SM-2302: Software for Mathematicians

Lecture 3: Solving Equations, Curve Fitting & Numerical Techniques

Dr Huda Ramli
Mathematical Sciences, FOS, UBD
`huda.ramli@ubd.edu.bn`

Semester I, 2025/26

Adapted from 6.057 IAP 2019 (`MIT OCW`)

Outline

Lecture 3: Solving Equations, Curve Fitting & Numerical Techniques

- Linear Algebra
- Polynomials
- Optimization
- Differentiation & Integration
- Differential Equations

Systems of linear equations

- Given the system of equations:

$$\begin{aligned} x + 2y - 3z &= 5 \\ -3x - y + z &= -8 \\ x - y + z &= 0 \end{aligned}$$

- Represent the system as $\mathbf{A}\mathbf{x} = \mathbf{b}$:

```
A = [1  2 -3; -3 -1  1; 1 -1  1];
b = [5; -8; 0];
% Solve with one line of code:
x = A \ b;
```

- Returns a 3×1 vector `x` containing the values of x, y, and z
- The backslash operator \ works with:
  - Square systems ($\mathbf{A}$ is square): exact solution
  - Rectangular systems: least squares solution (overdetermined or underdetermined)

MATLAB makes solving linear algebra problems fast and intuitive!

Worked examples

System 1:

$$\begin{aligned} x + 4y &= 34 \\ -3x + y &= 2 \end{aligned}$$

```
A = [1  4; -3  1];
b = [34; 2];
rank(A)
x = inv(A)*b;
x = A\b;
```

System 2:

$$\begin{aligned} 2x - 2y &= 4 \\ -x + y &= 3 \\ 3x + 4y &= 2 \end{aligned}$$

```
A = [2  -2; -1  1; 3  4];
b = [4; 3; 2];
rank(A) % rectangular matrix
x = A\b; % least squares solution
error = abs(A*x-b)
```

More linear algebra

Given the matrix:

$$\mathbf{M} = \begin{bmatrix} 1 & 2 & -3 \\ -3 & -1 & 1 \\ 1 & -1 & 1 \end{bmatrix}$$

Calculate:

`rank(M);` rank (the number of linearly independent rows or columns)

`det(M);` determinant (matrix must be square):
matrix invertible if the determinant is nonzero

`inv(M);` matrix inverse:
if an equation is of the form $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{A}$ a square matrix,
`x=A\b` is (mostly) the same as `x=inv(A)*b`

`c=cond(M);` conditional number:
if condition number is large when solving `A*x=b`, small errors in `b` can lea
to large errors in `x` (optimal c=1)

MAtrix decompositions

MATLAB has many built-in matrix decomposition methods. Some common ones are:

`>> [V,D] = eig(M)`	Eigenvalue decomposition
`>> [U,S,V] = svd(M)`	Singular value decomposition
`>> [Q,R] = qr(M)`	QR decomposition
`>> L,U] = lu(M)`	LU decomposition
`>> R = chol(M)`	Cholesky decomposition (M must be positive definite)

Example 1 (Fitting Polynomials)

Goal: Find the best second-order polynomial:

$$y = ax^2 + bx + c$$

that fits the following points: $(-1, 0),\ (0, -1),\ (2, 3).$

1. Substitute the points into the equation to form a system:

$$\begin{aligned} a(-1)^2 + b(-1) + c &= 0 \\ a(0)^2 + b(0) + c &= -1 \\ a(2)^2 + b(2) + c &= 3 \end{aligned}$$

2. Represent as a linear system $\mathbf{A}\mathbf{x} = \mathbf{y}$ and solve for `x = [a b c]`.

Example 1 (Fitting Polynomials)

Goal: Find the best second-order polynomial:

$$y = ax^2 + bx + c$$

that fits the following points: $(-1, 0),\ (0, -1),\ (2, 3)$.

