed on interval $x_2 - x_0 = 2\Delta x$.

Now we have:

$$
\begin{aligned}
S_i &= \frac{2}{3}[f(x_1)2\Delta x] + \frac{1}{3}\left[\frac{f(x_0) + f(x_2)}{2}2\Delta x\right] \\
&= \frac{4\Delta x}{3}f(x_1) + \frac{\Delta x}{3}f(x_0) + f(x_2)
\end{aligned}
$$

Simpson's rule

Consider the interval $i \in [x_0, x_2]$, subdivided in three equidistant interpolation points: x_0, x_1, x_2.

- Midpoint: $M_i = f(\frac{x_0 + x_2}{2})2\Delta x = f(x_1)2\Delta x$
- Trapezoid: $T_i = \frac{f(x_0) + f(x_2)}{2} 2\Delta x$
- Simpson: $S_i = \frac{2}{3} M_i + \frac{1}{3} T_i$

Note that M_i and T_i were computed on interval $x_2 - x_0 = 2\Delta x$.

Now we have:

$$
\begin{aligned}
S_i &= \frac{2}{3}[f(x_1)2\Delta x] + \frac{1}{3}\left[\frac{f(x_0) + f(x_2)}{2} 2\Delta x\right] \\
&= \frac{4\Delta x}{3} f(x_1) + \frac{\Delta x}{3} f(x_0) + f(x_2) = \frac{\Delta x}{3}(f(x_0) + 4f(x_1) + f(x_2))
\end{aligned}
$$

Simpson's rule

We write $f(x_k) = f_k$. The integral of an interval $i \in [x_0, x_2]$ is approximated as:

$$S_i = \frac{\Delta x}{3}(f_0 + 4f_1 + f_2)$$

Simpson's rule

We write $f(x_k) = f_k$. The integral of an interval $i \in [x_0, x_2]$ is approximated as:

$$S_i = \frac{\Delta x}{3}(f_0 + 4f_1 + f_2)$$

The next interval, S_j with $j \in [x_2, x_4]$ with midpoint $x_3 = \frac{x_2+x_4}{2}$ is approximated as:

$$S_j = \frac{\Delta x}{3}(f_2 + 4f_3 + f_4)$$

Simpson's rule

We write $f(x_k) = f_k$. The integral of an interval $i \in [x_0, x_2]$ is approximated as:

$$S_i = \frac{\Delta x}{3}(f_0 + 4f_1 + f_2)$$

The next interval, S_j with $j \in [x_2, x_4]$ with midpoint $x_3 = \frac{x_2+x_4}{2}$ is approximated as:

$$S_j = \frac{\Delta x}{3}(f_2 + 4f_3 + f_4)$$

If we sum these two intervals we obtain:

$$\begin{aligned} I \approx S_i + S_j &= \left[\frac{\Delta x}{3}(f_0 + 4f_1 + f_2)\right] + \left[\frac{\Delta x}{3}(f_2 + 4f_3 + f_4)\right] \\ &= \frac{\Delta x}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + f4) \end{aligned}$$

Simpson's rule

In general, Simpson's rule can be written as:

$$\int_a^b f(x)dx \approx \sum_{\substack{k=2 \\ k\,\text{even}}}^{n} \frac{\Delta x}{3}\left(f_{k-2} + 4f_{k-1} + f_k\right)$$

$$= \frac{\Delta x}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 2f_{n-2} + 4f_{n-1} + f_n\right)$$

Simpson's rule

In general, Simpson's rule can be written as:

$$\int_a^b f(x)dx \approx \sum_{\substack{k=2 \\ k\,\text{even}}}^{n} \frac{\Delta x}{3}(f_{k-2} + 4f_{k-1} + f_k)$$

$$= \frac{\Delta x}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 2f_{n-2} + 4f_{n-1} + f_n)$$

The error is given by:

$$|I - S_n| \leq \frac{f_{\max}^{(4)}(b-a)^5}{180n^4}$$

if integrand f is differentiable on $[a, b]$.

Simpson's rule: example

Recall our example data, described by $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\ldots$

Simpson's rule: example

Recall our example data, described by $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\ldots$

- Interpolating x_0, x_1 and x_2:
  $p_{2a}(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$
  $\int_0^2 p_{2a} = \frac{55}{9} \approx 6.1111$

Simpson's rule: example

Recall our example data, described by $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\ldots$

- Interpolating x_0, x_1 and x_2:
  $p_{2a}(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$
  $\int_0^2 p_{2a} = \frac{55}{9} \approx 6.1111$
- Interpolating x_2, x_3 and x_4:
  $p_{2b}(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$
  $\int_2^4 p_{2b} = \frac{25}{9} \approx 2.777\ldots$

Simpson's rule: example

Recall our example data, described by $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\ldots$

- Interpolating x_0, x_1 and x_2:
  $p_{2a}(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$
  $\int_0^2 p_{2a} = \frac{55}{9} \approx 6.1111$
- Interpolating x_2, x_3 and x_4:
  $p_{2b}(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$
  $\int_2^4 p_{2b} = \frac{25}{9} \approx 2.777\ldots$
- Adding the separate integrals:
  $\int_0^2 p_{2a} + \int_2^4 p_{2b} = \frac{80}{9}$

Simpson's rule: example

Recall our example data, described by $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\ldots$

- Interpolating x_0, x_1 and x_2:
  $p_{2a}(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$
  $\int_0^2 p_{2a} = \frac{55}{9} \approx 6.1111$
- Interpolating x_2, x_3 and x_4:
  $p_{2b}(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$
  $\int_2^4 p_{2b} = \frac{25}{9} \approx 2.777\ldots$
- Adding the separate integrals:
  $\int_0^2 p_{2a} + \int_2^4 p_{2b} = \frac{80}{9}$

Using Simpson's rule: $I \approx \frac{\Delta x}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + f_4) = \frac{1}{3}(1 + 4 \cdot 3.6667 + 2 \cdot 2.6667 + 4 \cdot 1.0000 + 1.6667) = 8.88888 = \frac{80}{9}$

Simpson's rule: example

Recall our example data, described by $f(x) = \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1$

$I = \int_0^4 \frac{x^3}{2} - \frac{10x^2}{3} + \frac{11x}{2} + 1 = \frac{80}{9} \approx 8.888\ldots$

- Interpolating x_0, x_1 and x_2:
  $p_{2a}(x) = -\frac{11}{6}x^2 + 4\frac{1}{2}x + 1$
  $\int_0^2 p_{2a} = \frac{55}{9} \approx 6.1111$
- Interpolating x_2, x_3 and x_4:
  $p_{2b}(x) = \frac{7x^2}{6} - 7\frac{1}{2}x + 13$
  $\int_2^4 p_{2b} = \frac{25}{9} \approx 2.777\ldots$
- Adding the separate integrals:
  $\int_0^2 p_{2a} + \int_2^4 p_{2b} = \frac{80}{9}$

Using Simpson's rule: $I \approx \frac{\Delta x}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + f_4) = \frac{1}{3}(1 + 4 \cdot 3.6667 + 2 \cdot 2.6667 + 4 \cdot 1.0000 + 1.6667) = 8.88888 = \frac{80}{9}$

Simpson's method is of fourth order, and it gives exact approximations of third order polynomials!

Integration in Matlab

Integration can be done numerically in Matlab.

- `trapz(x,y)` uses the trapezoid rule to integrate the data. Make sure you use the `x` variable if your data is not spaced with $\Delta x = 1$. Can handle non-equidistant data.
- Integration of functions can be done using the `integral(fun,xmin,xmax)` function:

```
fun = @(x) exp(-x.^2);
I = integral(fun,0,10)
I =
   0.886226925452758
```

Today's outline

- Introduction
- Riemann integrals
- Trapezoid rule
- Simpson's rule
- Conclusion

What hasn't been discussed?

This course is by no means complete, and further reading is possible.

- Legendre polynomials: Another way of performing the polynomial interpolation
- Gaussian quadrature: A third-order integration method that requires only two base points (in contrast to the third order Simpson's method, which requires three points)
- Adaptive techniques: Parts of a function that are relatively steady (no wild oscillations) and differentiable can be integrated with much larger step sizes than other parts of the function.
- Simpson's 3/8-rule: Yet another integration technique, requiring an additional data point

Summary

- Interpolation is used to obtain data between existing data points
  - (Bi-)Linear, polynomial and spline interpolation methods
  - Construction of Newton polynomials
  - Oscillations of high-order polynomials
- Several techniques for numerical integration were discussed:
  - Riemann sums, trapezoid rule, Simpson's rule
  - Upper-bound errors were given for each technique

Ordinary differential equations 1

Explicit techniques for ODEs

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Matlab

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Matlab

Overview

Ordinary differential equations

An equation containing a function of one independent variable and its derivatives, in contrast to a *partial differential equation*, which contains derivatives with respect to more independent variables.

Overview

Ordinary differential equations

An equation containing a function of one independent variable and its derivatives, in contrast to a *partial differential equation*, which contains derivatives with respect to more independent variables.

Main question

How to solve

$$\frac{d\boldsymbol{y}}{dx} = f(\boldsymbol{y}(x), x) \quad \text{with} \quad \boldsymbol{y}(x=0) = \boldsymbol{y}_0$$

accurately and efficiently?

What is an ODE?

- Algebraic equation:

$$f(y(x),x)=0 \qquad \text{e.g. } -\ln(K_{eq})=(1-\zeta)$$

- First order ODE:

$$f\left(\frac{dy}{dx}(x),y(x),x\right)=0 \quad \text{e.g. } \frac{dc}{dt}=-kc^n$$

- Second order ODE:

$$f\left(\frac{d^2y}{dx^2}(x),\frac{dy}{dx}(x),y(x),x\right)=0 \quad \text{e.g. } \mathcal{D}\frac{d^2c}{dx^2}=-\frac{kc}{1+Kc}$$

About second order ODEs

Very often a second order ODE can be rewritten into a system of first order ODEs (whether it is handy depends on the boundary conditions!)

About second order ODEs

Very often a second order ODE can be rewritten into a system of first order ODEs (whether it is handy depends on the boundary conditions!)

Example

Recall:

$$\mathcal{D}\frac{d^2c}{dx^2} = -\frac{kc}{1+Kc}$$

Define $y = -\mathcal{D}\frac{dc}{dx}$, then $\frac{dy}{dx} = \frac{kc}{1+Kc}$, thus solve system:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}y$$
$$\frac{dy}{dx} = \frac{kc}{1+Kc}$$

About second order ODEs

Very often a second order ODE can be rewritten into a system of first order ODEs (whether it is handy depends on the boundary conditions!)

More general

Consider the second order ODE:

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

Now define and solve using z as a new variable:

$$\frac{dy}{dx} = z(x)$$
$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

Importance of boundary conditions

The nature of boundary conditions determines the appropriate numerical method. Classification into 2 main categories:

- *Initial value problems (IVP)*
  We know the values of all y_i at some starting position x_s, and it is desired to find the values of y_i at some final point x_f.

- *Boundary value problems (BVP)*
  Boundary conditions are specified at more than one x. Typically, some of the BC are specified at x_s and the remainder at x_f.

Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Overview

Initial value problems:

- Explicit methods
  - First order: forward Euler
  - Second order: improved Euler (RK2)
  - Fourth order: Runge-Kutta 4 (RK4)
  - Step size control
- Implicit methods
  - First order: backward Euler
  - Second order: midpoint rule

Boundary value problems

- Shooting method

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Matlab

Euler's method

Consider the following single initial value problem:

$$\frac{dc}{dt} = f(c(t), t) \quad \text{with} \quad c(t=0) = c_0 \quad \text{(initial value problem)}$$

Euler's method

Consider the following single initial value problem:

$$\frac{dc}{dt} = f(c(t), t) \quad \text{with} \quad c(t=0) = c_0 \quad \text{(initial value problem)}$$

Easiest solution algorithm: Euler's method, derived here via Taylor series expansion:

$$c(t_0 + \Delta t) \approx c(t_0) + \left.\frac{dc}{dt}\right|_{t_0} \Delta t + \frac{1}{2} \left.\frac{d^2c}{dt^2}\right|_{t_0} (\Delta t)^2 + \mathcal{O}(\Delta t^3)$$

Euler's method

Consider the following single initial value problem:

$$\frac{dc}{dt} = f(c(t), t) \quad \text{with} \quad c(t=0) = c_0 \quad \text{(initial value problem)}$$

Easiest solution algorithm: Euler's method, derived here via Taylor series expansion:

$$c(t_0 + \Delta t) \approx c(t_0) + \left.\frac{dc}{dt}\right|_{t_0} \Delta t + \frac{1}{2} \left.\frac{d^2c}{dt^2}\right|_{t_0} (\Delta t)^2 + \mathcal{O}(\Delta t^3)$$

Neglect terms with higher order than two: $\left.\frac{dc}{dt}\right|_{t_0} = \frac{c(t_0+\Delta t)-c(t_0)}{\Delta t}$ Substitution:

$$\frac{c(t_0 + \Delta t) - c(t_0)}{\Delta t} = f(c_0, t_0) \Rightarrow c(t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$$

Euler's method: graphical example

$$\frac{c(t_0+\Delta t)-c(t_0)}{\Delta t} = f(c_0,t_0) \Rightarrow c(t_0+\Delta t) = c(t_0) + \Delta t f(c_0,t_0)$$

Euler's method - solution method

c_0

Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method

Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method

Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method

Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Euler's method - solution method

Start with $t = t_0$, $c = c_0$, then calculate at discrete points in time:
$c(t_1 = t_0 + \Delta t) = c(t_0) + \Delta t f(c_0, t_0)$.

Pseudo-code Euler's method: $\frac{dy}{dx} = f(x,y)$ and $y(x_0) = y_0$.

1. Initialize variables, functions; set $h = \frac{x_1 - x_0}{N}$
2. Set $x = x_0$, $y = y_0$
3. While $x < x_{\text{end}}$ do
   $x_{i+1} = x_i + h$; $y_{i+1} = y_i + hf(x_i, y_i)$

Euler's method - example

First order reaction in a batch reactor:

$$\frac{dc}{dt} = -kc \quad \text{with} \quad c(t=0) = 1\,\mathrm{mol\,m^{-3}}, \quad k = 1\,\mathrm{s^{-1}}, \quad t_{\mathrm{end}} = 2\,\mathrm{s}$$

Euler's method - example

First order reaction in a batch reactor:

$$\frac{dc}{dt} = -kc \quad \text{with} \quad c(t=0) = 1\,\mathrm{mol\,m^{-3}}, \quad k = 1\,\mathrm{s^{-1}}, \quad t_{\mathrm{end}} = 2\,\mathrm{s}$$

Time [s]	Concentration [mol m^{-3}]
$t_0 = 0$	$c_0 = 1.00$
$t_1 = t_0 + \Delta t$	$c_1 = c_0 + \Delta t \cdot (-kc_0)$
$= 0 + 0.1 = 0.1$	$= 1 + 0.1 \cdot (-1 \cdot 1) = 0.9$
$t_2 = t_1 + \Delta t$	$c_2 = c_1 + \Delta t \cdot (-kc_1)$
$= 0.1 + 0.1 = 0.2$	$= 0.9 + 0.1 \cdot (-1 \cdot 0.9) = 0.81$
$t_3 = t_2 + \Delta t$	$c_3 = c_2 + \Delta t \cdot (-kc_2)$
$= 0.2 + 0.1 = 0.3$	$= 0.81 + 0.1 \cdot (-1 \cdot 0.81) = 0.729$
...	...
$t_{i+1} = t_i + \Delta t$	$c_{i+1} = c_i + \Delta t \cdot (-kc_i)$
...	...
$t_{20} = 2.0$	$c_{20} = c_{19} + \Delta t \cdot (-kc_{19}) = 0.121577$

Euler's method - example

Euler's method - example

Euler's method - example

Euler's method - example

Euler's method - example

Problems with Euler's method

The question is: What step size, or how many steps to use?

