A crash course in Matlab

Pablo Garcia

Fall, 2023

What is Matlab?

- **MATLAB:** Short for **MAT**rix **LAB**oratory.
- A high-performance numerical computing environment.
- Key Features:
  - Matrix-based language for efficient mathematical operations.
  - Powerful tools for data analysis, visualization, and algorithm development.
  - Extensive library of pre-built functions for various domains.
- Widely used in engineering, science, finance, and academia.
- Designed for ease of use.

Matrices and Arrays

Matrices and Arrays

- Matrices and arrays are fundamental data structures.
- `A = [1, 2, 3; 4, 5, 6; 7, 8, 9];` creates a 3x3 matrix.
- `B = zeros(2, 3);` creates a 2x3 matrix of zeros.
- `C = ones(4, 2);` creates a 4x2 matrix of ones.
- `D = eye(3);` creates a 3x3 identity matrix.

Matrices and Arrays

- Concatenate matrices:
  - Horizontal concatenation: `D = [A, B];`
  - Vertical concatenation: `E = [A; C];`
- Access elements using indices:
  - Single element: `A(2, 3)` gives the element in the second row and third column.
  - Entire row: `row_2 = A(2, :);`
  - Entire column: `col_3 = A(:, 3);`

Examples of Matrix Operations

- **Example 1: Creating a Matrix**
  - `A = [1, 2, 3; 4, 5, 6; 7, 8, 9];`
- **Example 2: Matrix Multiplication**
  - `B = A * C;`
- **Example 3: Element-wise Operations**
  - `D = A .* E;`
- **Example 4: Transposition**
  - `F = A';` or `F = transpose(A);`
- **Example 5: Addition**
  - `C = A + B;` and `D = A + 5;`

Calling bult-in functions

Calling Functions

- MATLAB provides built-in functions for various operations.
- Common functions include:
  - `[max_value, max_index] = max(matrix)`: Returns the maximum element and its index in the matrix.
  - `[min_value, min_index] = min(matrix)`: Returns the minimum element and its index in the matrix.
  - `mean(matrix, 1)`: Calculates the mean along each column of the matrix.
  - `std(matrix, 0, 2)`: Computes the standard deviation along each row of the matrix.
  - `rand(m, n)`: Creates a matrix of random values uniformly distributed in the interval (0, 1).
  - `randn(m, n)`: Generates a matrix of random values from a standard normal distribution.

Solving Linear Systems

Solving Linear Systems

- Consider the system of linear equations:

$$2x + 3y = 8$$
$$4x - 2y = 6$$

- To solve for x and y, use the $\backslash$ operator:

```
A = [2 3; 4 -2];
B = [8; 6];
X = A \ B;
```

- The solution is stored in matrix X.
- Less efficient way: $X = inv(A) * B$;

Finding Eigen-things

Eigenvalues and Eigenvectors

Definition: For a square matrix A, a scalar λ is an eigenvalue and a non-zero vector $\mathbf{v}$ is an eigenvector if $A\mathbf{v} = \lambda\mathbf{v}$.

- **Eigenvalues:** Computed using `eig(A)`.
- **Eigenvectors:** Computed using `[V, D] = eig(A)`.
- Example:

```
A = [2 1; 1 3];
[V, D] = eig(A);

eigenvalues = diag(D);
eigenvector1 = V(:, 1);
eigenvector2 = V(:, 2);
```

- Check that $A * V(:, 1)$ is (almost) identical to $D(1, 1) * V(:, 1)$

Generating Random Numbers

Normal Distribution and Histogram

```
% Generate 10,000 random samples from a normal distribution
X = normrnd(0,1,[100000,1]);

% Plot histogram
figure;
histogram(X, 'Normalization', 'probability');
title('Histogram of random samples from a normal distribution');
xlabel('Value');
ylabel('Probability');
```

- **Other Distributions:** You can also draw random numbers from other common distributions, e.g., `rand(100000,1)` for a uniform distribution and `exprnd(1,[100000,1])` for an exponential distribution.