1. Substitute the points into the equation to form a system:

$$\begin{aligned} a(-1)^2 + b(-1) + c &= 0 \\ a(0)^2 + b(0) + c &= -1 \\ a(2)^2 + b(2) + c &= 3 \end{aligned}$$

2. Represent as a linear system $\mathbf{Ax} = \mathbf{y}$ and solve for `x = [a b c]`.

```
A = [1 -1 1; 0 0 1; 4 2 1]; y = [0; -1; 3]; x = A \ y;
```

Outline

Lecture 3: Solving Equations, Curve Fitting & Numerical Techniques

- Linear Algebra
- Polynomials
- Optimization
- Differentiation & Integration
- Differential Equations

Polynomials

- Many functions can be well described by a high-order polynomial
- MATLAB represents polynomials by a vector of coefficients:

$$P(x) = \underset{\substack{\uparrow \\ \texttt{P(1)}}}{a}\, x^3 + \underset{\substack{\uparrow \\ \texttt{P(2)}}}{b}\, x^2 + \underset{\substack{\uparrow \\ \texttt{P(3)}}}{c}\, x + \underset{\substack{\uparrow \\ \texttt{P(4)}}}{d}$$

- `P = [1 0 -2]` represents the polynomial $x^2 - 2$
- `P = [2 0 0 0]` represents the polynomial $2x^3$

Polynomial operations

`P` is a vector of length N+1 describing N-th order polynomial.

`r = roots(P)`	Gets the roots of a polynomial (`r` is a vector of length N)
`P = poly(r)`	Gets polynomial from the roots
`y0 = polyval(P, x0)`	Evaluates a polynomial at a single point `(x0, y0)`
`y = polyval(P, x)`	Evaluates a polynomial at many points, where `x` and `y` are vectors of the same size

Polynomial fitting

MATLAB simplifies the process of fitting polynomials to data.

Example:

```
X = [-1 0 2]; Y = [0 -1 3]; % data vectors
p2 = polyfit(X, Y, 2); % finds best 2nd order poly that fits ...
    points (-1,0), (0,-1) and (2,3)

plot(X, Y, 'o', 'MarkerSize', 8);
hold on;
x = -3:0.01:3;
plot(x, polyval(p2,x), 'r--');
```

Example 2 (Polynomial fitting)

Goal: Fit a 2nd-degree polynomial to noisy data.

1. Generate clean data:

$$x = -4 : 0.1 : 4, \quad y = x^2$$

2. Add noise to the data
   - Use `randn` to generate random noise
   - Add noise to `y`, store in `y_noisy`
   - Plot noisy data using `'.'` markers
3. Fit a 2nd degree polynomial using `polyfit` and evaluate the fitted polynomial using `polyval`
4. Plot the fitted curve on top of the noisy data, using the same `x` values and a red line.

Outline

Lecture 3: Solving Equations, Curve Fitting & Numerical Techniques

- Linear Algebra
- Polynomials
- Optimization
- Differentiation & Integration
- Differential Equations

Nonlinear root finding

- Many real-world problems require us to solve $f(x) = 0$
- In MATLAB, we can use `fzero` to calculate roots for any arbitrary function
- `fzero` requires a function as input:

```
>> x = fzero('myfun',1);
```

or

```
>> x = fzero(@myfun,1);
```

  where 1 indicates a point near the estimated root location.
- But a separate function must be created first:

```
function y = myfun(x)
y = cos(exp(x)) + x.^2 -1;
```

Minimizing a function

- `fminbnd` minimizes a function over a bounded interval:

  `>> x = fminbnd('myfun', -1, 2);` finds the local minimum of `myfun` for the interval $-1 \leq x \leq 2$

  `myfun(x)` should take a scalar input and return a scalar output.

- `fminsearch` minimizes a function with an unconstrained interval:

  `>> x = fminsearch('myfun', 0.5);` finds the local minimum of `myfun` starting at $x = -0.5$

- Maximizing a function:
  - MATLAB does not have `fmaxbnd` or `fmaxsearch`
  - To maximize $g(x)$, minimize the negative: $f(x) = -g(x)$
  - Use: `>> fminbnd(@(x) -g(x), a, b);`

- These solvers find **local minima**, not necessarily the global minimum!

