Numerical Computation using MATLAB

Debojit, Dibyendu, Javed, Goutam, Suprotim
Department of Physics, **Indian Institute of Technology Kanpur**

2025-11-04

Recap

What have we learned so far?

- What is MATLAB, how to access the online MATLAB portal
- Basics of MATLAB, **variables and Data Types**
- **Conditional statements** (`if, elseif, else` and `switch`)
- **Loops** (`for` and `while`)
- **MATLAB's in-buillt functions** (`sin`, `cos`, `exp` etc.)
- Plotting in MATLAB, **2D plots**; **3D plots**

What is numerical computation?

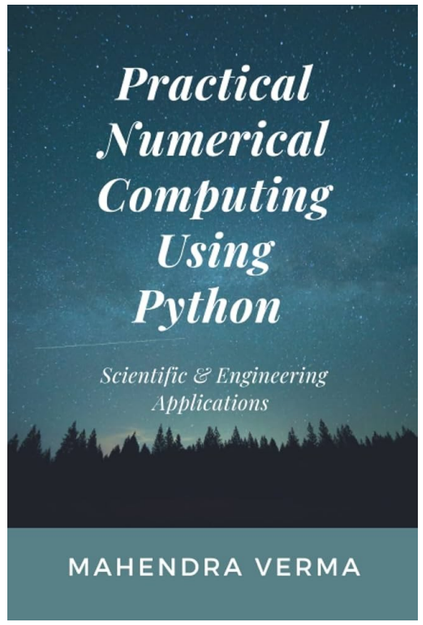
Crunching numbers, going beyond Symbolic math

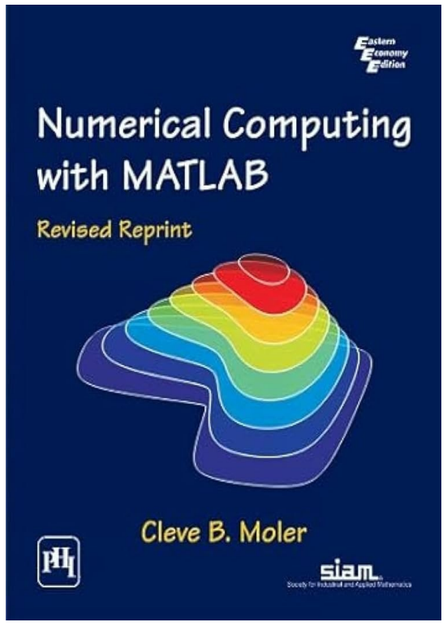
- **Why?** Real data are often obtained in form of numbers
- **What?** Interpolation, integration, differentiation etc.
- **How?** Let's see in the next few slides

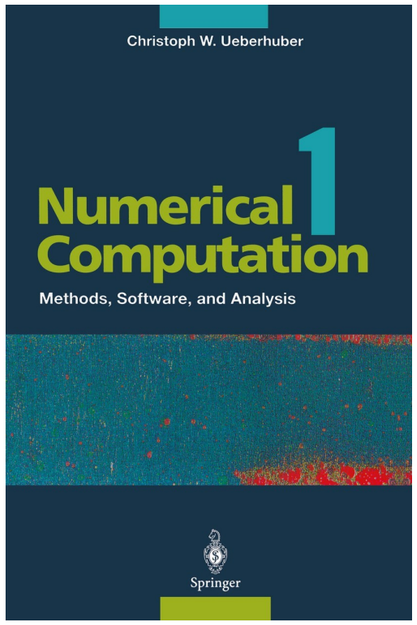
We will learn *concepts / theory about the numerical computation*, **how to write the code from scratch**, and pre-built functions in MATLAB

Resources

The concepts discussed here are mentioned in details in the following books;





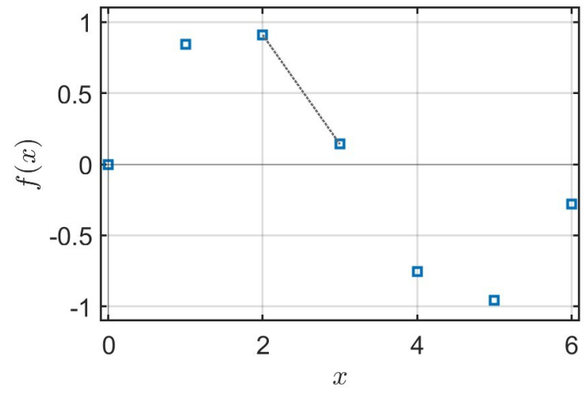


Interpolation

What is interpolation?