Optimisation

Optimization in MATLAB

- MATLAB provides a rich set of built-in functions for optimisation.
- These functions cater to various types of optimisation problems; e.g. constrained/unconstrained problems, univariate/multivariate problems...
- MATLAB's optimisation functions are carefully implemented and optimised. Use them whenever possible; chances are that your own optimisation routines are not as efficient.

Univariate Optimization: `fminbnd`

- The `fminbnd` function is used for univariate minimization on a bounded interval.
- Example: Minimize $f(x) = x^2 + x$ on the interval $[a, b]$.

```
% Define the objective function
f = @(x) x^2 + x;

% Set the interval [a, b]
a = -2;
b = 2;

% Use fminbnd for optimization
x_m = fminbnd(f, a, b);

% Display the result
fprintf('Minimum value found at x = %.4f\n', x_m)
```

- Check that x_m satisfies $f'(x) = 0$.

`fminsearch`: Unconstrained Optimization

- `fminsearch` performs unconstrained minimization of a scalar function of one or more variables.
- Example: Minimize $f(x) = x^2 + x$.

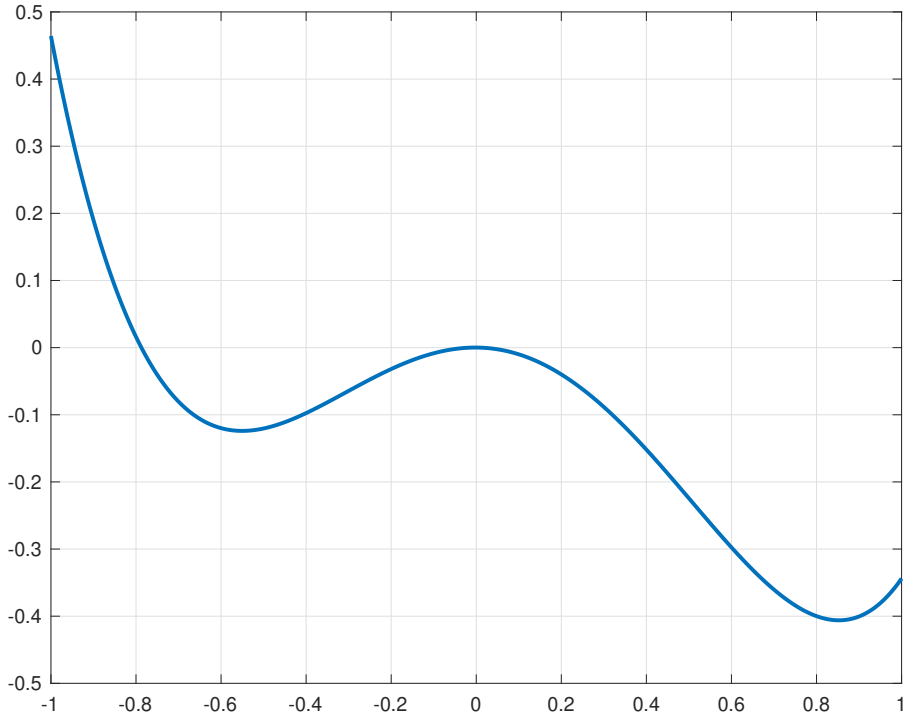
```
% Define the function
f = @(x) x.^2 + x;

% Use fminsearch for optimization
x_m = fminsearch(@(x) f(x), 0);

% Display the minimum found
fprintf('Minimum found at x = %.4f\n', x_m);
```

`fminsearch`: Local Minima

- `fminsearch` might not find the global minimum if the function has multiple minima.
- Consider $f = x^4 - x^2$
- Running x_{min} = fminsearch(@(x) f(x), -8) yields $x_{\text{min}} = -0.7071$
- But running x_{min} = fminsearch(@(x) f(x), 8) yields $x_{\text{min}} = 0.7071$
- The function has two local minima, and the initial guess determines which one you land in.
- This problem occurs with all optimization algorithms.
- Take-away: You must understand the mathematical structure of your problem before coding it. The computer will always provide results; you must be able to judge whether they are valid.