Anonymous functions

- Instead of creating a separate function like `myfun(x)`, we can utilize an **anonymous function**:

```
>> x = fzero(@( x ) (cos(exp(x)) + x.^2 -1) , 1);
```

input

function to evaluate

```
>> x = fminbnd(@(x) (cos(exp(x)) + x.^2 -1), -1, 2);
```

- MATLAB can also store the function handle:

```
>> func = @(x) (cos(exp(x)) + x.^2 -1);
>> func(1:10);
```

Example 3 (Min-finding)

Goal: Use MATLAB to find and verify the minimum of a nonlinear function.

- Define the function:

$$f(x) = \cos(4x) \cdot \sin(10x) \cdot e^{-|x|}$$

- Use `fminbnd` to find the minimum of $f(x)$ in the interval $x \in [-\pi, \pi]$, call it `x_min`
- Plot the function over the interval to verify the result, using the given code:

```
x = linspace(-pi, pi, 500);
plot(x, f(x)); hold on;
plot(x_min, f(x_min), 'ro', 'MarkerFaceColor', 'r');
legend('f(x)', 'Minimum');
xlabel('x'); ylabel('f(x)');
title('Local minimum of f(x) on [-\pi, \pi]');
```

Numerical Issues

- Numerical computations rely on approximations.
- Many MATLAB functions use floating-point numbers.
- This means results are approximations!

Examples:

Constants like `pi` *are approximate:*

```
>> sin(pi)
>> sin(2*pi)
>> sin(1e16*pi)
```

Ill-Conditioned Matrices:

```
>> A = (1e13)*ones(10) + rand(10);
>> cond(A)   % large condition number
>> inv(A)*A   % not exactly identity!
```

`A` is nearly singular $\Rightarrow$ numerical instability

A word of caution

- MATLAB **will** always return an answer – even if it's wrong
- It optimizes, differentiates, integrates:
  - That's powerful – but not always reliable
- Don't overtrust the output!
  - You may get an answer that looks correct,
  - But it might be a result of numerical errors or bad assumptions
- When in doubt: try symbolic math or verify analytically
  - Consider the **Symbolic Math Toolbox**

Outline

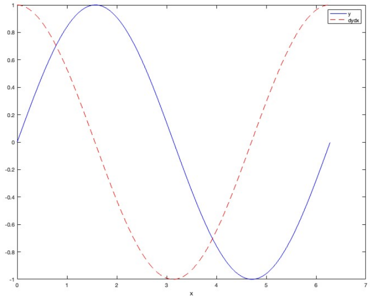
Lecture 3: Solving Equations, Curve Fitting & Numerical Techniques

- Linear Algebra
- Polynomials
- Optimization
- Differentiation & Integration
- Differential Equations

Numerical differentiation

- MATLAB can differentiate numerically:

```
x = 0:0.01:2*pi;
y = sin(x);
dydx = diff(y)./diff(x);
```



- It can also operate on matrices:

```
mat = [1 3 5; 4 8 6];
dm = diff(mat, 1, 2);
% dm = [2 2; 4 -2] - first difference along 2nd dimension
```

- The opposite of `diff` is the cumulative sum `cumsum`. For 2D gradient, use
  `[dx, dy] = gradient(mat);`

Numerical integration

MATLAB offers a variety of common numerical integration methods:

- Adaptive Simpson's quadrature (input is a **function**):

`q1 = quad('myFun',0,10);`	integral of the function `myFun` from 0 to 10
`q2 = quad(@(x) sin(x).*x,0,pi);`	integral of the $x \sin x$ from 0 to π

- Trapezoidal rule (input is a **vector**):

`x = 0:0.01:pi;`	define `x` vector
`z1 = trapz(x,sin(x));`	integral of $\sin x$ from 0 to π
`z2 = trapz(x,sqrt(exp(x))./x);`	integral of $\frac{\sqrt{e^x}}{x}$ from 0 to π

