

MATLAB MATH

>> 1997

>> **matlab** conference

Cleve Moler

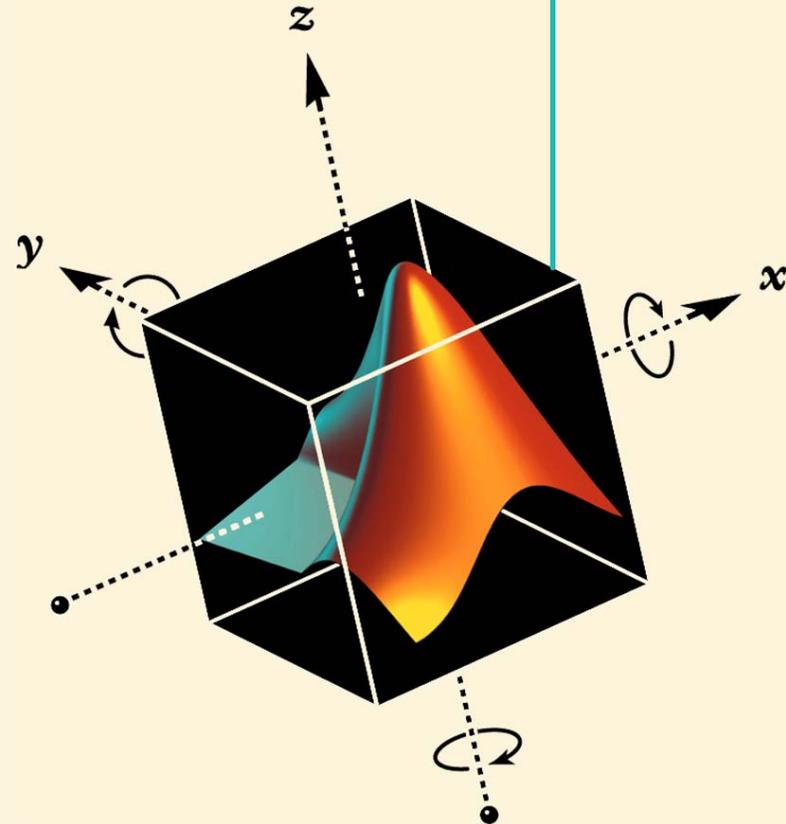
moler@mathworks.com

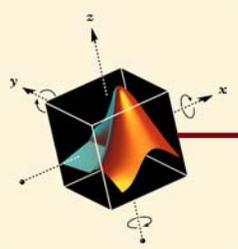
The MathWorks Inc.

24 Prime Park Way

Natick, MA 01760-1500

USA





MATLAB MATH

Floating point numbers

Matrix factorizations

Ordinary differential equations

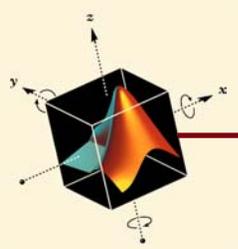
Zero finding

Fourier transforms

Random numbers

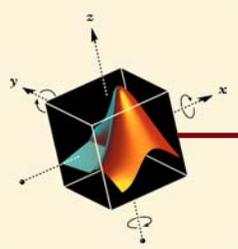
Sparse matrices

Symbolic computation



TAs

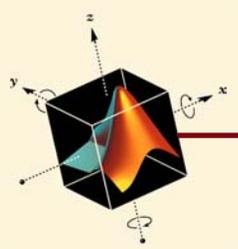
- ◆ Penny Anderson
- ◆ Mary Ann Branch



Important Note

These PowerPoint slides are NOT self contained.

Run MATLAB at the same time and enter anything you see in **blue.**

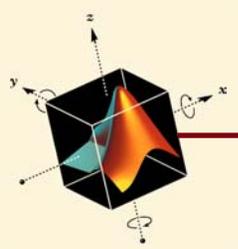


Questions

Please interrupt and ask questions.