Grid Search for Initial Condition

- We've just seen that finding a good initial condition is crucial for optimisation.
- A simple grid search can help identify a reasonable starting point.
- Consider $f(x) = (x - 0.1)^4 - x^2$



Grid Search for Initial Condition

- We want to make sure that we don't get stuck at the local minima around -0.6
- A simple grid search can help identify a reasonable starting point.
- Define a grid of values: $x_{\text{grid}} = [-1 : 0.1 : 1]$
- Evaluate the objective function: $f_{\text{grid}} = f(x_{\text{grid}})$
- Select the initial guess with the lowest f_{grid} as the starting point for the optimisation algorithm.

Grid Search: Example

```
% Grid search for initial condition
x_grid = linspace(-1, 1, 100);
f_values = (x_grid - 0.1).^4 - x_grid.^2;
[min_value, min_index] = min(f_values);
in_con = x_grid(min_index);

% Use  initial_condition in optimisation algorithm
result = fminsearch(@(x) (x-0.1)^4 - x^2, in_con);
```

- **Note:** For this simple example, a grid search is not needed, but it becomes extremely useful for more complex problems, especially in higher dimensions.

`fmincon`: Constrained Optimization

- `fmincon` is designed for constrained optimisation problems, including both equality and inequality constraints.
- Check the Matlab website for a detailed description and many examples
- From personal experience: understand the structure of your problem when using `fmincon`!
- Back to previous example, but adding a nonlinear constraint:
- **Objective Function:** $f(x) = (x - 0.1)^4 - x^2$
- **Nonlinear constraint:** $x + 0.5 \leq 0$
- **Optimization Goal:** Minimize $f(x)$ subject to the given constraints.
- Note that the global minimum 0.8 does not satisfy the constraint. Hopefully, `fmincon` will guide us to the local minima.

fmincon: Example

```
% Define the objective function
 objective = @(x) (x-0.1)^4 - x^2;

% Set up the options and initial guess
options = optimoptions('fmincon', 'Display', 'iter');
x0 = 0;

% Run fmincon
x_mc = fmincon(objective, x0,1,-0.5,[],[],[],[],[],options);
```

- As expected, `fmincon` guides us to the local minima: $x_{mc} = -0.55$.
- This is a very simple example, `fmincon` can address all types of linear and nonlinear constraints.

Derivatives and Integrals

Computing Integrals in MATLAB

- MATLAB provides the `integral` function for numerical integration.
- Syntax: `q = integral(fun, xmin, xmax)`

```
% Example: Integrate the function
% f(x) = x^2 over [0, 1]
fun = @(x) x.^2;
xmin = 0;
xmax = 1;
q = integral(fun, xmin, xmax);
```

- The result `q` is an approximation of the integral of $f(x)$ over the specified interval.
- Check that the numerical results is (almost) identical to the analytical one: $1/3$.

Forward Derivative

- **Forward Derivative:** The forward derivative at a point x is approximated by:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

  where h is a small interval in the positive direction.
- `diff` **Function:** computes forward differences.
- Note: the central derivative is often more accurate:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Forward Derivative: Example