- *Interpolation* is a type of **estimation**
- Method of finding new data points based on *some known data points*

x	$f(x)$
0	0
1	0.8415
2	0.9093
3	0.1411
4	−0.7568
5	−0.9589
6	−0.2794



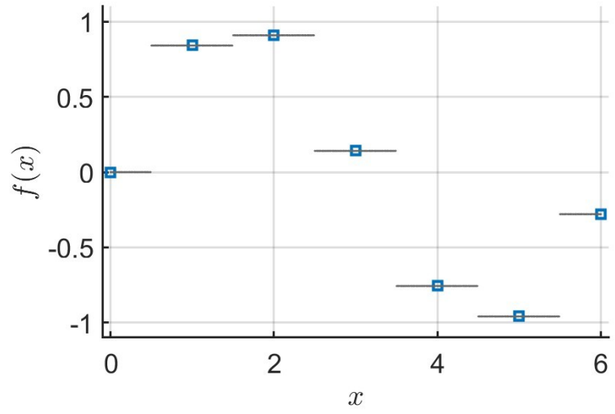
- What will be the value of $f(x = 2.5)$?

Different kinds of *interpolation*

- Nearest neighbor interpolation
- Linear interpolation
- Polynomial interpolation
- Spline interpolation

Nearest neighbor interpolation

- The simplest interpolation method
- Find out the nearest data value, and assign the same value.
- **This method is rarely used**



Linear interpolation

- Unknown values are obtained from a *straight line* approximation

- Consider the above example of estimating $f(x = 2.5)$.
- Since 2.5 is midway between 2 and 3, it is reasonable to take $f(2.5)$ midway between $f(2) = 0.9093$ and $f(3) = 0.1411$
- Generally, linear interpolation takes two data points, say (x_a, y_a) and (x_b, y_b), and approximate the values in between (x_a, x_b) *via a straight line*

> **Note**

$$y = y_a + (y_b - y_a)\frac{x - x_a}{x_b - x_a}$$

Implementing linear interpolation

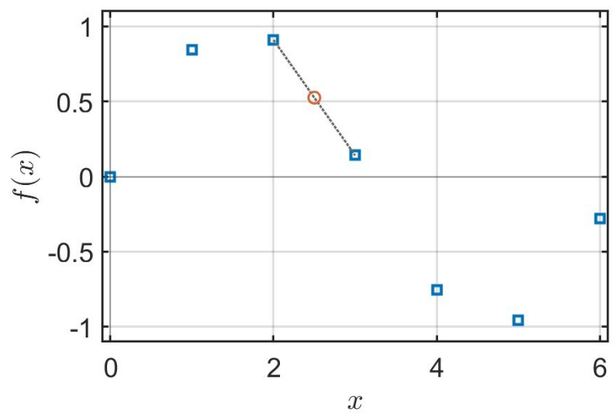
Let’s see how we can use linear interpolation.

```
x = 0:6;
y = [0, 0.8415, 0.9093, 0.1411, -0.7568, -0.9589, -0
plot(x, y, "s")
hold on

% What is the unknown point?
ux = 2.5;

% Find the points (xa, xb) in between
% which the unknown point lies
x_greater_than_ux = x(x - ux >= 0);
y_greater_than_ux = y(x - ux >= 0);
x_smaller_than_ux = x(x - ux <= 0);
y_smaller_than_ux = y(x - ux <= 0);
xb = x_greater_than_ux(1)
yb = y_greater_than_ux(1)
xa = x_smaller_than_ux(end)
ya = y_smaller_than_ux(end)

uy = ya + (yb -ya) * (ux- xa)/(xb-xa)
plot([xa, xb], [ya, yb])
plot([ux], [uy], "o")
```