“People who ask dumb questions are doing a public service”

-- Prof. Marc Kac



Floating Point Numbers

ANSI / IEEE Standard 754, double precision

$$x = \pm (1+f) \cdot 2^e$$

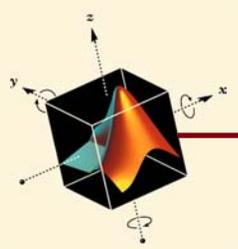
$$f = d_1/2 + \dots + d_{52}/2^{52}, \quad d_k = 0,1$$

$$-1022 \leq e \leq 1023$$

Roundoff: $\text{eps} = 2^{-52}$

Underflow: $\text{realmin} = 2^{-1022}$

Overflow: $\text{realmax} = (2-\text{eps}) \cdot 2^{1023}$



Floating Point Numbers

```
>> format hex
```

```
>> t = 1/10
```

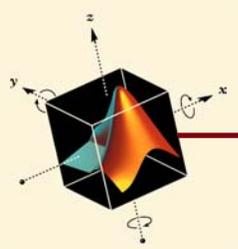
$$= 10^{-1}(1.0\dots)_{10}$$

$$= 2^{-4}(1.10011001100110011001\dots)_2$$

$$\approx 2^{-4}(1.999999999999999a)_{16}$$

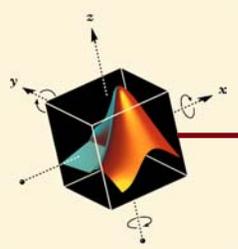
```
t =
```

```
3fb9999999999999a
```



Floating Point Numbers

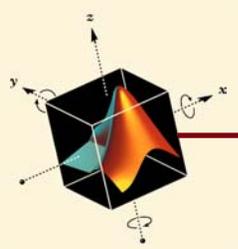
```
>> 1.0 - 0.8 - 0.2
>> x = [1.0 -0.8 -0.2]
>> format long, x
>> sum(x)
>> format hex, x
>> X = sym(x, 'e')
>> sum(X)
```



Exercise

First, do the following computation by hand.
Then, use MATLAB. Why do the results differ?
Where are the roundoff errors?

```
>> a = 4/3  
>> b = a - 1  
>> c = 3*b  
>> e = 1 - c
```



Test matrices

Magic squares

`magic(n)`

Hilbert matrices

`hilb(n)`

Pascal matrices

`pascal(n)`

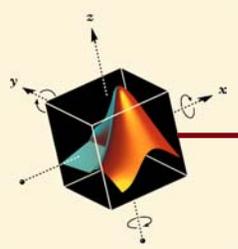
Random matrices

`rand(n)`, `randn(n)`

Higham's gallery

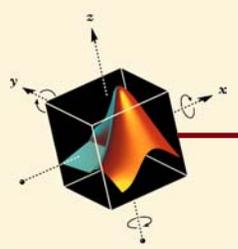
`help gallery`

`help private/xxx`



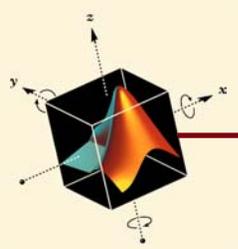
Exercise

Choose some test matrices for later use.



Matrix factorizations

- ◆ Triangular
 - ◆ `lu`, `chol`
- ◆ Orthogonal
 - ◆ `qr`
- ◆ Eigenvalue
 - ◆ `eig`, `qz`
- ◆ Singular value
 - ◆ `svd`

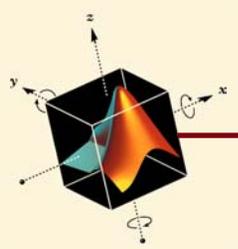


Triangular factorization

$$A = LU$$

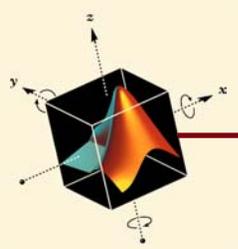
L permuted unit lower triangular

U upper triangular



Triangular factorization

- » $A = \text{test matrix}$
- » $[L,U] = \text{lu}(A)$



Exercise

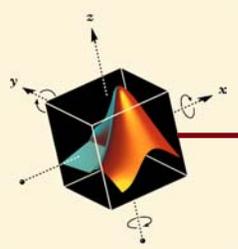
Find the LU factorization of your test matrices.