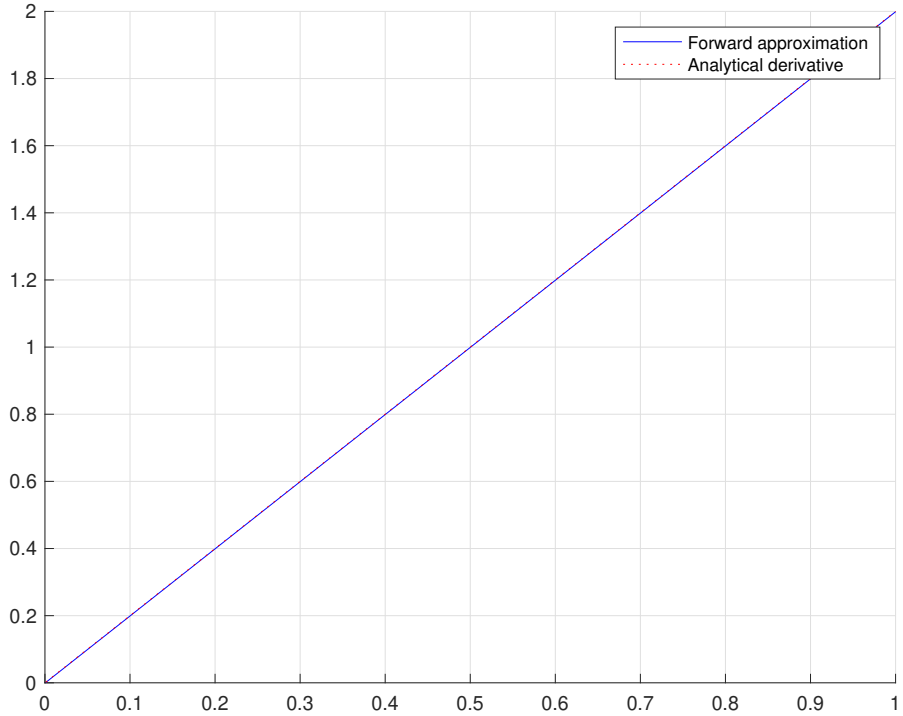
- Suppose we have a function $f(x) = x^2$. We can use `diff` to approximate the derivative at each point.

```
% Define domain and function
x = linspace(0, 1, 100);
y = x.^2;
h = diff(x);

% Compute forward differences
dy_dx_approx = diff(y) ./ h;
```

- Check that the result is (almost) identical to $f'(x) = 2x$

Forward Derivative: Example

- With $h = 0.001$, the approximation is great.



- Check that as h becomes larger, the approximation deteriorates.

A Piece of Advice

- Numerical integration and differentiation serve as valuable tools to validate analytical computations.
- When I have to analytically integrate or differentiate an ugly function $F(x)$, I often do so numerically and cross-check with my analytical solution.
- If numerical and analytical results don't align, it always means I've done a mistake somewhere in my analytical computations. (So far, MATLAB has never been wrong.)

Loops

Loops in MATLAB

- Loops are structures that allow the repetition of a set of instructions. They are used to execute a block of code multiple times.
- Loops:
  - Efficiently handle repetitive tasks.
  - Reduce code redundancy.
- Common types of loops:
  - `for` loop: Executes a block of code a specific number of times.
  - `while` loop: Repeats a block of code as long as a specified condition is true.
- Key Concepts:
  - **Iteration:** Each execution of the loop is called an iteration.
  - **Loop Control:** Conditions control when a loop starts and stops.

The `for` Loop in MATLAB

- The `for` loop repeats a set of statements a fixed number of times.
- Basic Syntax:

```
for variable = range
    % Code to be executed in each iteration
end
```

- Example:

```
for i = 1:5
    disp(['Iteration ', num2str(i)]);
end
```

- This loop will iterate five times, displaying the iteration number in each run.

Example: Forward Difference Using `for` Loop

```
% Define the function
f = @(x) x.^2;

% Set up parameters
x_values = linspace(0, 5, 10000);
h = x_values(2)-x_values(1);

% Initialize an array to store differences
forward_diff = zeros(size(x_values)-1);

% Compute forward differences
for i = 2:length(x_values)
    forward_diff(i-1) = (f(x_values(i)) - f(x_values(i-1))) / h;
end

% Plot the results
figure;
plot(x_values(2:end), forward_diff, 'b', 'LineWidth', 1.5);
title('Forward Difference of x^2');
```

Note: Using loops for operations that can be 'vectorised' is not efficient.