- The estimated value lies on the staright line, as expected

Implementing linear interpolation (contd)

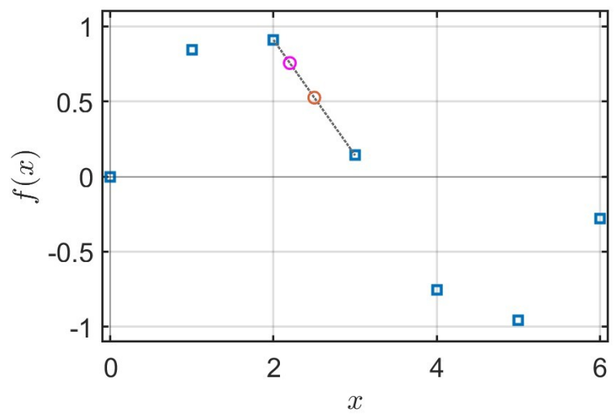
Let’s see how we can use linear interpolation.

```
x = 0:6;
y = [0, 0.8415, 0.9093, 0.1411, -0.7568, -0.9589, -0
plot(x, y, "s")
hold on

% What is the unknown point?
ux = 2.2;

% Find the points (xa, xb) in between
% which the unknown point lies
x_greater_than_ux = x(x - ux >= 0);
y_greater_than_ux = y(x - ux >= 0);
x_smaller_than_ux = x(x - ux <= 0);
y_smaller_than_ux = y(x - ux <= 0);
xb = x_greater_than_ux(1)
yb = y_greater_than_ux(1)
xa = x_smaller_than_ux(end)
ya = y_smaller_than_ux(end)

uy = ya + (yb -ya) * (ux- xa)/(xb-xa)
plot([xa, xb], [ya, yb])
plot([ux], [uy], "o")
```



Implementing linear interpolation (contd)

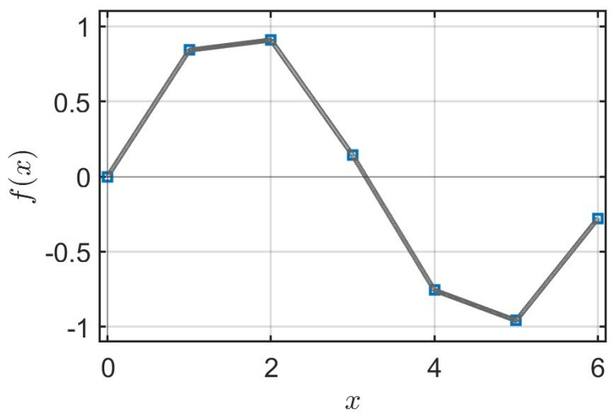
Now, we want to estimate the unknown values for the whole range; **use for loop**

```
% What are the unknown points?
ux_arr = 0:0.01:6;
uy_arr = []

for ux = ux_arr
    % Find the points (xa, xb)
    % such that ux belongs to (xa, xb)
    x_greater_than_ux = x(x - ux >= 0);
    y_greater_than_ux = y(x - ux >= 0);
    x_smaller_than_ux = x(x - ux <= 0);
    y_smaller_than_ux = y(x - ux <= 0);
    xb = x_greater_than_ux(1)
    yb = y_greater_than_ux(1)
    xa = x_smaller_than_ux(end)
    ya = y_smaller_than_ux(end)

    uy = ya + (yb -ya) * (ux- xa)/(xb-xa)
    uy_arr = [uy_arr uy]
end

plot(ux_arr, uy_arr, "o")
```



This is linear interpolation !!

