MATLAB Introduction Course: Lecture 3

Øivind K. Kjerstad

10. October 2014

TOC

Solving Linear Equations

- Given a system of linear equations

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

- Construct matrices so the system is described by $Ax = b$
  - `>> A=[1 2 -3;-3 -1 1;1 -1 1]`
  - `>> b=[5; -8; 0]`
- Solve with a single line of code!
  - `>> x = A\b`
  - `x` is a 3x1 vector containing the values of x, y, and z
- The `\` will work with square or rectangular systems
- Gives least squares solution for rectangular systems. Solution depends on whether the system is over or underdetermined

Solving Linear Equations

- Given a matrix
  - `>> mat=[1 2 -3;-3 -1 1;1 -1 1]`
- The rank of a matrix
  - `rank(mat)`
  - The number of linearly independent rows or columns
- The determinant
  - `det(mat)`
  - `mat` must be square
  - If the determinant is nonzero, then the matrix is invertible
- The matrix inverse
  - `inv(mat)`
  - if an equation is of the form $Ax = b$ with A a square matrix, $x = A\backslash b$ is the same as `x=inv(A)*b`

Matrix Decompositions

- MATLAB has several built-in matrix decomposition methods
- The most common are:
  - `[V,D]=eig(X)`
    - Eigenvalue decomposition
  - `[U,S,V]=svd(X)`
    - Singular value decomposition
  - `[Q,R]=qr(X)`
    - QR decomposition
- For more information, see `help\doc`

Exercise 1: Solve Linear Systems

Exercise 1a

The linear system:

$$x + 4y = 34$$
$$-3x + y = 2$$

- Determine the rank of the problem
- Solve for x and y

Exercise 1b

The linear system:

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

- Determine the rank of the problem
- Solve for x and y
- Determine the least squares error

Solution

Exercise 1a

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

Exercise 1b

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

TOC

Polynomials

- Many functions can be well described by a high-order polynomial
- MATLAB represents a polynomials by a vector of coefficients
  - $p(x) = ax^3 + bx^2 + cx + d$
  - `P = [a b c d]`

Examples

$p(x) = x^2 - 2 \Rightarrow$ `P = [1 0 -2]`

$p(x) = 2x^3 \Rightarrow$ `P = [2 0 0 0]`

Polynomial Operations

- P is a vector of length N+1 describing an N-th order polynomial
- Polynomial roots
  - `>> r = roots(P)`
  - `r` is a vector of length N
- Polynomial from the roots
  - `>> P = poly(r)`
  - `r` is a vector of length N
- Evaluate a polynomial at a point
  - `>> y0 = polyval(P,x0)`
  - `x0` is a single value; `y0` is a single value
- Evaluate a polynomial at many points
  - `>> y = polyval(P,x)`
  - `x` is a vector; `y` is a vector

Polynomial Fitting

- MATLAB makes it very easy to fit polynomials to data
  - polyfit

Example

Given data vectors $X = \begin{bmatrix}-1 & 0 & 2\end{bmatrix}$ and $Y = \begin{bmatrix}0 & -1 & 3\end{bmatrix}$

```
p =  polyfit(X,Y,2)
plot(X,Y,'o', 'MarkerSize', 10);
hold on;
plot(-3:.01:3,polyval(p,-3:.01:3), 'r--');
```

This finds the best second order polynomial that fits the points (-1,0),(0,-1), and (2,3). See `doc polyfit` for more info

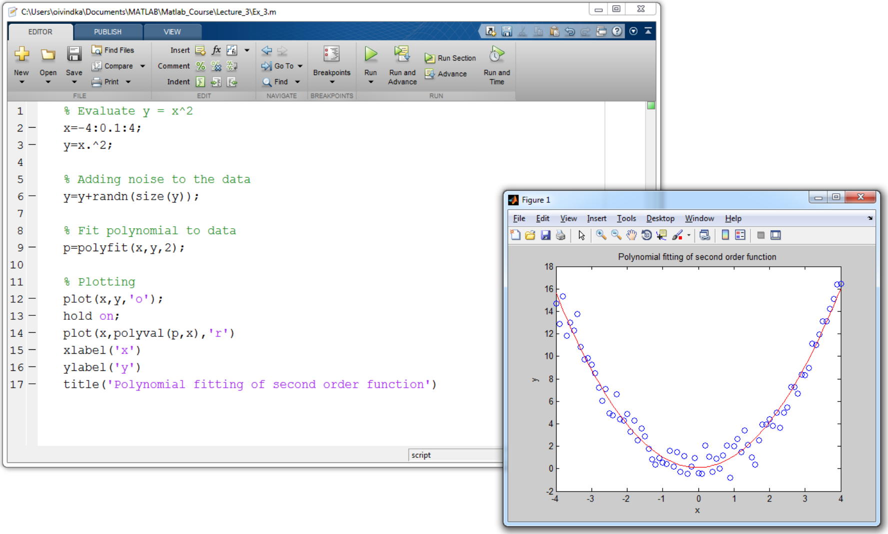
MATLAB has a toolbox for fitting data to expressions, see `cftool` and `splinetool`