Outline

Lecture 3: Solving Equations, Curve Fitting & Numerical Techniques

- Linear Algebra
- Polynomials
- Optimization
- Differentiation & Integration
- Differential Equations

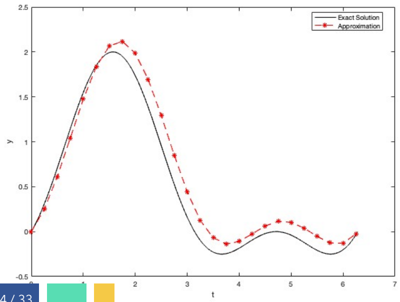
Ordinary Differential Equations (ODEs)

A differential equation can be solved by approximating its integral step-by-step.

- Given a differential equation $\frac{dy}{dt} = f(t, y)$
- Evaluate the slope (derivative) at each step, then update using the Euler's method (for example):

$$y_{n+1} = y_n + h \cdot f(t_n, y_n)$$

- This gives a piecewise linear approximation of the solution



- Errors accumulate with each step
- Smaller or adaptive timesteps improve accuracy

ODE Solvers

MATLAB provides implementations of widely used ODE solvers.

Choosing the appropriate ODE solver can significantly reduce computation time and yield more precise results.

ODE solver	Type	When to use
`ode23`	Low-order	When integrating over small intervals or when accuracy is less important than speed
`ode45`	High-order (Runge-Kutta)	High accuracy and reasonable speed. Most commonly used
`ode15s`	Stiff solver (Gear's algorithm)	When ODEs have time constants that vary by orders of magnitude

Standard syntax

```
>> [t,y] =  ode45 ( 'myODE' , [0,10] , [1;0] )
```

ODE solver:
23, 45, 15s

ODE function

time range

initial condition

Standard syntax

```
>> [t,y] =  ode45 ( 'myODE' , [0,10] , [1;0] )
```

ODE solver: 23, 45, 15s

ODE function

time range

initial condition

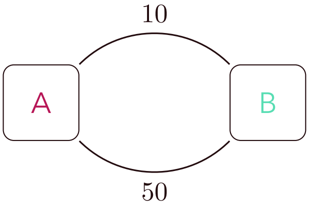
- **Inputs:**
  - ODE function or anonymous function should take inputs (t, y) and return dy/dt
  - Time interval is a 2-element vector with initial and final time
  - Initial condition is a column vectoir with an initial condition for each ODE. This is the first input passed to the ODE function
  - Ensure that all inputs are in the same (variable) order
- **Outputs:**
  - `t` contains the time points
  - `y` holds the corresponding values of the variables

ODE function

The ODE function should return the value of the derivative at a specific time and function value.

For example, take a chemical reaction:

$$\frac{dA}{dt} = -10A + 50B$$
$$\frac{dB}{dt} = 10A - 50B$$




```
% chem: chemical reaction ODE function
function dydt = chem(t,y)
dydt = zeros(2,1);
dydt(1) = -10*y(1) + 50*y(2);
dydt(2) = 10*y(1) - 50*y(2);
```

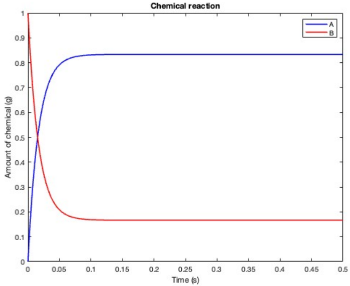
Here, `y` holds values `[A;B]` and `dydt` has `[dA/dt; dB/dt]`

Viewing results

To solve and plot the chemical ODE,

```
[t, y] = ode45('chem', [0 0.5], [0 1]);
% y0=[0 1] means only chemical B exists initially

plot(t, y(:,1), 'b', 'LineWidth', 1.5);
hold on;
plot(t, y(:,2), 'r', 'LineWidth', 1.5);
legend('A', 'B');
xlabel('Time (s)');
ylabel('Amount of chemical (g)');
title('Chemical reaction');
```



Higher order equations

- Higher order ODEs must be converted into a system of first-order equations to utilize ODE solvers.
- Nonlinear functions are acceptable.