Implementing linear interpolation (contd)

We can do the same thing just using the **in-built functions of MATLAB** also.

```
% What are the unknown point?
ux_arr = 0:0.01:6;

% You can simply use the MATLAB function `interp1`
% interp1(x_values, y_values, unknown_points)
uy_arr = interp1(x, y, ux_arr)
plot(ux_arr, uy_arr, "o")
```

- This will generate same result as the previous code
- The plot is not shown here, this is an exercise, check during the hands-on session
- You can also specify different interpolation methods in the `interp1` function

```
uy_arr = interp1(x, y, ux_arr, "linear")
% Alternative interpolation method:
% 'linear', 'nearest', 'next', 'previous', 'pchip',
% 'cubic', 'v5cubic', 'makima', or 'spline'
```

- We just used the `linear` method, which is the default

Polynomial interpolation

- Estimamte the unknown value by considering a n-order polynomial function
- Very similalr to fitting the data with a **polynomial curve**
- For n data points, there is exactly one polynomial of degree at most $n - 1$ going through all the data points
- Polynomial interpolation has some disadvantages: **Computationally expensive**, **may exhibit oscillatory artifacts**, etc.

> **Note**
>
> **Better options are available !!**

Spline interpolation

- Spline interpolation uses **piecewise polynomials** (splines) to approximate a smooth curve through the known data points
- With Splines, one can often achieve good accuracy with lower-degree polynomials

> **Caution**
>
> Polynomial interpolation uses a single polynomial of a certain degree to fit all points

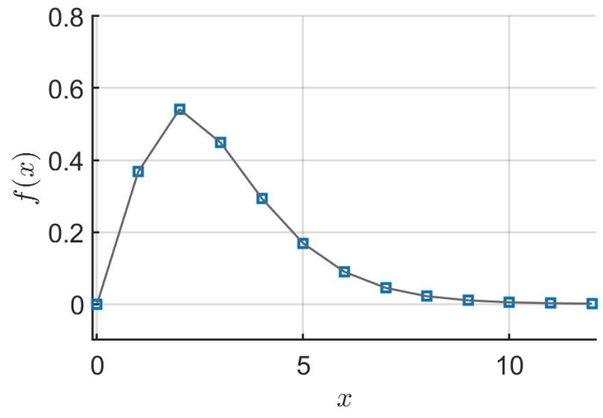
```
x = 0:6;
y = [0, 0.8415, 0.9093, 0.1411, -0.7568, -0.9589, -0.2794];
plot(x, y, "s");
hold on

ux_arr = 0:0.01:6;
uy_arr = interp1(x, y, ux_arr, "spline")
```

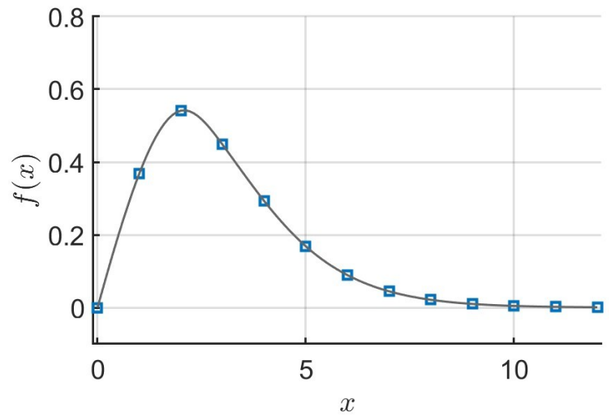
Linear vs. Spline interpolation

Now, let us compare the linear and spline interpolation results for some different datasets.

Linear interpolation



Spline interpolation



Differentiation

Recalling the basics

We all know how to do differentiations, right?

$$\frac{d}{dx}e^x = e^x \qquad \frac{d}{dx}\sin(x) = \cos(x) \qquad \frac{d}{dx}x^n = nx^{n-1} \qquad \frac{d}{dx}\ln(x) = \frac{1}{x}$$

- But, we are talking about differentiating numbers
- Let's go to the basics;

> **Note**

$$\frac{d}{dx}f(x)\bigg|_{x=x_1} = \lim_{x_2 - x_1 \to 0} \frac{f(x_2) - f(x_1)}{x_2 - x_1}$$

Differentiating discrete data points?