Exercise 2: Polynomial Fitting

Polynomial Fitting

- Evaluate $y = x^2$ for `x=-4:0.1:4`
- Add random noise to `y`, use `randn`
- Fit a 2nd degree polynomial to the noisy data
- Plot the noisy data using circular markers and the fitted polynomial using a solid red line

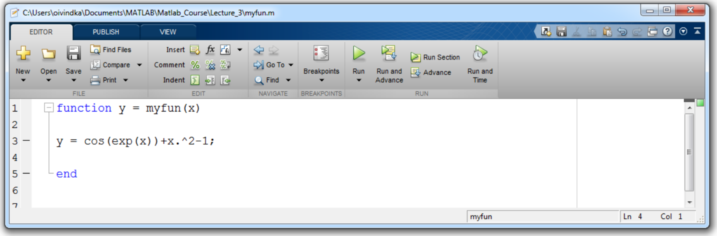
Solution


C:\Users\oivindka\Documents\MATLAB\Matlab_Course\Lecture_3\Ex_3.m
% Evaluate y = x^2
x=-4:0.1:4;
y=x.^2;
% Adding noise to the data
y=y+randn(size(y));
% Fit polynomial to data
p=polyfit(x,y,2);
% Plotting
plot(x,y,'o');
hold on;
plot(x,polyval(p,x),'r')
xlabel('x')
ylabel('y')
title('Polynomial fitting of second order function')
Figure 1
Polynomial fitting of second order function
x
y

TOC

Nonlinear Root Finding

- Many real-world problems require us to solve $f(x) = 0$
- Can use `fzero` to calculate roots for any arbitrary function
- `fzero` needs a function passed to it
  - We will see this type of operation more as we go into solving equations
  - `>> x=fzero(@myfun,x0)`



Minimizing a Function

- `fminbnd`: minimizing a function over a bounded interval
  - `>> x=fminbnd(@myfun,-1,2);`
    - ⋆ `myfun` takes a scalar input and returns a scalar output
    - ⋆ `myfun` find the minimum in the interval $-1 \leq x \leq 2$
- `fminsearch`: unconstrained interval
  - `>> x=fminsearch(@myfun,.5);`
    - ⋆ finds the local minimum of `myfun` starting at $x = 0.5$

Anonymous Functions

- What if `myfun` is relatively simple?
  - Then, using anonymous functions can be more efficient and simpler
  - In practice this is writing the function directly into the function call
- `>> x=fzero(@(input)(function expression),x0)`

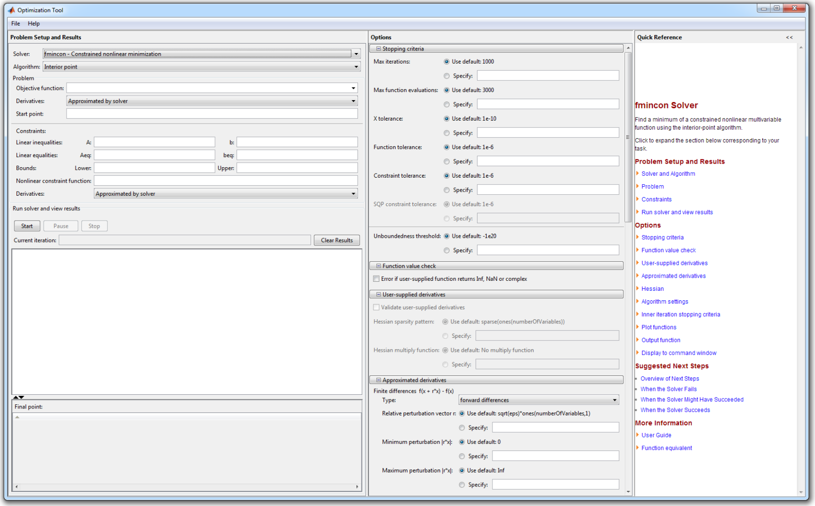
Examples

```
>> x=fzero(@(x)(cos(exp(x))+x^2-1),x0)
>> x=fminbnd(@(x)(cos(exp(x))+x^2-1),x_low,x_high);
>> x=fminsearch(@(x)(cos(exp(x))+x^2-1),x0);
```