Observe the structure of L and U.

Verify that LU is close to A.

```
>> [L,U] = lu(A)
```

```
>> L*U - A
```

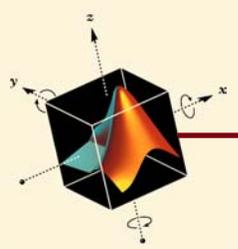


Cholesky factorization

If A is symmetric and positive definite

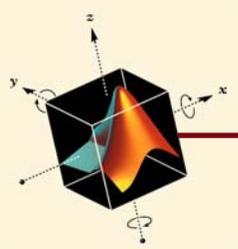
$$A = R^T R$$

R upper triangular



Cholesky factorization

- » $A = \textit{spd test matrix};$
- » $R = \text{chol}(A);$



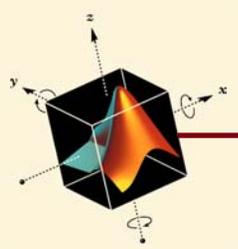
Exercise

Find a symmetric, positive definite test matrix.
Compute its Cholesky factor.

Find a symmetric, indefinite test matrix.
Try to compute its Cholesky factor.

```
>> help chol
```

```
>> [R,p] = chol(A)
```

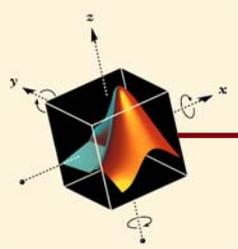


Orthogonal factorization

$$A = QR$$

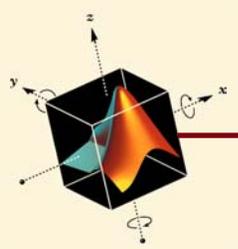
Q orthogonal

R upper triangular



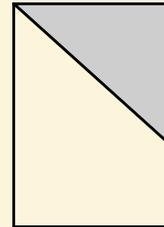
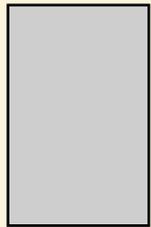
Orthogonal factorization

- » $A = \text{rectangular test matrix}$
- » $[Q,R] = \text{qr}(A)$

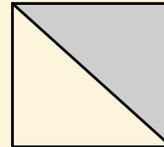
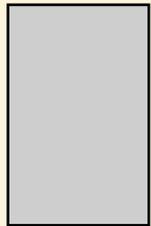


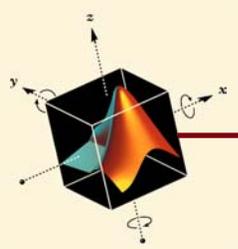
Orthogonal factorization

Full QR



Economy-sized QR





Exercise

Find the QR factorization of your test matrices.

Observe the structure of Q and R.

Verify that QR is close to A .

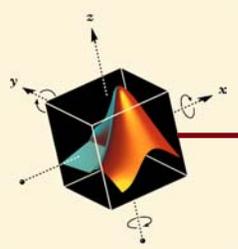
Verify that Q is close to orthogonal.

```
>> [Q,R] = qr(A)
```

```
>> Q*R - A
```

```
>> I = eye(size(A))
```

```
>> Q'*Q - I
```



Backslash

Problem:

$$\text{Solve } Ax = b$$

MATLAB:

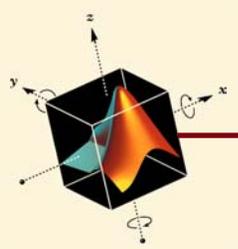
$$x = A \backslash b$$

Algorithm:

Triangular A -- solve directly.

Square A -- use Cholesky or LU.

Rectangular A -- use QR (least squares).



Existence and uniqueness

Problem:

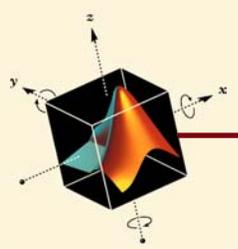
$$\text{Solve } Ax = b$$

Existence:

b is in the column space of A .

