

# Tutorial 3: Matlab Implementation and Examples of Nonlinear Systems

COMP5930M Scientific Computation

# Today

## Solving systems

Implementing Newton's method

Implementing the Jacobian

Matlab fsolve()

## Examples

## Further work

## Next

## Solving systems: Newton's Method

newtonSys.m help information:

```
% Basic Newton algorithm for systems of nonlinear equations
%
% function [ x,f ] = newtonSys( fnon, fjac, x0, tol, maxit )
%
% Input:
%     fnon - function handle for nonlinear system
%     fjac - function handle for Jacobian matrix
%     x0 - initial state (column vector)
%     tol - convergence tolerance
%     maxIt - maximum allowed number of iterations
%
% Output:
%     x - final point
%     f - final function value
```

## Solving systems: Newton's Method

newtonSys.m (basic algorithm):

```
function [x,f] = newtonSys(fnon,fjac,x0,tol,maxit)
```

```
n = length(x0);
```

```
x = x0;
```

```
f = feval( fnon, x );
```

```
it = 0;
```

```
while norm(f) > tol && it < maxit
```

```
    J = feval( fjac, n,x,f,fnon );
```

```
    delta = -J\f;
```

```
    x = x + delta;
```

```
    f = feval( fnon, x );
```

```
    it = it + 1;
```

```
end
```

## Solving systems: the Jacobian

fdJacobian.m:

```
function J = fdJacobian( n, x0, f0, fnon )
```

```
J = zeros(n);
```

```
h = 10*sqrt(eps);
```

```
for j = 1:n
```

```
    x = x0;
```

```
    x(j) = x(j) + h;
```

```
    f = feval( fnon, x );
```

```
    J(1:n,j) = ( f - f0 )/h;
```

```
end
```

## Solving systems: Matlab `fsolve()`

- ▶ Basic use is identical to `fzero()`
  - ▶ Basic: `x = fsolve( @(x)name(x,c),x0 )`
  - ▶ More output: `[x,f,flag]=fsolve(...)`
  - ▶ More detail: `fsolve(...,optimset('Display','iter'))`
- ▶ Internal computation of Jacobian
- ▶ Internal solution of `linear system`

## Example: 2-equation system

$$\begin{aligned}2x - y &= e^{-x} \\ -x + 2y &= e^{-y}\end{aligned}$$

In code: `exampleFun.m`

```
function y = exampleFun( x )  
  
y(1) = 2*x(1) - x(2) - exp(-x(1));  
y(2) = -x(1) + 2*x(2) - exp(-x(2));  
  
end
```

## Solving the system

- ▶ Both methods converge to  $(x, y) = (0.5671, 0.5671)$  from any  $x > 0, y > 0$  point
- ▶ `fsolve()` appears faster
  - ▶ but note function count - linesearch



## Analytical Jacobian

Simple to define for the problem here

$$\mathbf{J} = \begin{pmatrix} 2 + e^{-x} & -1 \\ -1 & 2 + e^{-y} \end{pmatrix} \quad 2$$

In code: `trueJacobian.m`

```
function J = trueJacobian( n, x, f, fnon)
```

```
J = [ 2+exp(-x(1), -1; ...  
      -1, 2+exp(-x(2)) ];
```

```
end
```

## Example: 2-equation system

$$-2x^2 + 3xy + 4\sin(y) = 6$$

$$3x^2 - 2xy^2 + 3\cos(x) = -4$$

In code: `example2.m`

```
function y = example2( x )
```

```
y = [ -2*x(1)^2+3*x(1)*x(2)+4*sin(x(2))-6;  
      3*x(1)^2-2*x(1)*x(2)^2+3*cos(x(1))+4 ];
```

```
end
```

## Solving the system

- ▶ The basic code diverges from almost any initial point 3
- ▶ `fsolve()` converges to  $(0.5798, 2.5462)$  from  $(0,0)$  and nearby points
- ▶ `fsolve()` converges to  $(2.59, 2.04)$  from other choices

## Further work

- ▶ Experiment with the examples above
  - explore the solution space
- ▶ The coursework will define some nonlinear problems to be formulated, implemented and solved numerically
- ▶ Use these examples as templates
  - very little implementation from scratch

## Next time...

- ▶ Our basic Newton algorithm struggles to converge for trivial problems even when initial guess is close to the solution
- ▶ There is a basic lack of robustness
- ▶ The addition of line search to the update step is critical

4