Optimization Toolbox

- If you are familiar with optimization methods, use the optimization toolbox
  - Useful for larger, more structured optimization problems
  - It is located under MATLAB apps
- Sample functions (see `help`for more info)
  - `linprog`
    - Linear programming using interior point methods
  - `quadprog`
    - Quadratic programming solver
  - `fmincon`
    - Constrained nonlinear optimization

Optimization Toolbox

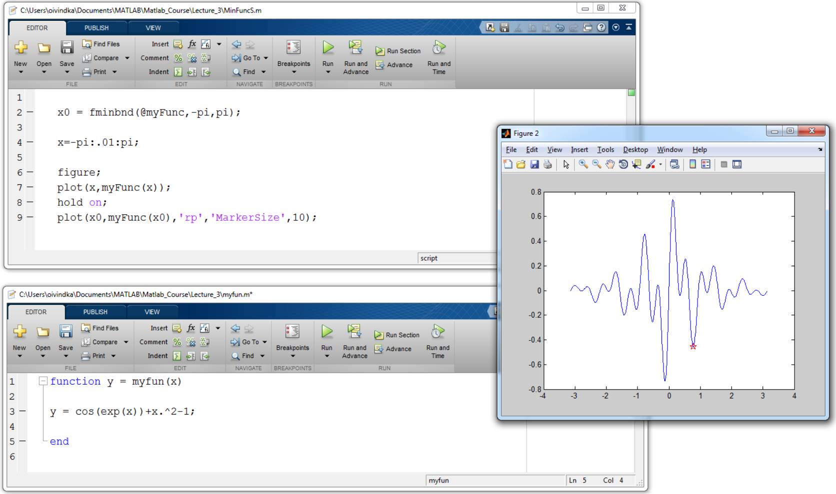


Exercise 3: Find Minimum

Exercise 3: Find Minimum

- Find the minimum of the function $f(x) = cos(x)sin(10x)e^{|x|}$ over the range $x \in [-\pi \; \pi]$ using `fminbnd`
- Plot the function for the given range
- Plot the found minimum solution in the same figure

Solution



Remember to check what built-in functions do!

TOC

Numerical Differentiation

- MATLAB can differentiate numerically using `diff`
  - `diff` computes the first difference
- `diff` also works on matrices
  - Computes the first difference along the 2nd dimension
  - The opposite of diff is the cumulative sum cumsum
  - See help for more details
- For the 2D gradient, see `gradient`

Example

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

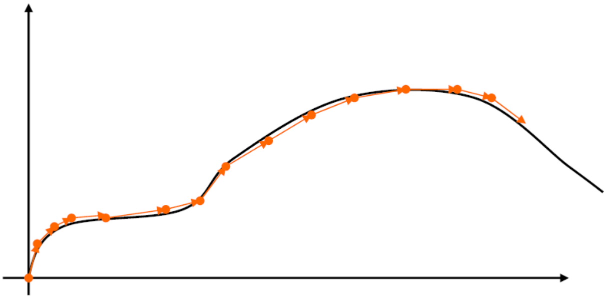
Numerical Integration

- MATLAB includes the most common integration methods
- Adaptive Simpson's quadrature (input is a function)
  - `q=quad('myFun',0,10);`
  - q is the integral of the function myFun from 0 to 10
  - `q2=quad(@(x) sin(x)*x,0,pi);`
  - q2 is the integral of $sin(x)x$ from 0 to π
- Trapezoidal rule (input is a vector)
  - `>> x=0:0.01:pi;`
  - `>> z=trapz(x,sin(x));`
  - z is the integral of sin(x) from 0 to π
  - `>>z2=trapz(x,sqrt(exp(x))./x);`
  - z2 is the integral of $\sqrt{\frac{e^x}{x}}$ from 0 to π

TOC

ODE Solvers: Method Overview

- Given a differential equation, the solution can be found by integration:



  - Evaluate the derivative at a point and approximate by straight line
  - Errors accumulate!
  - Variable timestep can decrease the number of iterations

ODE Solvers: MATLAB

- MATLAB contains implementations of common ODE solvers
- Using the correct ODE solver can save time and give more accurate results
  - `ode23`
    - ⋆ Low-order solver. Use when integrating over small intervals or when accuracy is less important than speed
  - `ode45`
    - ⋆ High order (Runge-Kutta) solver. High accuracy and reasonable speed. Most commonly used
  - `ode15s`
    - ⋆ Stiff ODE solver (Gear's algorithm), use when the diff eq's have time constants that vary by orders of magnitude

ODE Solvers: Standard Syntax

The ODE function call

```
[t,y]=ode45('myODE',[0,10],x0)
```

- ode45 is the solver
- 'myODE' is the function to be evaluated
- [0, 10] is the simulation time range
- x0 is the initial conditions

- Inputs
  - ODE function name (or anonymous function). This function takes inputs (t,y), and returns dy/dt
  - Time interval: 2-element vector specifying initial and final time
  - Initial conditions: column vector with an initial condition for each ODE. This is the first input to the ODE function
- Outputs
  - t contains the time points
  - y contains the corresponding values of the integrated variables.