Consider the pendulum example:

$$\ddot{\theta} + \frac{g}{L}\sin\theta = 0 \quad \Rightarrow \quad \ddot{\theta} = -\frac{g}{L}\sin\theta$$

$$\text{let} \quad \dot{\theta} = \gamma, \quad \text{then} \quad \dot{\gamma} = -\frac{g}{L}\sin\theta$$

Thus, we can write

$$\boxed{\vec{x} = \begin{bmatrix}\theta \\ \gamma\end{bmatrix}, \quad \frac{d\vec{x}}{dt} = \begin{bmatrix}\dot{\theta} \\ \dot{\gamma}\end{bmatrix}}$$

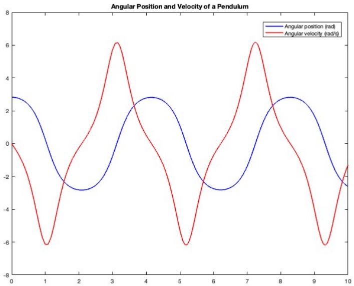
```
% pendulum
function dxdt = pendulum(t,x)
L = 1;
theta = x(1); gamma = x(2);

dtheta = gamma;
dgamma = -(9.8/L)*sin(theta);

dxdt = zeros(2,1);
dxdt(1) = dtheta;
dxdt(2) = dgamma;
```

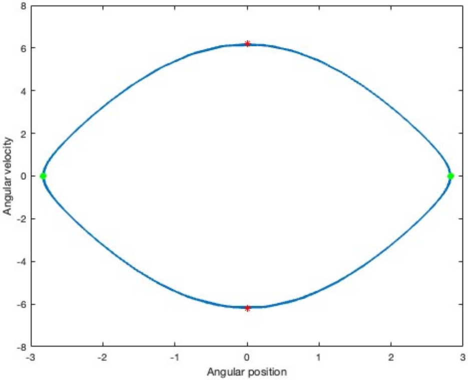
The following code solves for the position and velocity of the pendulum, and plots the outputs.

```
[t, x] = ode45('pendulum', [0 10], [0.9*pi 0]);
% assuming the pendulum is initially almost horizontal
plot(t, x(:,1), 'b', 'LineWidth', 1.5);
hold on; plot(t, x(:,2), 'r', 'LineWidth', 1.5);
legend('Angular position (rad)', 'Angular velocity (rad/s)', 'Location', ...
    'best');
title('Angular Position and Velocity of a Pendulum');
```



We can also plot in **phase plane**:

```
plot(x(:,1), x(:,2), 'LineWidth', 2);
xlabel('Angular position');
ylabel('Angular velocity');
```



- * Velocity $\dot{\theta} = 0$ when θ is the greatest.
- ∘ Velocity $\dot{\theta}$ is greatest when $\theta = 0$.

Custom options

MATLAB's ODE solvers use a variable timestep, but sometimes a fixed timestep is preferred:

```
[t,y] = ode45('chem', [0:0.001:0.5], [0 1]);
```

- Specify timestep by giving a vector of (increasing) times
- The function value will be returned at the specified points

You can customize the **error tolerances** using `odeset`:

```
options = odeset('RelTol', 1e-6, 'AbsTol', 1e-10);
[t,y] = ode45('chem', [0 0.5], [0 1], options);
```

- This ensures that the error at each step is less than `RelTol` times the value at that step and less than `AbsTol`
- Decreasing the error tolerance can significantly slow down the solver
- See `doc odeset` for a list of options you can customize

Example 4 (ODE)

Goal: Solve the first-order differential equation

$$\frac{dy}{dt} = -\frac{ty}{10}, \quad \text{with initial condition } \; y(0) = 10$$

on the interval $t \in [0, 10]$, and plot the solution using `ode45`.

Hint: Since the equation is simple, we can define the ODE using the anonymous function: `>> f = @(t, y) -t * y / 10;`

Bonus: Plot the result against the exact (analytical) solution.