For discrete datapoints, the formula shown earlier becomes; $y \equiv f(x)$

> Note

$$\left.\frac{dy}{dx}\right|_{x=x_{i-1}} = \frac{\Delta y_i}{\Delta x_i} = \frac{y_i - y_{i-1}}{x_i - x_{i-1}}$$

How to interpret this ??

index	1	2	3	4	5	6	7
x	$x(1)$	$x(2)$	$x(3)$	$x(4)$	$x(5)$	$x(6)$	$x(7)$
y	$y(1)$	$y(2)$	$y(3)$	$y(4)$	$y(5)$	$y(6)$	$y(7)$

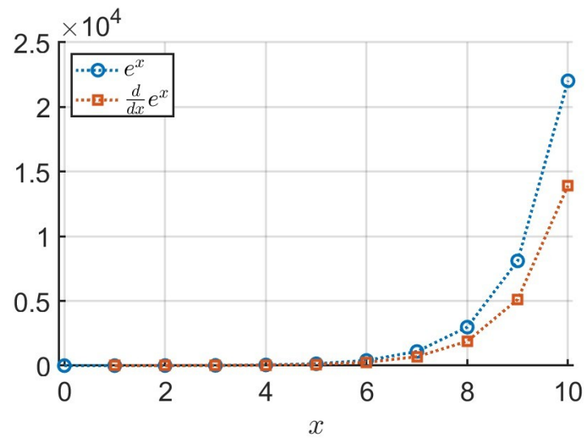
We omitted the `limit`, can anyone tell what the effect will be?

How to implement this ?

```
% Implementing differentiation
x = 0:10;
y = exp(x)
plot(x, y, ":o")
hold on

dydx = []
for i = 2:length(x)
    dydx_i = (y(i) - y(i-1)) / (x(i) - x(i-1))
    dydx = [dydx dydx_i]
end

plot(x(2:end), dydx, ":s")
```



- *Is this correct?*
- Are they matching?
- **Why is this happening?**

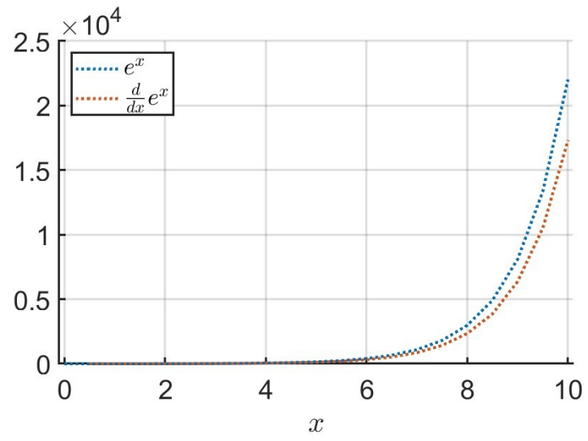
How to implement this (contd)?

Let us reduce the step size of x.

```
% Implementing differentiation
x = 0:0.5:10;
y = exp(x)
plot(x, y, "o")
hold on

dydx = []
for i = 2:length(x)
    dydx_i = (y(i) - y(i-1)) / (x(i) - x(i-1))
    dydx = [dydx dydx_i]
end

plot(x(2:end), dydx, "s")
```



Now, the derivative is almost matching the expected outcome.

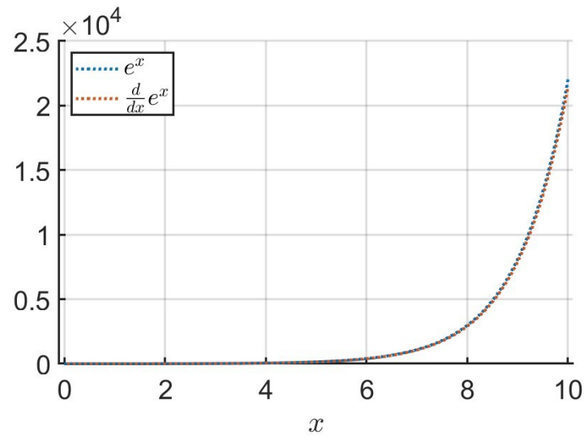
How to implement this (contd)?