ODE Example: The pendulum

Example: The pendulum equations

$$\ddot{\theta} + \frac{g}{L} sin(\theta) = 0$$

Define the state $x = [\theta \ \gamma]^\top$, where $\dot{\theta} = \gamma$. Then we can write the system as

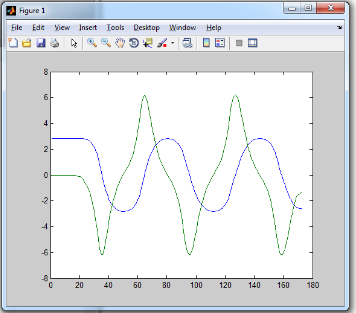
$$\dot{\theta} = \gamma$$

$$\dot{\gamma} = -\frac{g}{L} sin(\theta)$$

- Must make into a system of first-order equations to use ODE solvers
- Nonlinear equations are OK!

ODE Example: The pendulum

```matlab
function dxdt = pendulum(t,x)

dxdt = zeros(2,1);
L = 1;

theta = x(1);
gamma = x(2);

dt_theta = gamma;
dt_gamma = -(9.81/L)*sin(theta);

dxdt(1) = dt_theta;
dxdt(2) = dt_gamma;

end
```



Command Window

```matlab
>> [t,x]=ode45('pendulum',[0 10],[0.9*pi 0]);
>> plot(x)
>>
```

ODE Solvers: Custom Options

- MATLAB's ODE solvers use a variable timestep
- Sometimes a fixed timestep is desirable
  - `[t,y]=ode45('myODE',[0:0.001:0.5],x0);`
    - Specify the timestep by giving a vector of times
    - The function value will be returned at the specified points
    - Fixed timestep is usually slower because function values are interpolated to give values at the desired timepoints
- You can customize the error tolerances using `odeset`
  - `options=odeset('RelTol',1e-6,'AbsTol',1e-10);`
  - `[t,y]=ode45('myODE',[0 100],x0,options);`
    - This guarantees that the error at each step is less than RelToltimes the value at that step, and less than AbsTol
    - Decreasing error tolerance can considerably slow the solver
    - See doc odesetfor a list of options you can customize

Exercise 4: ODE

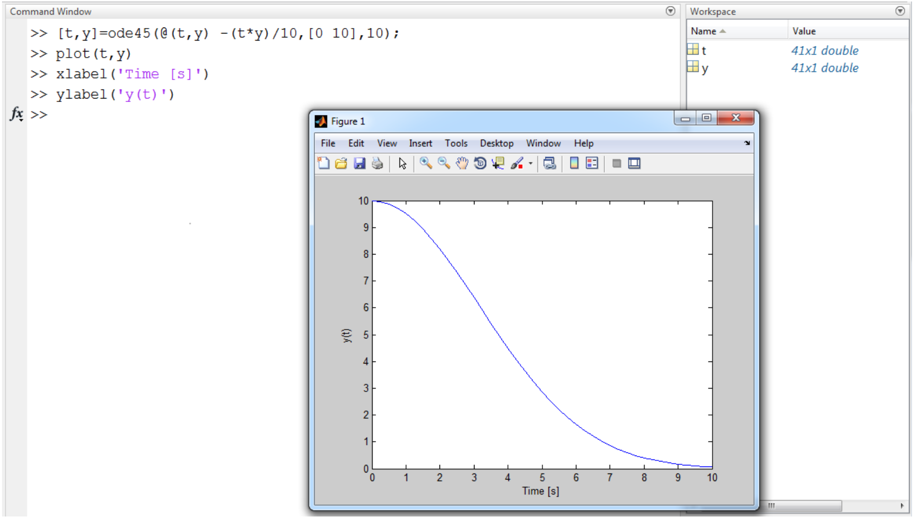
Exercise 4: ODE

- Implement the ODE:

$$\frac{dy}{dt} = -\frac{ty}{10}$$

- with initial condition $y(0) = 10$ over the interval $t = [0, 10]$
- Use `ode45`
- Plot $y(t)$

Exercise 4: Solution



`>> THE END`