1. *Accuracy* ⇒ need information on numerical error!
2. *Stability* ⇒ need information on stability limits!

Reaction rate: $k = 1\ \mathrm{s}^{-1}$

Reaction rate: $k = 50\ \mathrm{s}^{-1}$

Accuracy

Comparison with analytical solution for $k = 1\ \mathrm{s}^{-1}$:

$$c(t) = c_0 \exp(-kt) \Rightarrow \zeta = 1 - \exp(-kt) \Rightarrow \zeta_{\text{analytical}} = 0.864665$$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$
20	0.878423	0.015912
40	0.871488	0.007891
80	0.868062	0.003929
160	0.866360	0.001961
320	0.865511	0.000979

Accuracy

For Euler's method: Error halves when the number of grid points is doubled, i.e. error is proportional to Δt: first order method.

Error estimate:

$$\left.\frac{dx}{dt}\right|_{t_0} = \frac{x(t_0+\Delta t)-x(t_0)}{\Delta t} + \frac{1}{2}\left.\frac{d^2x}{dt^2}\right|_{t_0}(\Delta t) + \mathcal{O}(\Delta t)^2$$

$$\frac{x(t_0+\Delta t)-x(t_0)}{\Delta t} = f(x_0,t_0) - \frac{1}{2}\left.\frac{d^2x}{dt^2}\right|_{t_0}(\Delta t) + \mathcal{O}(\Delta t)^2$$

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \to 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

What to do when there is no analytical solution available?

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \to 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

What to do when there is no analytical solution available? Compare to calculations with different number of steps: $\epsilon_1 = c(\Delta x_1)^r$ and $\epsilon_2 = c(\Delta x_2)^r$ and solve for r:

$$\frac{\epsilon_2}{\epsilon_1} = \frac{c(\Delta x_2)^r}{c(\Delta x_1)^r} = \left(\frac{\Delta x_2}{\Delta x_1}\right)^r \Rightarrow \log\left(\frac{\epsilon_2}{\epsilon_1}\right) = \log\left(\frac{\Delta x_2}{\Delta x_1}\right)^r$$

$$\Rightarrow \quad r \quad = \quad \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{\Delta x_2}{\Delta x_1}\right)} \quad = \quad \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{N_1}{N_2}\right)} \quad \text{in the limit of} \quad \Delta x \;\to\; 0 \qquad \text{or} \qquad N \;\to\; \infty$$

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Matlab

Errors and convergence rate

L_2 norm (Euclidean norm)

$$\|\boldsymbol{v}\|_2 = \sqrt{v_1^2 + v_2^2 + \ldots + v_n^2} = \sqrt{\sum_{i=1}^{n} v_i^2}$$

L_∞ norm (maximum norm)

$$\|\boldsymbol{v}\|_\infty = \max(|v_1|, \ldots, |v_n|)$$

Absolute difference

$$\epsilon_{\text{abs}} = \left\| \boldsymbol{y}_{\text{numerical}} - \boldsymbol{y}_{\text{analytical}} \right\|_{2,\infty}$$

Relative difference

$$\epsilon_{\text{rel}} = \left\| \frac{\boldsymbol{y}_{\text{numerical}} - \boldsymbol{y}_{\text{analytical}}}{\boldsymbol{y}_{\text{analytical}}} \right\|_{2,\infty}$$

Errors and convergence rate

Convergence rate (or: order of convergence) r

$$\epsilon = \lim_{\Delta x \to 0} c(\Delta x)^r$$

- A first order method reduces the error by a factor 2 when increasing the number of steps by a factor 2
- A second order method reduces the error by a factor 4 when increasing the number of steps by a factor 2

Computing the rate of convergence

When the analytical solution is available, choose 1 or 2 for a particular number of grid points N:

1. Compute the relative or absolute error vector $\overline{\varepsilon}$. Take the norm to compute a single error value ϵ following:
   - Based on L_1-norm: $\epsilon = \dfrac{\|\overline{\varepsilon}\|_1}{N}$
   - Based on L_2-norm: $\epsilon = \dfrac{\|\overline{\varepsilon}\|_2}{\sqrt{N}}$
   - Based on L_∞-norm: $\epsilon = \|\overline{\varepsilon}\|_\infty$
2. Compute the relative or absolute error at a single indicative points (e.g. middle of domain, outlet).

Computing the rate of convergence

When the analytical solution is available, choose 1 or 2 for a particular number of grid points N:

1. Compute the relative or absolute error vector $\overline{\varepsilon}$. Take the norm to compute a single error value ϵ following:
   - Based on L_1-norm: $\epsilon = \dfrac{\|\overline{\varepsilon}\|_1}{N}$
   - Based on L_2-norm: $\epsilon = \dfrac{\|\overline{\varepsilon}\|_2}{\sqrt{N}}$
   - Based on L_∞-norm: $\epsilon = \|\overline{\varepsilon}\|_\infty$
2. Compute the relative or absolute error at a single indicative points (e.g. middle of domain, outlet).

Compare to calculations with different number of steps: $\epsilon_1 = c(\Delta x_1)^r$ and $\epsilon_2 = c(\Delta x_2)^r$ and solve for r:

$$\frac{\epsilon_2}{\epsilon_1} = \frac{c(\Delta x_2)^r}{c(\Delta x_1)^r} = \left(\frac{\Delta x_2}{\Delta x_1}\right)^r \Rightarrow \log\left(\frac{\epsilon_2}{\epsilon_1}\right) = \log\left(\frac{\Delta x_2}{\Delta x_1}\right)^r$$

$$\Rightarrow \; r \; = \; \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{\Delta x_2}{\Delta x_1}\right)} \; = \; \frac{\log\left(\frac{\epsilon_2}{\epsilon_1}\right)}{\log\left(\frac{N_1}{N_2}\right)} \quad \text{in the limit of} \quad \Delta x \;\rightarrow\; 0 \quad \text{or} \quad N \;\rightarrow\; \infty$$

Computing the rate of convergence

When the analytical solution is not available:

1. Compute the solution with $N+1$, N, $N-1$ and $N-2$ grid points
2. Select a single indicative grid point (e.g. middle of domain, outlet) that lies at exactly the same position in each computation
3. Use the solution c at this grid point for various grid sizes to compute:

$$r = \frac{\log \dfrac{c_{N+1} - c_N}{c_N - c_{N-1}}}{\log \dfrac{c_N - c_{N-1}}{c_{N-1} - c_{N-2}}}$$

4. Alternative for simulations with $2N$, N and $\frac{N}{2}$ grid points:

$$r = \frac{\log \left| \dfrac{c_{2N} - c_N}{c_N - c_{\frac{N}{2}}} \right|}{\log \left| \dfrac{2N}{N} \right|}$$

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

$\Rightarrow$ Euler's method is a first order method (as we already knew from the truncation error analysis)

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

$\Rightarrow$ Euler's method is a first order method (as we already knew from the truncation error analysis)

Wouldn't it be great to have a method that can give the answer using much less steps?

Example: Euler's method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.878423	0.015912	—
40	0.871488	0.007891	1.011832
80	0.868062	0.003929	1.005969
160	0.866360	0.001961	1.002996
320	0.865511	0.000979	1.001500

$\Rightarrow$ Euler's method is a first order method (as we already knew from the truncation error analysis)

Wouldn't it be great to have a method that can give the answer using much less steps? $\Rightarrow$ Higher order methods

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Matlab

Runge-Kutta methods

Propagate a solution by combining the information of several Euler-style steps (each involving one function evaluation) to match a Taylor series expansion up to some higher order.

Euler: $y_{i+1} = y_i + hf(x_i, y_i)$ with $h = \Delta x$, i.e. slope $= k_1 = f(x_i, y_i)$.

Euler's method

RK2 method

Classical second order Runge-Kutta (RK2) method

This method is also called Heun's method, or improved Euler method:

1. Approximate the slope at x_i: $k_1 = f(x_i, y_i)$
2. Approximate the slope at x_{i+1}: $k_2 = f(x_{i+1}, y_{i+1})$ where we use Euler's method to approximate $y_{i+1} = y_i + hf(x_i, y_i) = y_i + hk_1$
3. Perform an Euler step with the average of the slopes: $y_{i+1} = y_i + h\frac{1}{2}(k_1 + k_2)$

Classical second order Runge-Kutta (RK2) method

This method is also called Heun's method, or improved Euler method:

1. Approximate the slope at x_i: $k_1 = f(x_i, y_i)$
2. Approximate the slope at x_{i+1}: $k_2 = f(x_{i+1}, y_{i+1})$ where we use Euler's method to approximate $y_{i+1} = y_i + hf(x_i, y_i) = y_i + hk_1$
3. Perform an Euler step with the average of the slopes: $y_{i+1} = y_i + h\frac{1}{2}(k_1 + k_2)$

In pseudocode:

```
x = x_0, y = y_0
while x < x_end do
    x_{i+1} = x_i + h
    k_1 = f(x_i, y_i)
    k_2 = f(x_i + h, y_i + hk_1)
    y_{i+1} = y_i + h 1/2 (k_1 + k_2)
end while
```

Runge-Kutta methods — derivation

$$\frac{dy}{dx} = f(x, y(x))$$

Runge-Kutta methods — derivation

$$\frac{dy}{dx} = f(x, y(x))$$

Using Taylor series expansion: $y_{i+1} = y_i + h \left.\frac{dy}{dx}\right|_i + \frac{h^2}{2} \left.\frac{d^2y}{dx^2}\right|_i + \mathcal{O}(h^3)$

$$\left.\frac{dy}{dx}\right|_i = f(x_i, y_i) \equiv f_i$$

$$\left.\frac{d^2y}{dx^2}\right|_i = \left.\frac{d}{dx} f(x, y(x))\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i \left.\frac{\partial y}{\partial x}\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \quad \text{(chain rule)}$$

Runge-Kutta methods — derivation

$$\frac{dy}{dx} = f(x, y(x))$$

Using Taylor series expansion: $y_{i+1} = y_i + h \left.\frac{dy}{dx}\right|_i + \frac{h^2}{2} \left.\frac{d^2y}{dx^2}\right|_i + \mathcal{O}(h^3)$

$$\left.\frac{dy}{dx}\right|_i = f(x_i, y_i) \equiv f_i$$

$$\left.\frac{d^2y}{dx^2}\right|_i = \left.\frac{d}{dx} f(x, y(x))\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i \left.\frac{\partial y}{\partial x}\right|_i = \left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \quad \text{(chain rule)}$$

Substitution gives:

$$y_{i+1} = y_i + hf_i + \frac{h^2}{2}\left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

$$y_{i+1} = y_i + \frac{h}{2} f_i + \frac{h}{2}\left(f_i + h \left.\frac{\partial f}{\partial x}\right|_i + hf_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)$$

Runge-Kutta methods — derivation

Note multivariate Taylor expansion:

$$f(x_i + h, y_i + k) = f_i + h \left.\frac{\partial f}{\partial x}\right|_i + k \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\Rightarrow \frac{h}{2}\left(f_i + h \left.\frac{\partial f}{\partial x}\right|_i + h f_i \left.\frac{\partial f}{\partial y}\right|_i \right) = \frac{h}{2} f\left(x_i + h, y_i + h f_i\right) + \mathcal{O}(h^3)$$

Concluding:

$$y_{i+1} = y_i + \frac{h}{2} f_i + \frac{h}{2} f\left(x_i + h, y_i + h f_i\right) + \mathcal{O}(h^3)$$

Rewriting:

$$\begin{aligned} k_1 &= f\left(x_i, y_i\right) \\ k_2 &= f\left(x_i + h, y_i + h k_1\right) \\ \Rightarrow y_{i+1} &= y_i + \frac{h}{2}(k_1 + k_2) \end{aligned}$$

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i$, $k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i$, $k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_i + c_2 h, y_i + a_{2,1} h k_1) = f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\begin{aligned}
y_{i+1} &= y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3) \\
&= y_i + h\Big[b_1 f_i + b_2 f(x_i + c_2 h, y_1 + a_{2,1} h k_1)\Big] + \mathcal{O}(h^3) \\
&= y_i + h\left[b_1 f_i + b_2 \left\{ f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2) \right\}\right] + \mathcal{O}(h^3) \\
&= y_i + h(b_1 + b_2) f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)
\end{aligned}$$

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i,\ k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_i + c_2 h, y_i + a_{2,1} h k_1) = f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\begin{aligned}
y_{i+1} &= y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3) \\
&= y_i + h\Big[b_1 f_i + b_2 f(x_i + c_2 h, y_1 + a_{2,1} h k_1)\Big] + \mathcal{O}(h^3) \\
&= y_i + h\left[b_1 f_i + b_2 \left\{ f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2) \right\}\right] + \mathcal{O}(h^3) \\
&= y_i + h(b_1 + b_2) f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)
\end{aligned}$$

Comparison with Taylor:

$$y_{i+1} = y_i + h f_i + \frac{h^2}{2} \left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

Using $b_1 + b_2 = 1,\ c_2 b_2 = \frac{1}{2},\ a_{2,1} b_2 = \frac{1}{2} \Rightarrow$ 3 eqns and 4 unknowns $\Rightarrow$ multiple possibilities!

Runge-Kutta methods — derivation

Generalization: $y_{i+1} = y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3)$
with $k_1 = f_i,\ k_2 = f(x_i + c_2 h, y_1 + a_{2,1} h k_1)$
(Note that classical RK2: $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$.)

Bivariate Taylor expansion:

$$f(x_i + c_2 h, y_i + a_{2,1} h k_1) = f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2)$$

$$\begin{aligned}
y_{i+1} &= y_i + h(b_1 k_1 + b_2 k_2) + \mathcal{O}(h^3) \\
&= y_i + h\Big[b_1 f_i + b_2 f(x_i + c_2 h, y_1 + a_{2,1} h k_1)\Big] + \mathcal{O}(h^3) \\
&= y_i + h\left[b_1 f_i + b_2 \left\{ f_i + c_2 h \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} h k_1 \left.\frac{\partial f}{\partial y}\right|_i + \mathcal{O}(h^2) \right\}\right] + \mathcal{O}(h^3) \\
&= y_i + h(b_1 + b_2) f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)
\end{aligned}$$

Comparison with Taylor:

$$y_{i+1} = y_i + h f_i + \frac{h^2}{2} \left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

Using $b_1 + b_2 = 1,\ c_2 b_2 = \frac{1}{2},\ a_{2,1} b_2 = \frac{1}{2} \Rightarrow$ 3 eqns and 4 unknowns $\Rightarrow$ multiple possibilities!