Let us reduce the step size of x even further.

```
% Implementing differentiation
x = 0:0.05:10;
y = exp(x)
plot(x, y, "o")
hold on

dydx = []
for i = 2:length(x)
    dydx_i = (y(i) - y(i-1)) / (x(i) - x(i-1))
    dydx = [dydx dydx_i]
end

plot(x(2:end), dydx, "s")
```



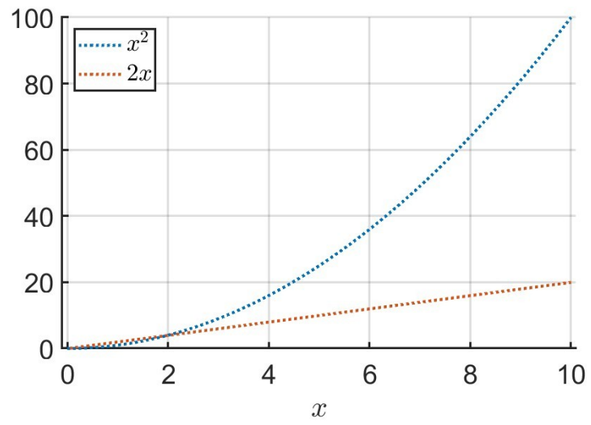
How to implement this (contd)?

Let us look at another function, $f(x) = x^2$.

```
% Implementing differentiation
x = 0:0.05:10;
y = x.^2
plot(x, y, "o")
hold on

dydx = []
for i = 2:length(x)
    dydx_i = (y(i) - y(i-1)) / (x(i) - x(i-1))
    dydx = [dydx dydx_i]
end

plot(x(2:end), dydx, "s")
```



We have to be careful about the step size of the independent variables, while performing derivatives numerically.

How to implement this (contd)?

Here also, we can use some buitin MATLAB functions, we will look at `diff`

```
% Implementing differentiation
x = 0:0.05:10;
y = exp(x)
plot(x, y, "o")
hold on

dydx = diff(y) ./ diff(x)

plot(x(2:end), dydx, "s")
```

- Like the interpolation case, I have not shown the plotted results here, check it during the hands-on session
- **Are there any other functions we can use?**
- **Search the internet :D**

Integration

Integration of numbers, start from the basics

We all know how to do differentiations, right?

$$\int e^x dx = e^x + c_1 \qquad \int \cos(x) dx = \sin(x) + c_2 \qquad \int \frac{1}{x} dx = \ln(x) + c_3$$

- But, again, we are talking about integration of numbers
- Let's go to the basics

> **Note**

$$I = \int_a^b y dx \equiv \sum_a^b y \Delta x \qquad I_{i+1} - I_i = y_i \Delta x_i$$

Need to provide an initial condiiton

This is called Euler forward method !!

Implementing Euler's forward method

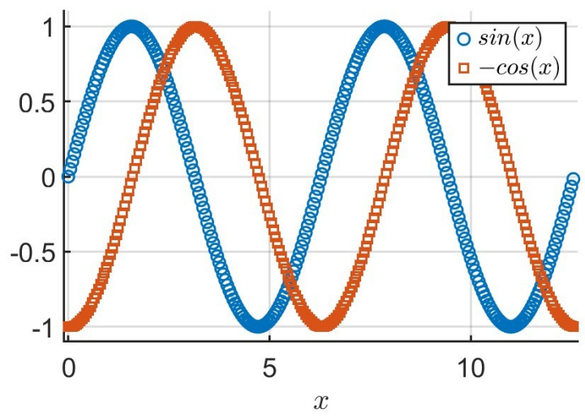
Here also, we have to make sure that the step size is very small. Let us take $\Delta x = 0.05$.

```
x = 0:0.05:4*pi;
dy = sin(x);
hold on

y = zeros(1, length(x));
y(1) = -1;

% Euler's Method
for i = 1:length(x)-1
    y(i+1) = y(i) + 0.05*dy(i);
end

plot(x, dy, "o", MarkerSize=8, LineWidth=1.5)
plot(x, y, "s", MarkerSize=8, LineWidth=1.5)
```



Implementing Euler's method (contd)