While Loop

- The while loop repeatedly executes a block of code as long as a specified condition is true.

```
% Example of a while loop
counter = 1;
while counter <= 5
    disp(['Iteration ', num2str(counter)]);
    counter = counter + 1;
end
```

- This loop will iterate five times, printing the current iteration number in each iteration.

A real example (only for those interested, not needed for the course)

Setup

- Consider the following equation

$$\frac{\partial V(t,x)}{\partial t} = \frac{\partial^2 V(t,x)}{\partial x^2},$$

subject to the initial condition

$$f(0,x) = \begin{cases} 2x & \text{if } 0 < x < 0.5 \\ 2(1-x) & \text{if } 0.5 < x < 1, \end{cases}$$

and the boundary condition

$$V(t,0) = V(t,1) = 0.$$

- This is actually the heat equation. But solving many macroeconomic models in continuous time boil down to solving slightly more difficult versions of it. The most famous example is the Black-Scholes equation. Heterogeneous agent models are also common examples.

Solution

- Create a grid for t and for x. Let dt and dx the distance between grid points along each dimension.
- Approximating the derivative with respect to t by a forward approximation and the second derivative with respect to x by

$$\frac{\partial^2 V(t,x)}{\partial x^2} = \frac{V(t,x+dx) - 2V(t,x) + V(t,x-dx)}{(dx^2)}$$

yields:

$$\frac{V(t+dt,x)}{dt} = \frac{V(t,x+dx) - 2V(t,x) + V(t,x-dx)}{dx^2}.$$

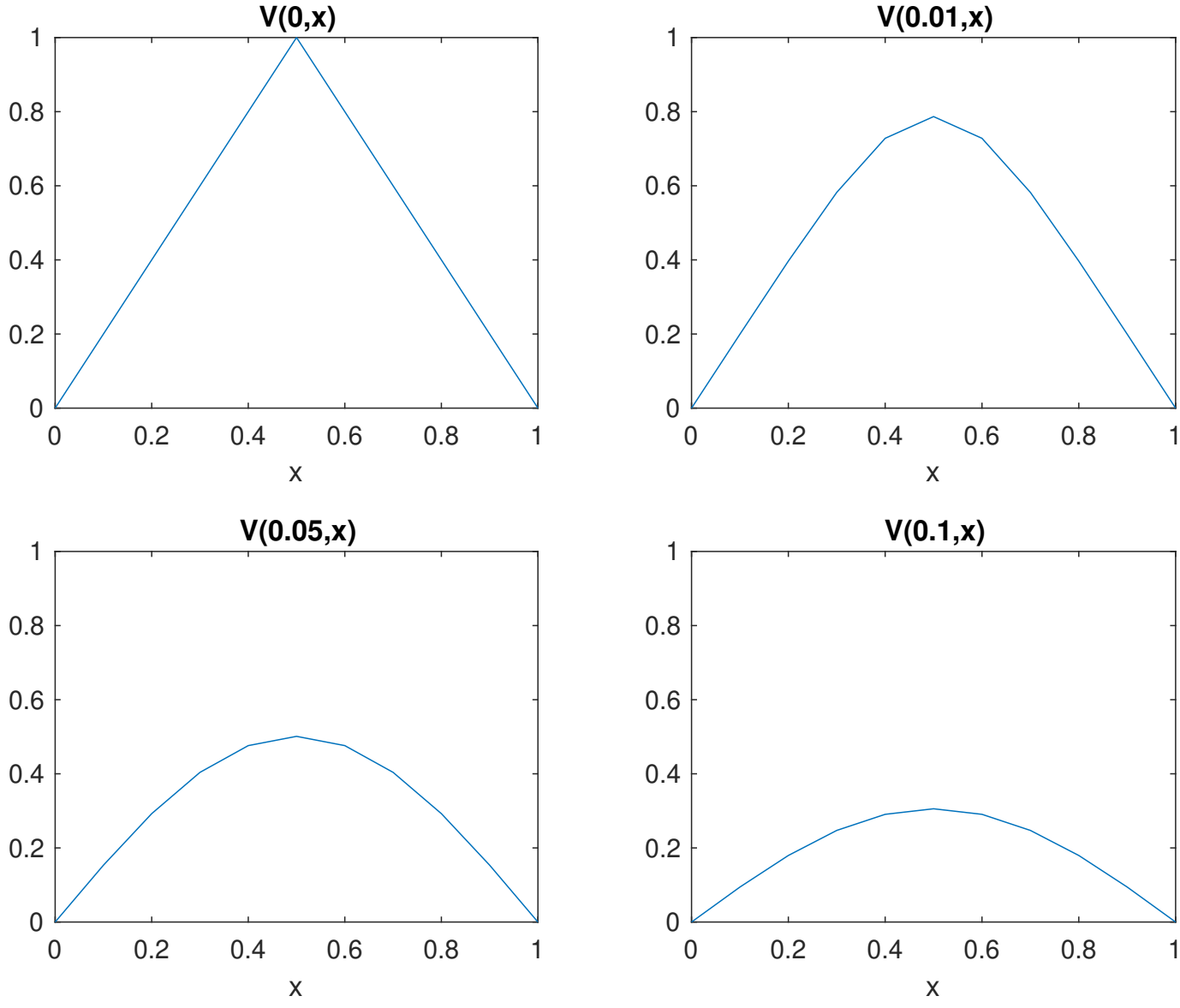
Solution

- Rearranging terms:

$$V(t+dt,x) = \rho V(t,x-dx)+(1-2\rho)V(t,x)+\rho V(t,x+dx),$$

  where $\rho = \frac{dt}{dx^2}$.
- Starting from the initial condition $t = 0$, we simply solve the above equation for increasing values of $t = 1, 2, ...$
- This method is known as finite differences. Any good reference in partial differential equations will provide a detailed description.
- The next slides provide the code.
- In sum, tackling complex problems doesn't always require being an expert in Matlab; patience and creativity can take you a long way.

Numerical solution of the heat equation



Heat Equation Solution - Part 1

```
% Inputs
dx  = 0.1;
dt  = 0.001;
T   = dt*100;
X   = 1;
rho = dt/(dx^2);

% Grid
x_gr = 0:dx:X;
t_gr = 0:dt:T;

% Value function
v  = zeros(numel(x_gr),numel(t_gr));
```

Heat Equation Solution - Part 2

```
% Initial conditions
for i = 1:numel(x_gr)
    x = x_gr(i);
    if x < 0.5
        v(i,1) = 2*x;
    else
        v(i,1) = 2*(1-x);
    end
end

% Explicit finite differences
for j = 1:numel(t_gr)-1
    for i = 2:numel(x_gr)-1
        v(i,j+1) = rho * v(i-1,j) + (1-2*rho)*v(i,j) ...
                   + rho * v(i+1,j);
    end
end
```

Heat Equation Solution - Part 3

```
% Plot function at different times
figure;
subplot (2,2,1) ;
    plot(x_gr,v(:,1))
    title('V(0,x)')
    axis([0 1 0 1])
subplot(2,2,2);
    plot(x_gr,v(:,11))
      title('V(0.01,x)')
    axis([0 1 0 1])
subplot(2,2,3);
    plot(x_gr,v(:,51))
      title('V(0.05,x)')
    axis([0 1 0 1])
subplot(2,2,4);
    plot(x_gr,v(:,101))
      title('V(0.1,x)')
    axis([0 1 0 1])
```