Runge-Kutta methods — derivation

$$y_{i+1} = y_i + h(b_1 + b_2)f_i + h^2 b_2 \left(c_2 \left.\frac{\partial f}{\partial x}\right|_i + a_{2,1} f_i \left.\frac{\partial f}{\partial y}\right|_i \right) + \mathcal{O}(h^3)$$

$$y_{i+1} = y_i + hf_i + \frac{h^2}{2} \left(\left.\frac{\partial f}{\partial x}\right|_i + \left.\frac{\partial f}{\partial y}\right|_i f_i \right) + \mathcal{O}(h^3)$$

$\Rightarrow$ 3 eqns and 4 unknowns $\Rightarrow$ multiple possibilities!

1. Classical RK2:
   $b_1 = b_2 = \frac{1}{2}$ and $c_2 = a_{2,1} = 1$
2. Midpoint rule (modified Euler):
   $b_1 = 0,\ b_2 = 1,\ c_2 = a_{2,1} = \frac{1}{2}$

Second order Runge-Kutta methods

Classical RK2 method (= Heun's method, improved Euler method)	Explicit midpoint rule (modified Euler method)
$k_1 = f_i$	$k_1 = f_i$
$k_2 = f(x_i + h, y_i + hk_1)$	$k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$
$y_{i+1} = y_i + \frac{1}{2}h(k_1 + k_2)$	$y_{i+1} = y_i + hk_2$

Second order Runge-Kutta method — Example

First order reaction in a batch reactor: $\frac{dc}{dt} = -kc$ with $c(t=0) = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{s^{-1}}$, $t_{\mathrm{end}} = 2$ s.

Time [s]	C [mol m^{-3}]	$k_1 = hf(x_i, y_i)$	$k_2 = hf(x_i + \frac{1}{2}h, y_n + \frac{1}{2}k_1)$
0	1.00	$0.1 \cdot (-1 \cdot 1) = -0.1$	$0.1 \cdot (-1 \cdot (1 - 0.5 \cdot 0.1)) = -0.095$
0.1	$1 - 0.095 = 0.905$	$0.1 \cdot (-1 \cdot 0.0905) = -0.0905$	$0.1 \cdot (-1 \cdot (0.905 - 0.5 \cdot 0.0905)) = 0.085975$
...	...	...	...
2	0.1358225	−0.0135822	−0.0129031

RK2 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864178	5.634×10^{-4}	—
40	0.864548	1.355×10^{-4}	2.056
80	0.864636	3.323×10^{-5}	2.028
160	0.864658	8.229×10^{-6}	2.014
320	0.864663	2.048×10^{-6}	2.007

RK2 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864178	5.634×10^{-4}	—
40	0.864548	1.355×10^{-4}	2.056
80	0.864636	3.323×10^{-5}	2.028
160	0.864658	8.229×10^{-6}	2.014
320	0.864663	2.048×10^{-6}	2.007

$\Rightarrow$ RK2 is a second order method. Doubling the number of cells reduces the error by a factor 4!

Can we do even better?

RK4 method (classical fourth order Runge-Kutta method)

$$k_1 = f(x_i, y_i)$$

$$k_2 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$$

$$k_3 = f(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_2)$$

$$k_4 = f(x_i + h, y_i + hk_3)$$

$$y_{i+1} = y_i + h\left(\frac{1}{6}k_1 + \frac{1}{3}(k_2 + k_3) + \frac{1}{6}k_4\right)$$

RK4 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864664472	2.836×10^{-7}	—
40	0.864664702	1.700×10^{-8}	4.060
80	0.864664716	1.040×10^{-9}	4.030
160	0.864664717	6.435×10^{-11}	4.015
320	0.864664717	4.001×10^{-12}	4.007

RK4 method — order of convergence

N	ζ	$\frac{\zeta_{\text{numerical}}-\zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.864664472	2.836×10^{-7}	—
40	0.864664702	1.700×10^{-8}	4.060
80	0.864664716	1.040×10^{-9}	4.030
160	0.864664717	6.435×10^{-11}	4.015
320	0.864664717	4.001×10^{-12}	4.007

$\Rightarrow$ RK4 is a fourth order method: Doubling the number of cells reduces the error by a factor 16!

Can we do even better?

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Matlab

Adaptive step size control

The step size (be it either position, time or both (PDEs)) cannot be decreased indefinitely to favour a higher accuracy, since each additional grid point causes additional computation time. It may be wise to adapt the step size according to the computation requirements.

Globally two different approaches can be used:

1. Step doubling: compare solutions when taking one full step or two consecutive halve steps
2. Embedded methods: Compare solutions when using two approximations of different order

Adaptive step size control: step doubling

Adaptive step size control: step doubling

- RK4 with one large step of h: $y_{i+1} = y_1 + ch^5 + \mathcal{O}(h^6)$

Adaptive step size control: step doubling

- RK4 with one large step of h: $y_{i+1} = y_1 + ch^5 + \mathcal{O}(h^6)$
- RK4 with two steps of $\frac{1}{2}h$: $y_{i+1} = y_2 + 2c(\frac{1}{2}h)^5 + \mathcal{O}(h^6)$

Adaptive step size control: step doubling

- Estimation of truncation error by comparing y_1 and y_2:
  $\Delta = y_2 - y_1$
- If Δ too large, reduce step size for accuracy
- If Δ too small, increase step size for efficiency.
- Ignoring higher order terms and solving for c:
  $\Delta = \frac{15}{16} c h^5 \Rightarrow c h^5 = \frac{16}{15}\Delta \Rightarrow y_{i+1} = y_2 + \frac{\Delta}{15} + \mathcal{O}(h^6)$
  (local Richardson extrapolation)

Note that when we specify a tolerance *tol*, we can estimate the maximum allowable step size as: $h_{\text{new}} = \alpha h_{\text{old}} \left| \frac{\text{tol}}{\Delta} \right|^{\frac{1}{5}}$ with α a safety factor (typically $\alpha = 0.9$).

Adaptive step size control: embedded methods

Use a special fourth and a fifth order Runge Kutta method to approximate y_{i+1}

- The fourth order method is special because we want to use the same positions for the evaluation for computational efficiency.
- RK45 is the preferred method (minimum number of function evaluations) (this is built in Matlab as `ode45`).

Today's outline

- Introduction
- Euler's method
  - Forward Euler
- Rates of convergence
- Runge-Kutta methods
  - RK2 methods
  - RK4 method
- Step size control
- Solving ODEs in Matlab

Solving ODEs in Matlab

Matlab provides convenient procedures to solve (systems of) ODEs automatically.

The procedure is as follows:

1. Create a function that specifies the ODE(s). Specifically, this function returns the $\frac{dy}{dx}$ value (vector).
2. Initialise solver variables and settings (e.g. step size, initial conditions, tolerance), in a separate script
3. Call the ODE solver function, using a *function handle* to the ODE function described in point 1.
   - The ODE solver will return the vector for the independent variable, and a solution vector (matrix for systems of ODEs).

Solving ODEs in Matlab: example 1

We solve the system: $\frac{dx}{dt} = -k_1 x + k_2, k_1 = 0.2, k_2 = 2.5$

- Create an anonymous function handle:

```
>> myEqn = @(t,x) (-0.2*x + 2.5)
```

- Solve with a call to `ode45(function_handle, timespan, initial_condition)`:

```
>> ode45(myEqn, [0 40], 0);
```

- By omitting the output of this function, the graph is automatically drawn.

Solving ODEs in Matlab: example 2

We solve the system: $\dfrac{dx}{dt} = \begin{cases} -\dfrac{k_1}{x^2} & t \leq 10 \\ \dfrac{k_2}{x} - \dfrac{k_1}{x^2} & t > 10 \end{cases}$ with $k_1 = 0.5$, $k_2 = 1$, $x(0) = 2$

Create an ODE function

```
function [dxdt] = myEqnFunction(t,x)
k1 = 0.5;
k2 = 1;
dxdt = (t>10)*k2/x - k1/x^2;
```

Solving ODEs in Matlab: example 2

We solve the system: $\dfrac{dx}{dt} = \begin{cases} -\dfrac{k_1}{x^2} & t \leq 10 \\ \dfrac{k_2}{x} - \dfrac{k_1}{x^2} & t > 10 \end{cases}$ with $k_1 = 0.5$, $k_2 = 1$, $x(0) = 2$

Create an ODE function

```
function [dxdt] = myEqnFunction(t,x)
k1 = 0.5;
k2 = 1;
dxdt = (t>10)*k2/x - k1/x^2;
```

Create a solution script

```
tspan = [0 20];        % Time span
xinit = 2;             % Initial condition
options = odeset('RelTol',1e-8,'AbsTol',1e-8);
[t,x] = ode45(@myEqnFunction,tspan,x_init,options);
```

Solving ODEs in Matlab: example 2

Plot the solution:

```
plot(t,x,'r-x')
```

Solving ODEs in Matlab: example 2

Plot the solution:

```
plot(t,x,'r-x')
```

Note the refinement in regions where large changes occur.

Solving ODEs in Matlab: example

A few notes on working with `ode45` and other solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
function [dxdt] = myEqn(t,x,k1,k2)
```

The additional arguments can now be set in the solver script by *adding them after the options*:

```
[t,x] = ode45(@myEqn,tspan,x_0,options,k1,k2);
```

Solving ODEs in Matlab: example

A few notes on working with `ode45` and other solvers. If we want to give additional arguments (e.g. `k1` and `k2`) to our ODE function, we can list them in the function line:

```
function [dxdt] = myEqn(t,x,k1,k2)
```

The additional arguments can now be set in the solver script by *adding them after the options*:

```
[t,x] = ode45(@myEqn,tspan,x_0,options,k1,k2);
```

- Of course, in the solver script, the variables do not have to be called `k1` and `k2`:

  ```
  [t,x] = ode45(@myODE,tspan,x_0,options,q,u);
  ```

- These variables may be of any type (vectors, matrix, struct). Especially a struct is useful to carry many values in 1 variable.

Solving systems of ODEs in Matlab: example

You have noticed that the step size in t varied. This is because we have given just the begin and end times of our time span:

```
tspan = [0 10];
```

Solving systems of ODEs in Matlab: example

You have noticed that the step size in t varied. This is because we have given just the begin and end times of our time span:

```
tspan = [0 10];
```

You can also solve at specific steps, by supplying all steps explicitly, e.g.:

```
tspan = linspace(0,10,101);
```

This example provides 101 explicit time steps between 0 and 10 seconds.

Note that the results are interpolated to these data points afterwards; you do not influence the efficiency and accuracy of the solver algorithm this way!

Ordinary differential equations 2

Implicit methods, systems of ODEs and boundary value problems

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
  - Backward Euler
  - Implicit midpoint method
- Systems of ODEs
  - Solution methods for systems of ODEs
  - Solving systems of ODEs in Matlab
  - Stiff systems of ODEs
- Boundary value problems
  - Shooting method
- Conclusion

Today's outline

- Introduction
  - Backward Euler
  - Implicit midpoint method
- Systems of ODEs
  - Solution methods for systems of ODEs
  - Solving systems of ODEs in Matlab
  - Stiff systems of ODEs
- Boundary value problems
  - Shooting method
- Conclusion

Problems with Euler's method: instability

Consider the ODE:

$$\frac{dy}{dx} = f(x, y(x)) \qquad \text{with} \qquad y(x=0) = y_0$$

Problems with Euler's method: instability

Consider the ODE:

$$\frac{dy}{dx} = f(x, y(x)) \qquad \text{with} \qquad y(x=0) = y_0$$

First order approximation of derivative: $\frac{dy}{dx} = \frac{y_{i+1} - y_i}{\Delta x}$.

Where to evaluate the function f?

Problems with Euler's method: instability

Consider the ODE:

$$\frac{dy}{dx} = f(x, y(x)) \qquad \text{with} \qquad y(x=0) = y_0$$

First order approximation of derivative: $\frac{dy}{dx} = \frac{y_{i+1} - y_i}{\Delta x}$.

Where to evaluate the function f?

1. Evaluation at x_i: Explicit Euler method (forward Euler)
2. Evaluation at x_{i+1}: Implicit Euler method (backward Euler)

Problems with Euler's method: instability – forward Euler

Explicit Euler method (forward Euler):

- Use values at x_i:
  $\frac{y_{i+1}-y_i}{\Delta x} = f(x_i, y_i) \Rightarrow y_{i+1} = y_i + hf(x_i, y_i)$.
- This is an explicit equation for y_{i+1} in terms of y_i.
- It can give instabilities with large function values.

Problems with Euler's method: instability – forward Euler

Explicit Euler method (forward Euler):

- Use values at x_i:
  $\frac{y_{i+1}-y_i}{\Delta x} = f(x_i, y_i) \Rightarrow y_{i+1} = y_i + hf(x_i, y_i)$.
- This is an explicit equation for y_{i+1} in terms of y_i.
- It can give instabilities with large function values.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - k c_i \Delta t \Rightarrow \frac{c_{i+1}}{c_i} = 1 - k\Delta t$$

Problems with Euler's method: instability – forward Euler

Explicit Euler method (forward Euler):

- Use values at x_i:
  $\frac{y_{i+1}-y_i}{\Delta x} = f(x_i, y_i) \Rightarrow y_{i+1} = y_i + hf(x_i, y_i)$.
- This is an explicit equation for y_{i+1} in terms of y_i.
- It can give instabilities with large function values.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - kc_i\Delta t \Rightarrow \frac{c_{i+1}}{c_i} = 1 - k\Delta t$$

It follows that unphysical results are obtained for $k\Delta t \geq 1$!!

Stability requirement

$$k\Delta t < 1$$

(but probably accuracy requirements are more stringent here!)

Problems with Euler's method: instability – backward Euler

Implicit Euler method (backward Euler):

- Use values at x_{i+1}: $\frac{y_{i+1}-y_i}{\Delta x} = f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$.
- This is an implicit equation for y_{i+1}, because it also depends on terms of y_{i+1}.

Problems with Euler's method: instability – backward Euler

Implicit Euler method (backward Euler):

- Use values at x_{i+1}: $\frac{y_{i+1}-y_i}{\Delta x} = f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$.
- This is an implicit equation for y_{i+1}, because it also depends on terms of y_{i+1}.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - k c_{i+1} \Delta t \Rightarrow \frac{c_{i+1}}{c_i} = \frac{1}{1 + k\Delta t}$$

Problems with Euler's method: instability – backward Euler

Implicit Euler method (backward Euler):

- Use values at x_{i+1}: $\frac{y_{i+1}-y_i}{\Delta x} = f(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + hf(x_{i+1}, y_{i+1})$.
- This is an implicit equation for y_{i+1}, because it also depends on terms of y_{i+1}.

Consider the first order batch reactor:

$$\frac{dc}{dt} = -kc \Rightarrow c_{i+1} = c_i - k c_{i+1} \Delta t \Rightarrow \frac{c_{i+1}}{c_i} = \frac{1}{1 + k\Delta t}$$

This equation does never give unphysical results!
The implicit Euler method is *unconditionally stable*
(but maybe not very accurate or efficient).

Semi-implicit Euler method

Usually f is a non-linear function of y, so that linearization is required (recall Newton's method).

$$\frac{dy}{dx} = f(y) \Rightarrow y_{i+1} = y_i + hf(y_{i+1}) \quad \text{using} \quad f(y_{i+1}) = f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i) + \ldots$$

$$\Rightarrow y_{i+1} = y_i + h\left[f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)\right]$$

$$\Rightarrow \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_{i+1} = \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_i + hf(y_i)$$

$$\Rightarrow \quad y_{i+1} \quad = \quad y_i \quad + \quad h\left(1 - h\left.\frac{df}{dy}\right|_i\right)^{-1} f(y_i)$$

Semi-implicit Euler method

Usually f is a non-linear function of y, so that linearization is required (recall Newton's method).

$$\frac{dy}{dx} = f(y) \Rightarrow y_{i+1} = y_i + hf(y_{i+1}) \quad \text{using} \quad f(y_{i+1}) = f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i) + \ldots$$

$$\Rightarrow y_{i+1} = y_i + h\left[f(y_i) + \left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)\right]$$

$$\Rightarrow \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_{i+1} = \left(1 - h\left.\frac{df}{dy}\right|_i\right) y_i + hf(y_i)$$

$$\Rightarrow \quad y_{i+1} \quad = \quad y_i \quad + \quad h\left(1 - h\left.\frac{df}{dy}\right|_i\right)^{-1} f(y_i)$$

For the case that $f(x, y(x))$ we could add the variable x as an additional variable $y_{n+1} = x$. Or add one fully implicit Euler step (which avoids the computation of $\frac{\partial f}{\partial x}$):

$$y_{i+1} = y_i + hf(x_{i+1}, y_{i+1}) \Rightarrow y_{i+1} = y_i + h\left(1 - h\left.\frac{df}{dy}\right|_i\right)^{-1} f(x_{i+1}, y_i)$$

Semi-implicit Euler method - example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

Semi-implicit Euler method - example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc \Rightarrow c_{i+1} = c_i - \frac{hkc_i^2}{1+2hkc_i}$.

Semi-implicit Euler method - example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc \Rightarrow c_{i+1} = c_i - \frac{hkc_i^2}{1+2hkc_i}$.

N	ζ	$\frac{\zeta_{\mathrm{numerical}} - \zeta_{\mathrm{analytical}}}{\zeta_{\mathrm{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.654066262	1.89×10^{-2}	—
40	0.660462687	9.31×10^{-3}	1.02220
80	0.663589561	4.62×10^{-3}	1.01162
160	0.665134433	2.30×10^{-3}	1.00594
320	0.665902142	1.15×10^{-3}	1.00300

Second order implicit method: Implicit midpoint method

Implicit midpoint rule (second order)	Explicit midpoint rule (modified Euler method)
$y_{i+1} = y_i + hf\left(x_i + \frac{1}{2}h, \frac{1}{2}(y_i + y_{i+1})\right)$	$y_{i+1} = y_i + hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$

in case $f(y)$ then:

$$f\left(\frac{1}{2}(y_i + y_{i+1})\right) = f_i + \left.\frac{df}{dy}\right|_i \left(\frac{1}{2}(y_i + y_{i+1}) - y_i\right) = f_i + \frac{1}{2}\left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)$$

Second order implicit method: Implicit midpoint method

Implicit midpoint rule (second order)	Explicit midpoint rule (modified Euler method)
$y_{i+1} = y_i + hf\left(x_i + \frac{1}{2}h, \frac{1}{2}(y_i + y_{i+1})\right)$	$y_{i+1} = y_i + hf(x_i + \frac{1}{2}h, y_i + \frac{1}{2}hk_1)$

in case $f(y)$ then:

$$f\left(\frac{1}{2}(y_i + y_{i+1})\right) = f_i + \left.\frac{df}{dy}\right|_i \left(\frac{1}{2}(y_i + y_{i+1}) - y_i\right) = f_i + \frac{1}{2}\left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)$$

Implicit midpoint rule reduces to:

$$y_{i+1} = y_i + hf_i + \frac{h}{2}\left.\frac{df}{dy}\right|_i (y_{i+1} - y_i)$$

$$\Rightarrow \left(1 - \frac{h}{2}\left.\frac{df}{dy}\right|_i\right) y_{i+1} = \left(1 - \frac{h}{2}\left.\frac{df}{dy}\right|_i\right) y_i + hf_i$$

$$\Rightarrow \quad y_{i+1} \quad = \quad y_i \quad + \quad h\left(1 - \frac{h}{2}\left.\frac{df}{dy}\right|_i\right)^{-1} f_i$$

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2$ s (Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$).

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$ (Analytical solution: $c(t) = \frac{c_0}{1+kc_0t}$).

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc$.

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2$ s (Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$).

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc$.

Substitution:

$$c_{i+1} = c_i + h\left(1 - \frac{h}{2}\cdot(-2kc_i)\right)^{-1}\cdot(-kc_i^2)$$
$$= c_i - \frac{hkc_i^2}{1+hkc_i} = \frac{c_i + hkc_i^2 - hkc_i^2}{1+hkc_i} \Rightarrow c_{i+1} = \frac{c_i}{1+hkc_i}$$

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2$ s (Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$).

Define $f = -kc^2$, then $\frac{df}{dc} = -2kc$.

Substitution:

$$c_{i+1} = c_i + h\left(1 - \frac{h}{2}\cdot(-2kc_i)\right)^{-1}\cdot(-kc_i^2)$$
$$= c_i - \frac{hkc_i^2}{1+hkc_i} = \frac{c_i + hkc_i^2 - hkc_i^2}{1+hkc_i} \Rightarrow c_{i+1} = \frac{c_i}{1+hkc_i}$$

You will find that this method is exact for all step sizes h because of the quadratic source term!

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

$$c_{i+1} = \frac{c_i}{1 + hkc_i}$$

Implicit midpoint method — example

Second order reaction in a batch reactor:
$\frac{dc}{dt} = -kc^2$ with $c_0 = 1\ \mathrm{mol\,m^{-3}}$, $k = 1\ \mathrm{m^3\,mol^{-1}\,s^{-1}}$, $t_{\mathrm{end}} = 2\ \mathrm{s}$
Analytical solution: $c(t) = \frac{c_0}{1+kc_0 t}$

$$c_{i+1} = \frac{c_i}{1 + hkc_i}$$

N	ζ	$\frac{\zeta_{\mathrm{numerical}} - \zeta_{\mathrm{analytical}}}{\zeta_{\mathrm{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.6666666667	1.665×10^{-16}	—
40	0.6666666667	0	—
80	0.6666666667	0	—
160	0.6666666667	0	—
320	0.6666666667	0	—

Implicit midpoint method — example

Third order reaction in a batch reactor: $\frac{dc}{dt} = -kc^3$
Analytical solution: $c(t) = \frac{c_0}{\sqrt{1+2kc_0^2 t}}$

$$c_{i+1} = c_i - \frac{hkc_i^3}{1 + \frac{3}{2}hkc_i^2}$$

Implicit midpoint method — example

Third order reaction in a batch reactor: $\frac{dc}{dt} = -kc^3$
Analytical solution: $c(t) = \frac{c_0}{\sqrt{1+2kc_0^2 t}}$

$$c_{i+1} = c_i - \frac{hkc_i^3}{1 + \frac{3}{2}hkc_i^2}$$

N	ζ	$\frac{\zeta_{\text{numerical}} - \zeta_{\text{analytical}}}{\zeta_{\text{analytical}}}$	$r = \frac{\log\left(\frac{\epsilon_i}{\epsilon_{i-1}}\right)}{\log\left(\frac{N_{i-1}}{N_i}\right)}$
20	0.5526916174	1.71×10^{-4}	—
40	0.5527633731	4.17×10^{-5}	2.041
80	0.5527807304	1.03×10^{-5}	2.021
160	0.5527849965	2.55×10^{-6}	2.011
320	0.5527860538	6.34×10^{-7}	2.005

Today's outline

- Introduction
  - Backward Euler
  - Implicit midpoint method
- Systems of ODEs
  - Solution methods for systems of ODEs
  - Solving systems of ODEs in Matlab
  - Stiff systems of ODEs
- Boundary value problems
  - Shooting method
- Conclusion

Systems of ODEs

A system of ODEs is specified using vector notation:

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x, \boldsymbol{y}(x))$$

for

$$\frac{dy_1}{dx} = f_1(x, y_1(x), y_2(x)) \quad \text{or} \quad f_1(x, y_1, y_2)$$
$$\frac{dy_2}{dx} = f_2(x, y_1(x), y_2(x)) \quad \text{or} \quad f_2(x, y_1, y_2)$$

Systems of ODEs

A system of ODEs is specified using vector notation:

$$\frac{d\boldsymbol{y}}{dx} = \boldsymbol{f}(x, \boldsymbol{y}(x))$$

for

$$\frac{dy_1}{dx} = f_1(x, y_1(x), y_2(x)) \quad \text{or} \quad f_1(x, y_1, y_2)$$
$$\frac{dy_2}{dx} = f_2(x, y_1(x), y_2(x)) \quad \text{or} \quad f_2(x, y_1, y_2)$$

The solution techniques discussed before can also be used to solve systems of equations.

Systems of ODEs: Explicit methods

Forward Euler method

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{f}(x_i, \boldsymbol{y}_i)$$

Improved Euler method (classical RK2)

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + \frac{h}{2}(\boldsymbol{k}_1 + \boldsymbol{k}_2) \quad \text{using} \quad \begin{aligned} \boldsymbol{k}_1 &= \boldsymbol{f}(x_i, \boldsymbol{y}_i) \\ \boldsymbol{k}_2 &= \boldsymbol{f}(x_i + h, \boldsymbol{y}_i + h\boldsymbol{k}_1) \end{aligned}$$

Modified Euler method (midpoint rule)

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\boldsymbol{k}_2 \quad \text{using} \quad \begin{aligned} \boldsymbol{k}_1 &= \boldsymbol{f}(x_i, \boldsymbol{y}_i) \\ \boldsymbol{k}_2 &= \boldsymbol{f}(x_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_1) \end{aligned}$$

Systems of ODEs: Explicit methods

Classical fourth order Runge-Kutta method (RK4)

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\left(\frac{\boldsymbol{k}_1}{6} + \frac{1}{3}(\boldsymbol{k}_2 + \boldsymbol{k}_3) + \frac{\boldsymbol{k}_4}{6}\right)$$

using

$$\boldsymbol{k}_1 = \boldsymbol{f}(x_i, \boldsymbol{y}_i)$$

$$\boldsymbol{k}_2 = \boldsymbol{f}(x_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_1)$$

$$\boldsymbol{k}_3 = \boldsymbol{f}(x_i + \frac{h}{2}, \boldsymbol{y}_i + \frac{h}{2}\boldsymbol{k}_2)$$

$$\boldsymbol{k}_4 = \boldsymbol{f}(x_i + h, \boldsymbol{y}_i + h\boldsymbol{k}_3)$$

Solving systems of ODEs in Matlab

Solving systems of ODEs in Matlab is completely analogous to solving a single ODE:

1. Create a function that specifies the ODEs. This function returns the $\frac{dy}{dx}$ vector.
2. Initialise solver variables and settings (e.g. step size, initial conditions, tolerance), in a separate script. Initial conditions and tolerances should be given per-equation, i.e. as a vector.
3. Call the ODE solver function, using a *function handle* to the ODE function described in point 1.
   - The ODE solver will return the vector for the independent variable, and a solution matrix, with a column as the solution for each equation in the system.

Solving systems of ODEs in Matlab: example

We solve the system: $\dfrac{dx_1}{dt} = -x_1 - x_2, \quad \dfrac{dx_2}{dt} = x_1 - 2x_2$

Create an ODE function

```
function [dxdt] = myODEFunction(t,x)
dxdt(1) = -x(1) - x(2);
dxdt(2) =  x(1) - 2*x(2);
dxdt=dxdt'; % Transpose to column vector
return
```

Solving systems of ODEs in Matlab: example

We solve the system: $\dfrac{dx_1}{dt} = -x_1 - x_2, \quad \dfrac{dx_2}{dt} = x_1 - 2x_2$

Create an ODE function

```
function [dxdt] = myODEFunction(t,x)
dxdt(1) = -x(1) - x(2);
dxdt(2) =  x(1) - 2*x(2);
dxdt=dxdt'; % Transpose to column vector
return
```

Create a solution script

```
x_init = [0 1];          % Initial conditions
tspan = [0 10];          % Time span
options = odeset('RelTol',1e-4,'AbsTol',[1e-4 1e-4]);
[t,x] = ode45(@myODEfunction,tspan,x_init,options);
```

Solving systems of ODEs in Matlab: example

Plot the solution:

```
plot(t,x(:,1),'r-x',t,x(:,2),'b-o')
```

Solving systems of ODEs in Matlab: example

Plot the solution:

```
plot(t,x(:,1),'r-x',t,x(:,2),'b-o')
```

Solving systems of ODEs in Matlab: repeated notes

A few notes on working with `ode45` and other solvers. If we want to give additional arguments (e.g. `a`, `b` and `c`) to our ODE function, we can list them in the function line:

```
function [dxdt] = myODE(t,x,a,b,c)
```

The additional arguments can now be set in the solver script by *adding them after the options*:

```
[t,x] = ode45(@myODE,tspan,x_0,options,a,b,c);
```

Solving systems of ODEs in Matlab: repeated notes

A few notes on working with `ode45` and other solvers. If we want to give additional arguments (e.g. `a`, `b` and `c`) to our ODE function, we can list them in the function line:

```
function [dxdt] = myODE(t,x,a,b,c)
```

The additional arguments can now be set in the solver script by *adding them after the options*:

```
[t,x] = ode45(@myODE,tspan,x_0,options,a,b,c);
```

- Of course, in the solver script, the variables do not need to be called `a`, `b` and `c`:

  ```
  [t,x] = ode45(@myODE,tspan,x_0,options,k1,phi,V);
  ```

- These variables may be of any type (vectors, matrix, struct). Especially a struct is useful to carry many values in 1 variable.

Solving systems of ODEs in Matlab: example

You may have noticed that the step size in t varied. This is because we have given the begin and end times of our time span, and `ode45` uses adaptive step size for efficiency:

```
tspan = [0 10];
```

Solving systems of ODEs in Matlab: example

You may have noticed that the step size in t varied. This is because we have given the begin and end times of our time span, and `ode45` uses adaptive step size for efficiency:

```
tspan = [0 10];
```

You can also retrieve the solution at specific steps, by supplying all steps explicitly as a vector, e.g.:

```
tspan = linspace(0,10,101);
```

This example provides 101 explicit time steps between 0 and 10 seconds.

Note that this is an interpolated result. The solver uses, in the background, still the adaptive step size functionality!

Systems of ODEs: Implicit methods

Backward Euler method

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\left(\boldsymbol{I} - h\left.\frac{d\boldsymbol{f}}{d\boldsymbol{y}}\right|_i\right)^{-1}\boldsymbol{f}(\boldsymbol{y}_i)$$

Implicit midpoint method

$$\boldsymbol{y}_{i+1} = \boldsymbol{y}_i + h\left(\boldsymbol{I} - \frac{h}{2}\left.\frac{d\boldsymbol{f}}{d\boldsymbol{y}}\right|_i\right)^{-1}\boldsymbol{f}(\boldsymbol{y}_i)$$

Stiff systems of ODEs

A system of ODEs can be stiff and require a different solution method.

Stiff systems of ODEs

A system of ODEs can be stiff and require a different solution method. For example:

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \qquad \frac{dc_2}{dt} = -999c_1 - 1999c_2$$

with boundary conditions $c_1(t=0) = 1$ and $c_2(t=0) = 0$.
The analytical solution is:

$$c_1 = 2e^{-t} - e^{-1000t} \qquad c_2 = -e^{-t} + e^{-1000t}$$

For the explicit method we require $\Delta t < 10^{-3}$ despite the fact that the term is completely negligible, but essential to keep stability.

Stiff systems of ODEs

A system of ODEs can be stiff and require a different solution method. For example:

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \qquad \frac{dc_2}{dt} = -999c_1 - 1999c_2$$

with boundary conditions $c_1(t=0) = 1$ and $c_2(t=0) = 0$.
The analytical solution is:

$$c_1 = 2e^{-t} - e^{-1000t} \qquad c_2 = -e^{-t} + e^{-1000t}$$

For the explicit method we require $\Delta t < 10^{-3}$ despite the fact that the term is completely negligible, but essential to keep stability.

The "disease" of stiff equations: we need to follow the solution on the shortest length scale to maintain stability of the integration, although accuracy requirements would allow a much larger time step.

Demonstration with example

Forward Euler (explicit)

$$\frac{c_{1,i+1} - c_{1,i}}{dt} = 998c_{1,i} + 1998c_{2,i}$$
$$\frac{c_{2,i+1} - c_{2,i}}{dt} = -999c_{1,i} - 1999c_{2,i}$$

$$\Rightarrow \begin{aligned} c_{1,i+1} &= (1 + 998\Delta t)\, c_{1,i} + 1998\Delta t c_{2,i} \\ c_{2,i+1} &= -999\Delta t c_{1,i} + (1 - 1999\Delta t)\, c_{2,i} \end{aligned}$$

Demonstration with example

Backward Euler (implicit)

$$\frac{dc_{1,i+1} - c_{1,i+1}}{dt} = 998c_{1,i+1} + 1998c_{2,i+1}$$
$$\frac{dc_{2,i+1} - c_{2,i+1}}{dt} = -999c_{1,i+1} - 1999c_{2,i+1}$$

$$\Rightarrow \begin{array}{l} (1 - 998\Delta t)\, c_{1,i+1} - 1998\Delta t c_{2,i} = c_{1,i} \\ 999\Delta t c_{1,i+1} + (1 + 999\Delta t)\, c_{2,i+1} = c_{2,i} \end{array}$$

Demonstration with example

Backward Euler (implicit)

$$\frac{dc_{1,i+1} - c_{1,i+1}}{dt} = 998c_{1,i+1} + 1998c_{2,i+1}$$
$$\frac{dc_{2,i+1} - c_{2,i+1}}{dt} = -999c_{1,i+1} - 1999c_{2,i+1}$$

$$\Rightarrow \begin{array}{l} (1 - 998\Delta t)\, c_{1,i+1} - 1998\Delta t c_{2,i} = c_{1,i} \\ 999\Delta t c_{1,i+1} + (1 + 999\Delta t)\, c_{2,i+1} = c_{2,i} \end{array}$$

$$A\boldsymbol{c}_{i+1} = \boldsymbol{c}_i \text{ with } A = \begin{pmatrix} 1 - 998\Delta t & -1998\Delta t \\ 999\Delta t & 1 + 1999\Delta t \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$$

Demonstration with example

Backward Euler (implicit) $A\boldsymbol{c}_{i+1} = \boldsymbol{c}_i$ with $A = \begin{pmatrix} 1-998\Delta t & -1998\Delta t \\ 999\Delta t & 1+1999\Delta t \end{pmatrix}$ and $\boldsymbol{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$

Demonstration with example

Backward Euler (implicit) $A\boldsymbol{c}_{i+1} = \boldsymbol{c}_i$ with $A = \begin{pmatrix} 1-998\Delta t & -1998\Delta t \\ 999\Delta t & 1+1999\Delta t \end{pmatrix}$ and $\boldsymbol{b} = \begin{pmatrix} c_{1,i} \\ c_{2,i} \end{pmatrix}$

Cramers rule:

$$c_{1,i+1} = \frac{\begin{vmatrix} c_{1,i} & -1998\Delta t \\ c_{2,i} & 1+1999\Delta t \end{vmatrix}}{\det|A|} = \frac{(1+1999\Delta t)c_{1,i}+1998\Delta t c_{2,i}}{(1-998\Delta t)(1+1999\Delta t)+1998\cdot 999\Delta t^2}$$

$$c_{2,i+1} = \frac{\begin{vmatrix} 1-998\Delta t & c_{1,i} \\ 999\Delta t & c_{2,i} \end{vmatrix}}{\det|A|} = \frac{-999\Delta t c_{1,i}+(1-998\Delta t)c_{2,i}}{(1-998\Delta t)(1+1999\Delta t)+1998\cdot 999\Delta t^2}$$

Forward Euler: $\Delta t \leq 0.001$ for stability

Backward Euler: always stable, even for $\Delta t > 100$ (but then not very accurate!)

Demonstration with example

Cure for stiff problems: use implicit methods! To find out whether your system is stiff: check whether one of the eigenvalues have an imaginary part

Implicit methods in Matlab

Matlab offers a stabilized solver, `ode15s`, for stiff problems.

$$\frac{dc_1}{dt} = 998c_1 + 1998c_2 \quad \frac{dc_2}{dt} = -999c_1 - 1999c_2,\ c_1(0) = 1,\ c_2(0) = 0$$

- Create the ode function

```
function [dcdt] = stiff_ode(t,c)
dcdt = zeros(2,1); % Pre-allocation
dcdt(1) =  998 * c(1) + 1998*c(2);
dcdt(2) = -999 * c(1) - 1999*c(2);
return
```

- Compare the resolution of the solutions

```
subplot(2,1,1);
ode45(@stiff_ode, [0 1], [1 0]);
subplot(2,1,2);
ode15s(@stiff_ode, [0 1], [1 0]);
```

Implicit methods in Matlab

The explicit solver requires 1245 data points (default settings), the implicit solver requires just 48!

Today's outline

- Introduction
  - Backward Euler
  - Implicit midpoint method
- Systems of ODEs
  - Solution methods for systems of ODEs
  - Solving systems of ODEs in Matlab
  - Stiff systems of ODEs
- Boundary value problems
  - Shooting method
- Conclusion

Importance of boundary conditions

The nature of boundary conditions determines the appropriate numerical method. Classification into 2 main categories:

- *Initial value problems (IVP)*
  We know the values of all y_i at some starting position x_s, and it is desired to find the values of y_i at some final point x_f.

- *Boundary value problems (BVP)*
  Boundary conditions are specified at more than one x. Typically, some of the BC are specified at x_s and the remainder at x_f.

Shooting method

How to solve a BVP using the shooting method:

- Define the system of ODEs
- Provide an initial guess for the unknown boundary condition
- Solve the system and compare the resulting boundary condition to the expected value
- Adjust the guessed boundary value, and solve again. Repeat until convergence.
  - Of course, you can subtract the expected value from the computed value at the boundary, and use a non-linear root finding method

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ:

$$\mathcal{D}\frac{d^2c}{dx^2} = k_R c \text{ with } \quad \begin{aligned} c(x=0) &= C_{A,i,L} = 1 && \text{(interface concentration)} \\ c(x=\delta) &= 0 && \text{(bulk concentration)} \end{aligned}$$

Question:

compute the concentration profile in the film layer.

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ:

$$\mathcal{D}\frac{d^2c}{dx^2} = k_R c \text{ with } \quad \begin{aligned} c(x=0) &= C_{A,i,L} = 1 \quad &\text{(interface concentration)} \\ c(x=\delta) &= 0 \quad &\text{(bulk concentration)} \end{aligned}$$

Question:

compute the concentration profile in the film layer.

Step 1: Define the system of ODEs

This second-order ODE can be rewritten as a system of first-order ODEs, if we define the flux q as:

$$q = -\mathcal{D}\frac{dc}{dx}$$

Now, we find:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}q$$
$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Consider a chemical reaction in a liquid film layer of thickness δ:

$$\mathcal{D}\frac{d^2c}{dx^2} = k_R c \text{ with } \quad \begin{aligned} c(x=0) &= C_{A,i,L} = 1 \quad &&\text{(interface concentration)} \\ c(x=\delta) &= 0 &&\text{(bulk concentration)} \end{aligned}$$

Question:

compute the concentration profile in the film layer.

Step 2: Set the boundary conditions

The boundary conditions for the concentrations at $x = 0$ and $x = \delta$ are known.

The flux at the interface, however, is not known, and should be solved for.

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}}q$$

$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Solving the two first-order ODEs in Excel. First, the cells with constants:

	A	B	C
1	CAiL	1	mol/m3
2	D	1e–8	m2/s
3	kR	10	1/s
4	delta	1e–4	m
5	N	100	
6	dx	=B4/B5	

$$\frac{dc}{dx} = -\frac{1}{D}q$$

$$\frac{dq}{dx} = -k_R c$$

BVP: example in Excel

Solving the two first-order ODEs in Excel. First, the cells with constants:

	A	B	C
1	CAiL	1	mol/m3
2	D	1e–8	m2/s
3	kR	10	1/s
4	delta	1e–4	m
5	N	100	
6	dx	=B4/B5	

$$\frac{dc}{dx} = -\frac{1}{D}q$$

$$\frac{dq}{dx} = -k_R c$$

Now, we program the forward Euler (explicit) schemes for c and q below:

	A	B	C
10	x	c	q
11	0	=B1	10
12	=A11+$B$6	=B11+$B$6*(–1/$B$2*C11)	=C11+$B$6*(-$B$3*B11)
13	=A12+$B$6	=B12+$B$6*(–1/$B$2*C12)	=C12+$B$6*(-$B$3*B12)
...	...	...	...
111	=A110+$B$6	=B110+$B$6*(–1/$B$2*C110)	=C110+$B$6*(-$B$3*B110)

BVP: example in Excel

- We now have profiles for c and q as a function of position x.
- The concentration $c(x = \delta)$ depends (eventually) on the boundary condition at the interface $q(x = 0)$
- We can use the solver to change $q(x = 0)$ such that the concentration at the bulk meets our requirement: $c(x = \delta) = 0$

BVP: example in Matlab

We first program the system of ODEs in a separate function:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}} q$$
$$\frac{dq}{dx} = -k_R c$$

```
function [dxdt] = BVPODE(t,x,ps)
dxdt(1)=-1/ps.D*x(2);
dxdt(2)=-ps.kR*x(1);
dxdt=dxdt';
return
```

BVP: example in Matlab

We first program the system of ODEs in a separate function:

$$\frac{dc}{dx} = -\frac{1}{\mathcal{D}} q$$
$$\frac{dq}{dx} = -k_R c$$

```
function [dxdt] = BVPODE(t,x,ps)
dxdt(1)=-1/ps.D*x(2);
dxdt(2)=-ps.kR*x(1);
dxdt=dxdt';
return
```

Note that we pass a variable (type: struct) that contains required parameters: `ps`.

BVP: example in Matlab

The ODE function is solved via ode45, after setting a number of initial and boundary conditions:

```
function f = RunBVP(bcq,ps)
[x,cq] = ode45(@BVPODE,[0 ps.delta],[1 bcq], [], ps);
f = cq(end,1) - 0;
plotyy(x,cq(:,1),x,cq(:,2));
return;
```

BVP: example in Matlab

The ODE function is solved via ode45, after setting a number of initial and boundary conditions:

```
function f = RunBVP(bcq,ps)
[x,cq] = ode45(@BVPODE,[0 ps.delta],[1 bcq], [], ps);
f = cq(end,1) - 0;
plotyy(x,cq(:,1),x,cq(:,2));
return;
```

Note the following:

- We use the interval $0 \leq x \leq \delta$
- Boundary conditions are given as: $c(x=0)=1$ and $q(x=0) = \texttt{bcq}$, which is given as an argument to the function (i.e. changable from 'outside'!)
- The function returns `f`, the difference between the computed and desired concentration at $x=\delta$.

BVP: example in Matlab

Finally, we should solve the system so that we obtain the right boundary condition $q = \texttt{bcq}$ such that $c(x = \delta) = 0$. We can use the built-in function `fzero` to do this

```
% Parameter definition
ps.D=1e-8;
ps.kR=10;
ps.delta=1e-4;