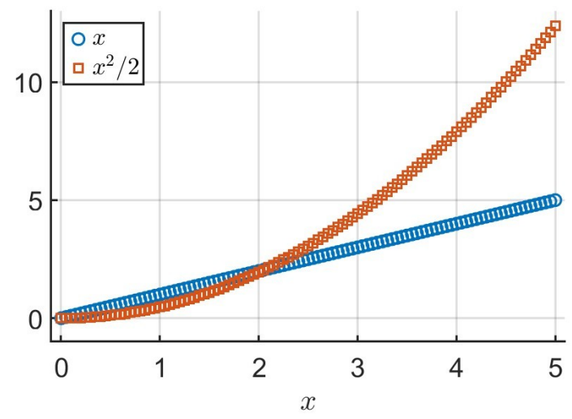
Let's try integrating a different function, $f(x) = x$

```
x = 0:0.05:4*pi;
dy = x;
hold on

y = zeros(1, length(x));
y(1) = -1;

% Euler's Method
for i = 1:length(x)-1
    y(i+1) = y(i) + 0.05*dy(i);
end

plot(x, dy, "o", MarkerSize=8, LineWidth=1.5)
plot(x, y, "s", MarkerSize=8, LineWidth=1.5)
```



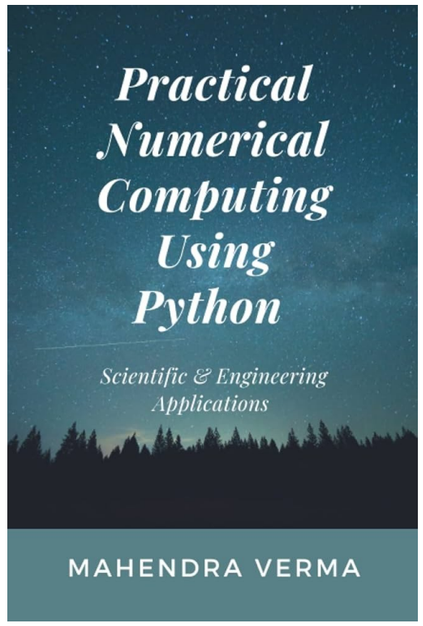
You will try out other problems in the hands-on session !!

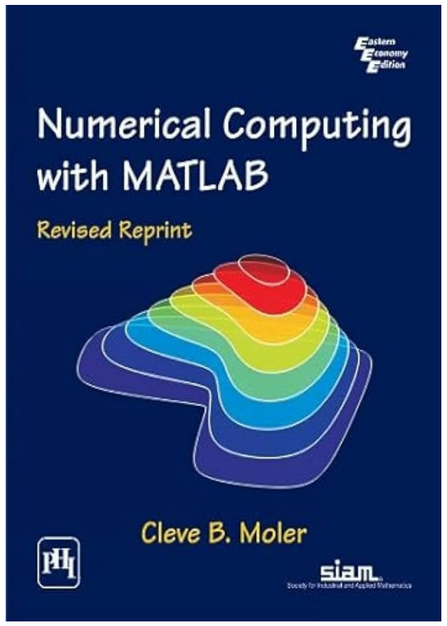
Other integration techniques

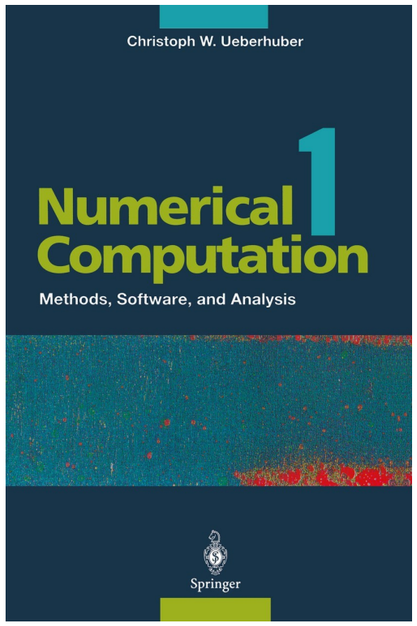
Every case is different;

- Quadrature rules (Trapezoidal rule, Simpson's 1/3rd rule etc.) $\longrightarrow$ *Newton-Cotes formula*
- **Leap-frog method**
- **Integrals over infinite intervals**: Gauss-Hermite, Gauss-Laguerre quadrature rules
- **Monte Carlo**
- **Bayesian quadrature**

Further reading







THANK YOU VERY MUCH !!