% Solve for flux boundary condition (initial guess: 0)
opt = optimset('Display','iter');
flux = fzero(@RunBVP,0,opt,ps);
```

BVP example: analytical solution

Compare with the analytical solution:

$$q = k_L E_A C_{A,i,L} \quad \text{with}$$

$$E_A = \frac{\mathrm{Ha}}{\tanh \mathrm{Ha}} \qquad \text{(Enhancement factor)}$$

$$\mathrm{Ha} = \frac{\sqrt{k_R \mathcal{D}}}{k_L} \qquad \text{(Hatta number)}$$

$$k_L = \frac{\mathcal{D}}{\delta} \qquad \text{(mass transfer coefficient)}$$

Today's outline

- Introduction
  - Backward Euler
  - Implicit midpoint method
- Systems of ODEs
  - Solution methods for systems of ODEs
  - Solving systems of ODEs in Matlab
  - Stiff systems of ODEs
- Boundary value problems
  - Shooting method
- Conclusion

Other methods

Other explicit methods:

- Burlisch-Stoer method (Richardson extrapolation + modified midpoint method)

Other implicit methods:

- Rosenbrock methods (higher order implicit Runge-Kutta methods)
- Predictor-corrector methods

Summary

- Several solution methods and their derivation were discussed:
  - Explicit solution methods: Euler, Improved Euler, Midpoint method, RK45
  - Implicit methods: Implicit Euler and Implicit midpoint method
  - A few examples of their spreadsheet implementation were shown
- We have paid attention to accuracy and instability, rate of convergence and step size
- Systems of ODEs can be solved by the same algorithms. Stiff problems should be treated with care.
- An example of solving ODEs with Matlab was demonstrated.

Partial differential equations

Dr.ir. Ivo Roghair, Prof.dr.ir. Martin van Sint Annaland

Chemical Process Intensification group
Eindhoven University of Technology

Numerical Methods (6E5X0), 2019/2020

Today's outline

- Introduction
- Instationary diffusion equation
  - Discretization
  - Solving the diffusion equation
  - Non-linear source terms
- Convection
  - Discretization
  - Central difference scheme
  - Upwind scheme
- Conclusions
  - Other methods
  - Summary

Today's outline

- Introduction
- Instationary diffusion equation
  - Discretization
  - Solving the diffusion equation
  - Non-linear source terms
- Convection
  - Discretization
  - Central difference scheme
  - Upwind scheme
- Conclusions
  - Other methods
  - Summary

Overview

Main question

How to solve parabolic PDEs like:

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} - u\frac{\partial c}{\partial x} + R$$

with

$$\begin{aligned} t = 0; 0 \leq x \leq \ell \quad &\Rightarrow c = c_0 \\ t > 0; x = 0 \quad &\Rightarrow -\mathcal{D}\frac{\partial c}{\partial x} + uc = u_{\text{in}} c_{\text{in}} \\ t > 0; x = \ell \quad &\Rightarrow \frac{\partial c}{\partial x} = 0 \end{aligned}$$

accurately and efficiently?

What is a PDE?

Partial differential equation

An equation containing a function and their derivatives to multiple independent variables.

Order of PDE

The highest derivative appearing in the PDE

What is a PDE?

Partial differential equation

An equation containing a function and their derivatives to multiple independent variables.

Order of PDE

The highest derivative appearing in the PDE

General second order PDE:

$$A\frac{\partial^2 f}{\partial x^2} + B\frac{\partial^2 f}{\partial x \partial y} + C\frac{\partial^2 f}{\partial y^2} + D\frac{\partial f}{\partial x} + E\frac{\partial f}{\partial y} + Ff = G$$

- Linear equation: Coefficients $A, B, \ldots, G$ do not depend on x and y.
- Non-linear equation: Coefficients $A, B, \ldots, G$ are a function of x and y.

Classification of PDE's

$$A\frac{\partial^2 f}{\partial x^2} + B\frac{\partial^2 f}{\partial x \partial y} + C\frac{\partial^2 f}{\partial y^2} + D\frac{\partial f}{\partial x} + E\frac{\partial f}{\partial y} + Ff = G$$

The discriminant Δ of a quadratic polynomial is computed as (note: only the higher order coefficients are important):

$\Delta = B^2 - 4AC$

Classification of PDE's

$$A\frac{\partial^2 f}{\partial x^2} + B\frac{\partial^2 f}{\partial x \partial y} + C\frac{\partial^2 f}{\partial y^2} + D\frac{\partial f}{\partial x} + E\frac{\partial f}{\partial y} + Ff = G$$

The discriminant Δ of a quadratic polynomial is computed as (note: only the higher order coefficients are important):

$\Delta = B^2 - 4AC$

- $\Delta < 0 \Rightarrow$ Elliptic equation
  (e.g. Laplace equation for stationary diffusion in 2D)
- $\Delta = 0 \Rightarrow$ Parabolic equation
  (e.g. instationary heat penetration in 1D)
- $\Delta > 0 \Rightarrow$ Hyperbolic equation
  (e.g. wave equation)

Classification of PDE's

$$A\frac{\partial^2 f}{\partial x^2} + B\frac{\partial^2 f}{\partial x \partial y} + C\frac{\partial^2 f}{\partial y^2} + D\frac{\partial f}{\partial x} + E\frac{\partial f}{\partial y} + Ff = G$$

The discriminant Δ of a quadratic polynomial is computed as (note: only the higher order coefficients are important):

$\Delta = B^2 - 4AC$

- $\Delta < 0 \Rightarrow$ Elliptic equation
  (e.g. Laplace equation for stationary diffusion in 2D)
- $\Delta = 0 \Rightarrow$ Parabolic equation
  (e.g. instationary heat penetration in 1D)
- $\Delta > 0 \Rightarrow$ Hyperbolic equation
  (e.g. wave equation)

Classification of PDE's

$$A\frac{\partial^2 f}{\partial x^2} + B\frac{\partial^2 f}{\partial x \partial y} + C\frac{\partial^2 f}{\partial y^2} + D\frac{\partial f}{\partial x} + E\frac{\partial f}{\partial y} + Ff = G$$

The discriminant Δ of a quadratic polynomial is computed as (note: only the higher order coefficients are important):

$\Delta = B^2 - 4AC$

- $\Delta < 0 \Rightarrow$ Elliptic equation
  (e.g. Laplace equation for stationary diffusion in 2D)
- $\Delta = 0 \Rightarrow$ Parabolic equation
  (e.g. instationary heat penetration in 1D)
- $\Delta > 0 \Rightarrow$ Hyperbolic equation
  (e.g. wave equation)

Importance of classification

Different PDE types require different solution techniques because of the difference in range of influence:

- *Characteristics*
  Curves in xy-domain along with signal propagation takes place
- *Domain of dependence of point P*
  points in xy-domain which influence the value of f in point P
- *Range of influence of point P*
  points in xy-domain which are influenced by the value of f in point P

Importance of classification

Different PDE types require different solution techniques because of the difference in range of influence:

- *Characteristics*
  Curves in xy-domain along with signal propagation takes place
- *Domain of dependence of point P*
  points in xy-domain which influence the value of f in point P
- *Range of influence of point P*
  points in xy-domain which are influenced by the value of f in point P

Importance of classification

Different PDE types require different solution techniques because of the difference in range of influence:

- *Characteristics*
  Curves in xy-domain along with signal propagation takes place
- *Domain of dependence of point P*
  points in xy-domain which influence the value of f in point P
- *Range of influence of point P*
  points in xy-domain which are influenced by the value of f in point P

Importance of classification

Different PDE types require different solution techniques because of the difference in range of influence:

- *Characteristics*
  Curves in xy-domain along with signal propagation takes place
- *Domain of dependence of point P*
  points in xy-domain which influence the value of f in point P
- *Range of influence of point P*
  points in xy-domain which are influenced by the value of f in point P

Example elliptic PDE (boundary value problems: BVP)

Grid point at which dependent variable has to be computed

Grid point at which boundary condition is specified

Typical example: Poisson equation

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = f(x,y)$$

Efficiency (memory requirements, CPU time) of the numerical method is of crucial importance.

Example parabolic PDE (initial value problem: IVP)

Grid point at which dependent variable has to be computed

Grid point at which boundary condition is specified

Typical example: Poisson equation

$$\frac{\partial c}{\partial t} + u\frac{\partial c}{\partial x} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} + R$$

Stability (in numerical sense) of the numerical method is of crucial importance.

Boundary conditions

- Dirichlet or fixed condition: prescribed value of f at boundary

  $f = f_0$ $\quad$ f_0 is a known function

- Neumann condition: prescribed value of derivative of f at boundary

  $\frac{\partial f}{\partial n} = q$ $\quad$ q is a known function

- Mixed or Robin condition: relation between f and $\frac{\partial f}{\partial n}$ at boundary

  $a\frac{\partial f}{\partial n} + bf = c$ $\quad$ a, b and c are known functions

Boundary conditions

- Dirichlet or fixed condition: prescribed value of f at boundary

$$f = f_0 \qquad f_0 \text{ is a known function}$$

- Neumann condition: prescribed value of derivative of f at boundary

$$\frac{\partial f}{\partial n} = q \qquad q \text{ is a known function}$$

- Mixed or Robin condition: relation between f and $\frac{\partial f}{\partial n}$ at boundary

$$a\frac{\partial f}{\partial n} + bf = c \qquad a,\ b \text{ and } c \text{ are known functions}$$

Boundary conditions

- Dirichlet or fixed condition: prescribed value of f at boundary

$$f = f_0 \qquad f_0 \text{ is a known function}$$

- Neumann condition: prescribed value of derivative of f at boundary

$$\frac{\partial f}{\partial n} = q \qquad q \text{ is a known function}$$

- Mixed or Robin condition: relation between f and $\frac{\partial f}{\partial n}$ at boundary

$$a\frac{\partial f}{\partial n} + bf = c \qquad a,\ b \text{ and } c \text{ are known functions}$$

Numerical solution method

Finite differences (method of lines, MOL):

1. Discretize spatial domain in discrete grid points
2. Find suitable approximation for the spatial derivatives
3. Substitute approximations in PDE, which gives a system of ODE's, one for every grid points
4. Advance in time with a suitable ODE solver

Alternative methods: collocation, Galerkin or Finite Element methods

Today's outline

- Introduction
- Instationary diffusion equation
  - Discretization
  - Solving the diffusion equation
  - Non-linear source terms
- Convection
  - Discretization
  - Central difference scheme
  - Upwind scheme
- Conclusions
  - Other methods
  - Summary

Instationary diffusion equation (Fick's second law)

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2}, \quad \text{with} \quad \begin{array}{l} t = 0; 0 \leq x \leq \ell \Rightarrow c = c_0 \\ t > 0; x = 0 \Rightarrow c = c_L \\ t > 0; x = \ell \Rightarrow c = c_R \end{array}$$

Second derivative $\frac{\partial^2 c}{\partial x^2}$

Instationary diffusion equation (Fick's second law)

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2}, \quad \text{with} \quad \begin{array}{l} t = 0; 0 \le x \le \ell \Rightarrow c = c_0 \\ t > 0; x = 0 \Rightarrow c = c_L \\ t > 0; x = \ell \Rightarrow c = c_R \end{array}$$

Second derivative $\dfrac{\partial^2 c}{\partial x^2}$

$$c_{i+1} = c_i + \left.\frac{\partial c}{\partial x}\right|_i \Delta x + \frac{1}{2}\left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 + \frac{1}{6}\left.\frac{\partial^3 c}{\partial x^3}\right|_i \Delta x^3 + \ldots$$

$$c_{i-1} = c_i - \left.\frac{\partial c}{\partial x}\right|_i \Delta x + \frac{1}{2}\left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 - \frac{1}{6}\left.\frac{\partial^3 c}{\partial x^3}\right|_i \Delta x^3 + \ldots$$

Instationary diffusion equation (Fick's second law)

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2}, \quad \text{with} \quad \begin{array}{l} t = 0; 0 \le x \le \ell \Rightarrow c = c_0 \\ t > 0; x = 0 \Rightarrow c = c_L \\ t > 0; x = \ell \Rightarrow c = c_R \end{array}$$

Second derivative $\dfrac{\partial^2 c}{\partial x^2}$

$$c_{i+1} = c_i + \left.\frac{\partial c}{\partial x}\right|_i \Delta x + \frac{1}{2}\left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 + \frac{1}{6}\left.\frac{\partial^3 c}{\partial x^3}\right|_i \Delta x^3 + \ldots$$

$$c_{i-1} = c_i - \left.\frac{\partial c}{\partial x}\right|_i \Delta x + \frac{1}{2}\left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 - \frac{1}{6}\left.\frac{\partial^3 c}{\partial x^3}\right|_i \Delta x^3 + \ldots$$

$$c_{i+1} + c_{i-1} = 2c_i + \left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 + \mathcal{O}(\Delta x^4)$$

$$\Rightarrow \left.\frac{\partial^2 c}{\partial x^2}\right|_i = \frac{c_{i+1} - 2c_i + c_{i-1}}{\Delta x^2} + \mathcal{O}(\Delta x^2)$$

Instationary diffusion equation (Fick's second law)

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2}, \quad \text{with} \quad \begin{array}{l} t = 0; 0 \le x \le \ell \Rightarrow c = c_0 \\ t > 0; x = 0 \Rightarrow c = c_L \\ t > 0; x = \ell \Rightarrow c = c_R \end{array}$$

Second derivative $\dfrac{\partial^2 c}{\partial x^2}$

$$c_{i+1} = c_i + \left.\frac{\partial c}{\partial x}\right|_i \Delta x + \frac{1}{2}\left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 + \frac{1}{6}\left.\frac{\partial^3 c}{\partial x^3}\right|_i \Delta x^3 + \ldots$$

$$c_{i-1} = c_i - \left.\frac{\partial c}{\partial x}\right|_i \Delta x + \frac{1}{2}\left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 - \frac{1}{6}\left.\frac{\partial^3 c}{\partial x^3}\right|_i \Delta x^3 + \ldots$$

$$c_{i+1} + c_{i-1} = 2c_i + \left.\frac{\partial^2 c}{\partial x^2}\right|_i \Delta x^2 + \mathcal{O}(\Delta x^4)$$

$$\Rightarrow \left.\frac{\partial^2 c}{\partial x^2}\right|_i = \frac{c_{i+1} - 2c_i + c_{i-1}}{\Delta x^2} + \mathcal{O}(\Delta x^2)$$

Due to symmetric discretization: second order (central discretization).

Instationary diffusion equation (Fick's second law)

An alternative discretization:

$$\left.\frac{\partial^2 c}{\partial x^2}\right|_i = \frac{\left.\frac{\partial c}{\partial x}\right|_{i+\frac{1}{2}} - \left.\frac{\partial c}{\partial x}\right|_{i-\frac{1}{2}}}{\Delta x} + \mathcal{O}(\Delta x^2)$$

Instationary diffusion equation (Fick's second law)

An alternative discretization:

$$\left.\frac{\partial^2 c}{\partial x^2}\right|_i = \frac{\left.\frac{\partial c}{\partial x}\right|_{i+\frac{1}{2}} - \left.\frac{\partial c}{\partial x}\right|_{i-\frac{1}{2}}}{\Delta x} + \mathcal{O}(\Delta x^2)$$

$$= \frac{\frac{c_{i+1} - c_i}{\Delta x} - \frac{c_i - c_{i-1}}{\Delta x}}{\Delta x} = \frac{c_{i+1} - 2c_i + c_{i-1}}{\Delta x^2}$$

Instationary diffusion equation (Fick's second law)

An alternative discretization:

$$\left.\frac{\partial^2 c}{\partial x^2}\right|_i = \frac{\left.\frac{\partial c}{\partial x}\right|_{i+\frac{1}{2}} - \left.\frac{\partial c}{\partial x}\right|_{i-\frac{1}{2}}}{\Delta x} + \mathcal{O}(\Delta x^2)$$

$$= \frac{\frac{c_{i+1}-c_i}{\Delta x} - \frac{c_i - c_{i-1}}{\Delta x}}{\Delta x} = \frac{c_{i+1} - 2c_i + c_{i-1}}{\Delta x^2}$$

This is convenient for the derivation of $\frac{\partial}{\partial x}\left(\mathcal{D}\frac{\partial c}{\partial x}\right)$:

$$\frac{\partial}{\partial x}\left(\mathcal{D}\frac{\partial c}{\partial x}\right) = \frac{\mathcal{D}_{i+\frac{1}{2}}\left.\frac{\partial c}{\partial x}\right|_{i+\frac{1}{2}} - \mathcal{D}_{i-\frac{1}{2}}\left.\frac{\partial c}{\partial x}\right|_{i-\frac{1}{2}}}{\Delta x} = \frac{\mathcal{D}_{i+\frac{1}{2}}\frac{c_{i+1}-c_i}{\Delta x} - \mathcal{D}_{i-\frac{1}{2}}\frac{c_i - c_{i-1}}{\Delta x}}{\Delta x}$$

$$= \frac{\mathcal{D}_{i+\frac{1}{2}}c_{i+1} - \left(\mathcal{D}_{i+\frac{1}{2}} + \mathcal{D}_{i-\frac{1}{2}}\right)c_i + \mathcal{D}_{i-\frac{1}{2}}c_{i-1}}{(\Delta x)^2}$$

Instationary diffusion equation (Fick's second law)

$$\frac{\partial^2 f}{\partial x^2}$$

Instationary diffusion equation (Fick's second law)

$$\frac{\partial^2 f}{\partial x^2}$$

$$f_{i+\frac{1}{2}} = f_i + \frac{1}{2}\Delta x \left.\frac{\partial f}{\partial x}\right|_i \Delta x + \frac{1}{2}\left(\frac{1}{2}\Delta x\right)^2 \left.\frac{\partial^2 f}{\partial x^2}\right|_i + \mathcal{O}(\Delta x^3)$$

$$f_{i-\frac{1}{2}} = f_i - \frac{1}{2}\Delta x \left.\frac{\partial f}{\partial x}\right|_i \Delta x + \frac{1}{2}\left(\frac{1}{2}\Delta x\right)^2 \left.\frac{\partial^2 f}{\partial x^2}\right|_i + \mathcal{O}(\Delta x^3)$$

$$f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}} = \Delta x \frac{\partial f}{\partial x} + \mathcal{O}(\Delta x^3)$$

Instationary diffusion equation (Fick's second law)

$$\frac{\partial^2 f}{\partial x^2}$$

$$f_{i+\frac{1}{2}} = f_i + \frac{1}{2}\Delta x \left.\frac{\partial f}{\partial x}\right|_i \Delta x + \frac{1}{2}\left(\frac{1}{2}\Delta x\right)^2 \left.\frac{\partial^2 f}{\partial x^2}\right|_i + \mathcal{O}(\Delta x^3)$$

$$f_{i-\frac{1}{2}} = f_i - \frac{1}{2}\Delta x \left.\frac{\partial f}{\partial x}\right|_i \Delta x + \frac{1}{2}\left(\frac{1}{2}\Delta x\right)^2 \left.\frac{\partial^2 f}{\partial x^2}\right|_i + \mathcal{O}(\Delta x^3)$$

$$f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}} = \Delta x \frac{\partial f}{\partial x} + \mathcal{O}(\Delta x^3)$$

$$\Rightarrow \left.\frac{\partial f}{\partial x}\right|_i = \frac{f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}}}{\Delta x} + \mathcal{O}(\Delta x^2)$$

Symmetric discretization yields second order!

Instationary diffusion equation: spatial discretization

Substitution of spatial derivatives yields:

$$\frac{dc_i}{dt} = \mathcal{D}\frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2} \quad \text{for } i = 0, \ldots, N$$

For example, using 6 (ridiculously low number!) grid points:

Instationary diffusion equation: spatial discretization

Substitution of spatial derivatives yields:

$$\frac{dc_i}{dt} = \mathcal{D}\frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2} \quad \text{for } i = 0, \ldots, N$$

For example, using 6 (ridiculously low number!) grid points:

Instationary diffusion equation: spatial discretization

Substitution of spatial derivatives yields:

$$\frac{dc_i}{dt} = \mathcal{D}\frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2} \quad \text{for } i = 0, \ldots, N$$

For example, using 6 (ridiculously low number!) grid points:

Instationary diffusion equation: spatial discretization

Substitution of spatial derivatives yields:

$$\frac{dc_i}{dt} = \mathcal{D}\frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2} \quad \text{for } i = 0, \ldots, N$$

For example, using 6 (ridiculously low number!) grid points:

Instationary diffusion equation: spatial discretization

Substitution of spatial derivatives yields:

$$\frac{dc_i}{dt} = \mathcal{D}\frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2} \quad \text{for } i = 0, \ldots, N$$

For example, using 6 (ridiculously low number!) grid points:

Instationary diffusion equation: boundary conditions

Two options:

1. Keep boundary conditions as additional equations:

$$c_0 = c_L, \frac{dc_1}{dt} = \mathcal{D}\frac{c_0 - 2c_1 + c_2}{\Delta x^2}, \frac{dc_2}{dt} = \mathcal{D}\frac{c_1 - 2c_2 + c_3}{\Delta x^2},$$
$$\frac{dc_3}{dt} = \mathcal{D}\frac{c_2 - 2c_3 + c_4}{\Delta x^2}, \frac{dc_4}{dt} = \mathcal{D}\frac{c_3 - 2c_4 + c_5}{\Delta x^2}, c_5 = c_R$$

2. Substitute boundary conditions to reduce number of equations:

$$\frac{dc_1}{dt} = \mathcal{D}\frac{c_L - 2c_1 + c_2}{\Delta x^2}, \frac{dc_2}{dt} = \mathcal{D}\frac{c_1 - 2c_2 + c_3}{\Delta x^2},$$
$$\frac{dc_3}{dt} = \mathcal{D}\frac{c_2 - 2c_3 + c_4}{\Delta x^2}, \frac{dc_4}{dt} = \mathcal{D}\frac{c_3 - 2c_4 + c_R}{\Delta x^2}$$

Instationary diffusion equation: boundary conditions

Two options:

1. Keep boundary conditions as additional equations:

$$c_0 = c_L, \frac{dc_1}{dt} = \mathcal{D}\frac{c_0 - 2c_1 + c_2}{\Delta x^2}, \frac{dc_2}{dt} = \mathcal{D}\frac{c_1 - 2c_2 + c_3}{\Delta x^2},$$
$$\frac{dc_3}{dt} = \mathcal{D}\frac{c_2 - 2c_3 + c_4}{\Delta x^2}, \frac{dc_4}{dt} = \mathcal{D}\frac{c_3 - 2c_4 + c_5}{\Delta x^2}, c_5 = c_R$$

2. Substitute boundary conditions to reduce number of equations:

$$\frac{dc_1}{dt} = \mathcal{D}\frac{c_L - 2c_1 + c_2}{\Delta x^2}, \frac{dc_2}{dt} = \mathcal{D}\frac{c_1 - 2c_2 + c_3}{\Delta x^2},$$
$$\frac{dc_3}{dt} = \mathcal{D}\frac{c_2 - 2c_3 + c_4}{\Delta x^2}, \frac{dc_4}{dt} = \mathcal{D}\frac{c_3 - 2c_4 + c_R}{\Delta x^2}$$

Instationary diffusion equation: temporal discretization

$$\frac{dc_i}{dt} = \mathcal{D}\frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2}$$

Time discretization: forward Euler (explicit)

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \mathcal{D}\frac{c_{i-1}^n - 2c_i^n + c_{i+1}^n}{\Delta x^2}$$

$$\Rightarrow c_i^{n+1} = \mathrm{Fo}\, c_{i-1}^n + (1 - 2\mathrm{Fo})c_i^n + \mathrm{Fo}\, c_{i+1}^n \quad \text{with } \mathrm{Fo} = \frac{\mathcal{D}\Delta t}{\Delta x^2}$$

Straightforward updating (explicit equation), simple to implement in a program but stability constraint $\mathrm{Fo} = \frac{\mathcal{D}\Delta t}{\Delta x^2} < \frac{1}{2}$!

Small $\Delta x \Rightarrow$ small $\Delta t \Rightarrow$ patience required ☹

Instationary diffusion equation: temporal discretization

$$\frac{dc_i}{dt} = \mathcal{D}\frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2}$$

Time discretization: backward Euler (implicit)

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \mathcal{D}\frac{c_{i-1}^{n+1} - 2c_i^{n+1} + c_{i+1}^{n+1}}{\Delta x^2}$$

$$\Rightarrow -\mathrm{Fo}\, c_{i-1}^{n+1} + (1 + 2\mathrm{Fo})c_i^{n+1} - \mathrm{Fo}\, c_{i+1}^{n+1} = c_i^n \quad \text{with } \mathrm{Fo} = \frac{\mathcal{D}\Delta t}{\Delta x^2}$$

Requires the solution of a system of linear equations, but no stability constraints!

Note: extension to higher order schemes (with time step adaptation) straightforward. Often second or third order optimal, because for each Euler-like step in the additional order an often large system needs to be solved (not treated in this course).

Solving the instationary diffusion equation: example

Solve the diffusion problem using explicit discretization:

$$\frac{\partial c_i}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} \quad \text{with} \quad \begin{array}{l} 0 \leq x \leq \delta,\ \delta = 5 \cdot 10^{-3}\ \text{m} \\ \delta/\Delta x = 100\ \text{grid cells} \\ \mathcal{D} = 1 \cdot 10^{-8}\ \text{m}^2\,\text{s}^{-1} \\ t_{\text{end}} = 5000\ \text{s} \\ c_{\text{L}} = 1\ \text{mol}\,\text{m}^{-3}\ \ c_{\text{R}} = 0\ \text{mol}\,\text{m}^{-3} \end{array}$$

Solving the instationary diffusion equation: example

Solve the diffusion problem using explicit discretization:

$$\frac{\partial c_i}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} \quad \text{with} \quad \begin{array}{l} 0 \leq x \leq \delta,\ \delta = 5 \cdot 10^{-3}\ \text{m} \\ \delta/\Delta x = 100\ \text{grid cells} \\ \mathcal{D} = 1 \cdot 10^{-8}\ \text{m}^2\,\text{s}^{-1} \\ t_{\text{end}} = 5000\ \text{s} \\ c_{\text{L}} = 1\ \text{mol}\,\text{m}^{-3}\ \ c_{\text{R}} = 0\ \text{mol}\,\text{m}^{-3} \end{array}$$

$$c_i^{n+1} = \text{Fo}\,c_{i-1}^n + (1 - 2\text{Fo})c_i^n + \text{Fo}\,c_{i+1}^n \quad \text{with}\ \text{Fo} = \frac{\mathcal{D}\Delta t}{\Delta x^2}$$

Solving the instationary diffusion equation: example

Solve the diffusion problem using explicit discretization:

$$\frac{\partial c_i}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} \quad \text{with} \quad \begin{array}{l} 0 \leq x \leq \delta,\ \delta = 5 \cdot 10^{-3}\ \text{m} \\ \delta/\Delta x = 100\ \text{grid cells} \\ \mathcal{D} = 1 \cdot 10^{-8}\ \text{m}^2\,\text{s}^{-1} \\ t_{\text{end}} = 5000\ \text{s} \\ c_{\text{L}} = 1\ \text{mol}\,\text{m}^{-3}\ \ c_{\text{R}} = 0\ \text{mol}\,\text{m}^{-3} \end{array}$$

$$c_i^{n+1} = \text{Fo}\,c_{i-1}^n + (1 - 2\text{Fo})c_i^n + \text{Fo}\,c_{i+1}^n \quad \text{with}\ \text{Fo} = \frac{\mathcal{D}\Delta t}{\Delta x^2}$$

1. Initialise variables
2. Compute time step so that $\text{Fo} \leq \frac{1}{2} \Rightarrow \Delta t = 0.125\text{s}$!
3. Compute 40000 time steps times 100 grid nodes!
4. Store solution

Solving the instationary diffusion equation: example

Initialise the variables and matrices:

```
Nx = 100;              % Nc grid points
Nt = 40000;            % Nt time steps
D = 1e-8;              % m/s
c_L = 1.0; c_R = 0;    % mol/m3
t_end = 5000.0;        % s
x_end = 5e-3;          % m

% Time step and grid size
dt = t_end/Nt;
dx = x_end/Nx;

% Fourier number
Fo=D*dt/dx/dx

% Initial matrices for solutions (Nx times Nt)
c = zeros(Nt+1,Nx+1);  % All concentrations are zero
c(:,1)   = c_L;      % Concentration at left side
c(:,Nx+1) = c_R;      % Concentration at right side

% Grid node and time step positions
x = linspace(0,x_end,Nx+1);
```

Solving the instationary diffusion equation: example

Compute the solution (nested time-and-grid loop):

```
for n = 1:Nt % time loop
    for i = 2:Nx % Nested loop for grid nodes
        c(n+1,i) = Fo*c(n,i-1) + (1-2*Fo)*c(n,i) + ...
            Fo*c(n,i+1);
    end
end
```

Solving the instationary diffusion equation: example

Compute the solution (nested time-and-grid loop):

```
for n = 1:Nt % time loop
    for i = 2:Nx % Nested loop for grid nodes
        c(n+1,i) = Fo*c(n,i-1) + (1-2*Fo)*c(n,i) + ...
            Fo*c(n,i+1);
    end
end
```

Plotting the solution at $t = \{12.5, 62.5, 125, 625, 5000\}$ s.

Solving the diffusion equation implicitly

Linear system $A\boldsymbol{x} = \boldsymbol{b}$ from $-\mathrm{Fo}c_{i-1}^{n+1} + (1+2\mathrm{Fo})c_i^{n+1} - \mathrm{Fo}c_{i+1}^{n+1} = c_i^n$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & 0 & \cdots & 0 \\ 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & \cdots & 0 \\ 0 & 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0^{n+1} \\ c_1^{n+1} \\ c_2^{n+1} \\ c_3^{n+1} \\ \vdots \\ c_m^{n+1} \end{pmatrix} = \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ c_3^n \\ \vdots \\ c_m^n \end{pmatrix}$$

Solving the diffusion equation implicitly

Linear system $A\boldsymbol{x} = \boldsymbol{b}$ from $-\mathrm{Fo}c_{i-1}^{n+1} + (1+2\mathrm{Fo})c_i^{n+1} - \mathrm{Fo}c_{i+1}^{n+1} = c_i^n$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & 0 & \cdots & 0 \\ 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & \cdots & 0 \\ 0 & 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0^{n+1} \\ c_1^{n+1} \\ c_2^{n+1} \\ c_3^{n+1} \\ \vdots \\ c_m^{n+1} \end{pmatrix} = \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ c_3^n \\ \vdots \\ c_m^n \end{pmatrix}$$

$1 \times c_0^{n+1} = c_0^n$ (boundary condition)

Solving the diffusion equation implicitly

Linear system $A\boldsymbol{x} = \boldsymbol{b}$ from $-\mathrm{Fo}c_{i-1}^{n+1} + (1+2\mathrm{Fo})c_i^{n+1} - \mathrm{Fo}c_{i+1}^{n+1} = c_i^n$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & 0 & \cdots & 0 \\ 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & \cdots & 0 \\ 0 & 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0^{n+1} \\ c_1^{n+1} \\ c_2^{n+1} \\ c_3^{n+1} \\ \vdots \\ c_m^{n+1} \end{pmatrix} = \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ c_3^n \\ \vdots \\ c_m^n \end{pmatrix}$$

$1 \times c_0^{n+1} = c_0^n$ (boundary condition)

$-\mathrm{Fo}c_0^{n+1} + (1+2\mathrm{Fo})c_1^{n+1} - \mathrm{Fo}c_2^{n+1} = c_1^n$

Solving the diffusion equation implicitly

Linear system $A\boldsymbol{x} = \boldsymbol{b}$ from $-\mathrm{Fo}c_{i-1}^{n+1} + (1+2\mathrm{Fo})c_i^{n+1} - \mathrm{Fo}c_{i+1}^{n+1} = c_i^n$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & 0 & \cdots & 0 \\ 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & \cdots & 0 \\ 0 & 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0^{n+1} \\ c_1^{n+1} \\ c_2^{n+1} \\ c_3^{n+1} \\ \vdots \\ c_m^{n+1} \end{pmatrix} = \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ c_3^n \\ \vdots \\ c_m^n \end{pmatrix}$$

$1 \times c_0^{n+1} = c_0^n$ (boundary condition)

$-\mathrm{Fo}c_0^{n+1} + (1+2\mathrm{Fo})c_1^{n+1} - \mathrm{Fo}c_2^{n+1} = c_1^n$

$-\mathrm{Fo}c_1^{n+1} + (1+2\mathrm{Fo})c_2^{n+1} - \mathrm{Fo}c_3^{n+1} = c_2^n$

Solving the diffusion equation implicitly

Linear system $A\boldsymbol{x} = \boldsymbol{b}$ from $-\mathrm{Fo}c_{i-1}^{n+1} + (1+2\mathrm{Fo})c_i^{n+1} - \mathrm{Fo}c_{i+1}^{n+1} = c_i^n$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & 0 & \cdots & 0 \\ 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & \cdots & 0 \\ 0 & 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0^{n+1} \\ c_1^{n+1} \\ c_2^{n+1} \\ c_3^{n+1} \\ \vdots \\ c_m^{n+1} \end{pmatrix} = \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ c_3^n \\ \vdots \\ c_m^n \end{pmatrix}$$

$1 \times c_0^{n+1} = c_0^n$ (boundary condition)

$-\mathrm{Fo}c_0^{n+1} + (1+2\mathrm{Fo})c_1^{n+1} - \mathrm{Fo}c_2^{n+1} = c_1^n$

$-\mathrm{Fo}c_1^{n+1} + (1+2\mathrm{Fo})c_2^{n+1} - \mathrm{Fo}c_3^{n+1} = c_2^n$

$-\mathrm{Fo}c_2^{n+1} + (1+2\mathrm{Fo})c_3^{n+1} - \mathrm{Fo}c_4^{n+1} = c_3^n$

Solving the diffusion equation implicitly

Linear system $A\boldsymbol{x} = \boldsymbol{b}$ from $-\mathrm{Fo}c_{i-1}^{n+1} + (1+2\mathrm{Fo})c_i^{n+1} - \mathrm{Fo}c_{i+1}^{n+1} = c_i^n$

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & 0 & \cdots & 0 \\ 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & -\mathrm{Fo} & \cdots & 0 \\ 0 & 0 & -\mathrm{Fo} & (1+2\mathrm{Fo}) & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \cdot \begin{pmatrix} c_0^{n+1} \\ c_1^{n+1} \\ c_2^{n+1} \\ c_3^{n+1} \\ \vdots \\ c_m^{n+1} \end{pmatrix} = \begin{pmatrix} c_0^n \\ c_1^n \\ c_2^n \\ c_3^n \\ \vdots \\ c_m^n \end{pmatrix}$$

$1 \times c_0^{n+1} = c_0^n$ (boundary condition)

$-\mathrm{Fo}c_0^{n+1} + (1+2\mathrm{Fo})c_1^{n+1} - \mathrm{Fo}c_2^{n+1} = c_1^n$

$-\mathrm{Fo}c_1^{n+1} + (1+2\mathrm{Fo})c_2^{n+1} - \mathrm{Fo}c_3^{n+1} = c_2^n$

$-\mathrm{Fo}c_2^{n+1} + (1+2\mathrm{Fo})c_3^{n+1} - \mathrm{Fo}c_4^{n+1} = c_3^n$

$1 \times c_m^{n+1} = c_m^n$ (boundary condition)

Solving the diffusion equation implicitly in Matlab

To solve the linear system, we need to define matrix A. It is clear that storing many zeros is not efficient in terms of memory. We use a *sparse matrix* format:

```
% Bands in matrix (internal cells)
A = sparse(Nx+1,Nx+1);
for i=2:Nx
    A(i,i-1) = -Fo;
    A(i,i  ) = (1+2*Fo);
    A(i,i+1) = -Fo;
end

% Set boundary cells, independent on neighbors:
A(1,1 ) = 1;          % Left
A(Nx+1,Nx+1) = 1;     % Right
```

Solving the diffusion equation implicitly in Matlab

The command `spy(A)` shows a figure with the non-zero positions.

Solving the diffusion equation implicitly in Matlab

The concentration matrix is initialised and the boundary conditions are set as follows:

```
% Initial matrices for solutions (Nx times Nt)
c = zeros(Nt+1,Nx+1);  % All concentrations are zero
c(:,1)     = c_L;      % Concentration at left side
c(:,Nx+1)  = c_R;      % Concentration at right side
```

The right hand side vector ($\boldsymbol{b}$) can now be set during the time-loop:

```
for n = 1:Nt-1          % time loop
    b = c(n,:)';        % Set right hand side
    solX = A\b;         % Solve linear system
    c(n+1,:) = solX;    % Store solution each time step
end
```

Solving the diffusion equation implicitly in Matlab

Plotting the solution at $t = \{12.5, 62.5, 125, 625, 5000\}$ s.

About explicit vs. implicit solutions

- Explicit solution:
  - Easy to implement
  - Very small time steps required.
  - This problem took about 0.5 s.
- Implicit solution:
  - Harder to implement, needs sparse matrix solver
  - No stability constraint
  - This problem took about 0.05 s
- The difference will become much larger for systems with e.g. more grid nodes!

Extension with non-linear source terms

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} + R(c) \quad \text{with} \quad \begin{array}{l} t = 0; 0 \leq x \leq \ell \Rightarrow c = c_0 \\ t > 0; x = 0 \Rightarrow c = c_L \\ t > 0; x = \ell \Rightarrow c = c_R \end{array}$$

Extension with non-linear source terms

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} + R(c) \quad \text{with} \quad \begin{aligned} &t = 0; 0 \leq x \leq \ell \Rightarrow c = c_0 \\ &t > 0; x = 0 \Rightarrow c = c_L \\ &t > 0; x = \ell \Rightarrow c = c_R \end{aligned}$$

- Forward Euler (explicit): simply add to right-hand side

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \mathcal{D}\frac{c_{i-1}^n - 2c_i^n + c_{i+1}^n}{\Delta x^2} + R(c_i^n)$$

$$\Rightarrow c_i^{n+1} = \mathrm{Fo}\, c_{i-1}^n + (1 - 2\mathrm{Fo})c_i^n + \mathrm{Fo}\, c_{i+1}^n + R_i^n \Delta t$$

Extension with non-linear source terms

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} + R(c) \quad \text{with} \quad \begin{aligned} &t = 0; 0 \le x \le \ell \Rightarrow c = c_0 \\ &t > 0; x = 0 \Rightarrow c = c_L \\ &t > 0; x = \ell \Rightarrow c = c_R \end{aligned}$$

- Forward Euler (explicit): simply add to right-hand side

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \mathcal{D}\frac{c_{i-1}^n - 2c_i^n + c_{i+1}^n}{\Delta x^2} + R(c_i^n)$$

$$\Rightarrow c_i^{n+1} = \mathrm{Fo}c_{i-1}^n + (1 - 2\mathrm{Fo})c_i^n + \mathrm{Fo}c_{i+1}^n + R_i^n \Delta t$$

- Backward Euler (implicit): linearization required

$$R(c_i^{n+1}) = R(c_i^n) + \left.\frac{dR}{dc}\right|_i^n (c_i^{n+1} - c_i^n)$$

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = \mathcal{D}\frac{c_{i-1}^{n+1} - 2c_i^{n+1} + c_{i+1}^{n+1}}{\Delta x^2} + R(c_i^{n+1})$$

$$\Rightarrow -\mathrm{Fo}c_{i-1}^{n+1} + (1 + 2\mathrm{Fo} - \left.\frac{dR}{dc}\right|_i^n \Delta t)c_i^{n+1} - \mathrm{Fo}c_{i+1}^{n+1} = c_i^n + \left(R_i^n - \left.\frac{dR}{dc}\right|_i^n c_i^n\right)\Delta t$$

Today's outline

- Introduction
- Instationary diffusion equation
  - Discretization
  - Solving the diffusion equation
  - Non-linear source terms
- Convection
  - Discretization
  - Central difference scheme
  - Upwind scheme
- Conclusions
  - Other methods
  - Summary

Extension with convection terms

$$\frac{\partial c}{\partial t} = \mathcal{D}\frac{\partial^2 c}{\partial x^2} - u\frac{\partial c}{\partial x} + R$$

Discretization of first derivative $\frac{dc}{dx}$,
looks simple but is numerical headache!

Central discretization:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

$\Rightarrow$ simple and easy, too bad it doesn't work: yields unstable solutions if convection dominated.

Central difference scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x}$$

Central difference for first derivative:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Central difference scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x}$$

Central difference for first derivative:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Forward Euler discretization of temporal and spatial domain:

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = -u\frac{c_{i+1} - c_{i-1}}{2\Delta x} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_{i+1}^n - c_{i-1}^n}{2\Delta x}\Delta t$$

Central difference scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x}$$

Central difference for first derivative:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Forward Euler discretization of temporal and spatial domain:

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = -u\frac{c_{i+1} - c_{i-1}}{2\Delta x} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_{i+1}^n - c_{i-1}^n}{2\Delta x}\Delta t$$

Central difference scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x}$$

Central difference for first derivative:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Forward Euler discretization of temporal and spatial domain:

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = -u\frac{c_{i+1} - c_{i-1}}{2\Delta x} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_{i+1}^n - c_{i-1}^n}{2\Delta x}\Delta t$$

Central difference scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x}$$

Central difference for first derivative:

$$\frac{dc}{dx} = \frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

Forward Euler discretization of temporal and spatial domain:

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = -u\frac{c_{i+1} - c_{i-1}}{2\Delta x} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_{i+1}^n - c_{i-1}^n}{2\Delta x}\Delta t$$

Extension with convection terms

Solution: upwind discretization, like CSTR's in series:

First order upwind: $$-u\frac{dc}{dx}\bigg|_i = \begin{cases} -u\dfrac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\ -u\dfrac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases}$$

Extension with convection terms

Solution: upwind discretization, like CSTR's in series:

First order upwind: $-u\left.\frac{dc}{dx}\right|_i = \begin{cases} -u\dfrac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\ -u\dfrac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases}$ Stable if $\mathrm{Co} = \frac{u\Delta t}{\Delta x} < 1$ (with Co the Courant number). However, only 1st order accurate (large smearing of concentration fronts). Higher order upwind requires TVD schemes (trick of the trade)...

First order upwind scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x}$$

Upwind scheme for first derivative:

$$-u\left.\frac{dc}{dx}\right|_i = \begin{cases} -u\dfrac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\[2ex] -u\dfrac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases}$$

First order upwind scheme of 1st derivative

Unsteady convection:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x}$$

Upwind scheme for first derivative:

$$-u\left.\frac{dc}{dx}\right|_i = \begin{cases} -u\dfrac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\ -u\dfrac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases}$$

Forward Euler discretization of temporal and spatial domain:

$$\frac{c_i^{n+1} - c_i^n}{\Delta t} = -u\frac{c_{i+1} - c_{i-1}}{2\Delta x}$$

$$\Rightarrow c_i^{n+1} = \begin{cases} c_i^n - u\dfrac{c_i - c_{i-1}}{\Delta x} & \text{if } u \geq 0 \\ c_i^n - u\dfrac{c_{i+1} - c_i}{\Delta x} & \text{if } u < 0 \end{cases}$$

Upwind scheme: example

Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1\,\mathrm{m\,s^{-1}} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_i - c_{i-1}}{\Delta x}\Delta t$$

Upwind scheme: example

Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1\,\mathrm{m\,s^{-1}} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_i - c_{i-1}}{\Delta x}\Delta t$$

Using 100 grid cells

Upwind scheme: example

Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1\,\mathrm{m\,s^{-1}} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_i - c_{i-1}}{\Delta x}\Delta t$$

Using 100 grid cells

Upwind scheme: example

Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1\,\mathrm{m\,s^{-1}} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_i - c_{i-1}}{\Delta x}\Delta t$$

Using 100 grid cells

Upwind scheme: example

Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1\,\mathrm{m\,s^{-1}} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_i - c_{i-1}}{\Delta x}\Delta t$$

Using 100 grid cells

Using 1000 grid cells

Upwind scheme: example

Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1\,\mathrm{m\,s^{-1}} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_i - c_{i-1}}{\Delta x}\Delta t$$

Using 100 grid cells

Using 1000 grid cells

Upwind scheme: example

Unsteady convection through a pipe:

$$\frac{\partial c}{\partial t} = -u\frac{\partial c}{\partial x} \quad \text{with} \quad u = 0.1\,\mathrm{m\,s^{-1}} \Rightarrow c_i^{n+1} = c_i^n - u\frac{c_i - c_{i-1}}{\Delta x}\Delta t$$

Using 100 grid cells

Using 1000 grid cells

Central difference and upwind in Matlab

The results from the previous slides were computed using this script:

```
Nx = 1000;            % Nc grid points
Nt = 10000;           % Nt time steps
u = 0.001;            % m/s
c_in = 1.0;           % mol/m3
t_end = 100.0;        % s
x_end = 0.1;          % m

% Time step and grid size
dt = t_end/Nt; dx = x_end/Nx;

% Courant number
Co=u*dt/dx

% Initial matrices for solutions (Nx times Nt)
c1 = zeros(Nt+1,Nx+1);  % All concentrations are zero
c1(:,1) = c_in;     % Concentration at inlet (all time steps)
an = c1; c2 = c1;   % Analytical and upwind solution

% Grid node and time step positions
x = linspace(0,x_end,Nx+1);
t = linspace(0,t_end,Nt+1);
```

Central difference and upwind in Matlab

(continued)

```
for n = 1:Nt % time loop
    for i = 2:Nx % Nested loop for grid nodes
        % Central difference
        c1(n+1,i) = c1(n,i) - u*((c1(n,i+1) - ...
            c1(n,i-1))/(2*dx))*dt;
        % Upwind
        c2(n+1,i) = c2(n,i) - u*((c2(n,i) - ...
            c2(n,i-1))/(dx))*dt;
        % Analytical
        an(n+1,i) = (x(i) < u*t(n+1))*c_in;
    end
end
```

Today's outline

- Introduction
- Instationary diffusion equation
  - Discretization
  - Solving the diffusion equation
  - Non-linear source terms
- Convection
  - Discretization
  - Central difference scheme
  - Upwind scheme
- Conclusions
  - Other methods
  - Summary

Extension to systems of PDE's

- Explicit methods: straightforward extension
- Implicit methods: yields block-tridiagonal matrix (note ordering of equations: all variables per grid cell)

Extension to 2D or 3D systems

Spatial discretization in 2 directions — different methods available:

- Explicit
- Fully implicit
  - 1D gives tri-diagonal matrix
  - 2D gives penta-diagonal matrix
  - 3D gives hepta-diagonal matrix

  Use of dedicated matrix solvers (e.g. ICCG, multigrid, ...)
- Alternating direction implicit (ADI)
  - Per direction implicit, but still overall unconditionally stable

Further extensions for parabolic PDEs

- Higher order temporal discretization (multi-step) with time step adaptation
- Non-uniform grids with automatic grid adaptation
- Higher-order discretization methods, especially higher order TVD (flux delimited) schemes for convective fluxes (e.g. WENO schemes)
- Higher-order finite volume schemes (Riemann solvers)

Summary

- Several classes of PDEs were introduced
  - Elliptic, Parabolic, Hyperbolic PDEs
- Diffusion equation: discretization of temporal and spatial domain was discussed
  - Solutions of the diffusion equation using explicit and implicit methods
  - How to add non-linear source terms
- Convection: upwind vs. central difference schemes