Uniqueness:

$Ax = 0$ has no nonzero solutions.



Existence and uniqueness

Example: 1-by-1

$$\text{Solve } ax = b$$

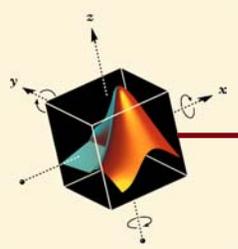
Solution

$$x = a \backslash b$$

What if b is zero?

What if a is zero?

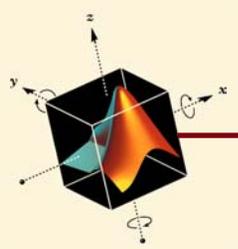
What if both are zero?



Moler's rules

It's very hard to compute things that don't exist.

It's also hard to compute things that aren't unique.



Condition number

Problem:

Change A to $A + \delta A$.

How much does $x = A \setminus b$ change?

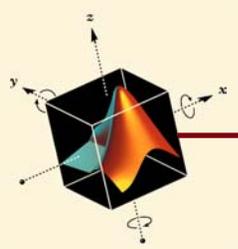
Answer (approximately):

$$\|\delta x\| / \|x\| \leq \kappa(A) \|\delta A\| / \|A\|$$

$\kappa(A)$ is the *condition number* of A .

Computed by `cond(A)`

Estimated by `condest(A)`

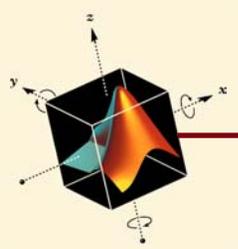


Condition number

Rule of thumb:

A matrix is *singular to working precision* if

$$\kappa(A) > 1/\text{eps} \approx 10^{15}$$



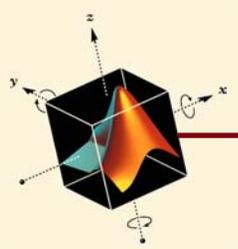
Exercise

Check out

$A \setminus b$

`cond(A)`

`condest(A)`

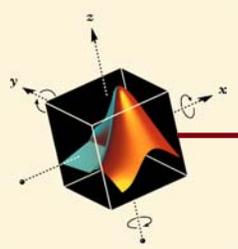


Eigenvalue decomposition

$$A = X \Lambda X^{-1}$$

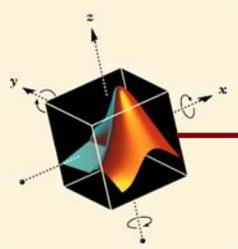
X eigenvectors

Λ diagonal, eigenvalues



Eigenvalue decomposition

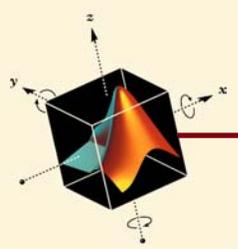
- » $A = \text{test matrix}$
- » $[V, D] = \text{eig}(A)$



Exercise

Distribution of eigenvalues

```
>> format short e  
>> e = eig(A)  
>> plot(e, 'o')  
>> semilogy(abs(e), 'o')
```



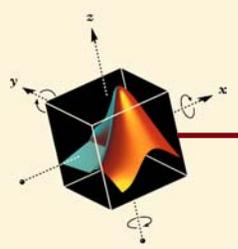
Alternate Exercise

Condition numbers of eigenvalues

```
>> help condeig
```

```
>> [V,D,s] = condeig(A)
```

```
>> cond(V)
```



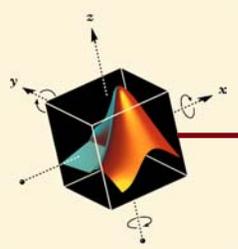
Singular Value Decomposition

$$A = U \Sigma V^T$$

U orthogonal

Σ diagonal, singular values

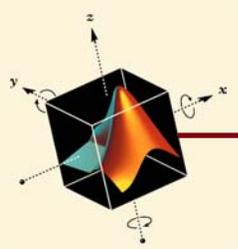
V orthogonal



Demonstration

```
>> eigshow
```

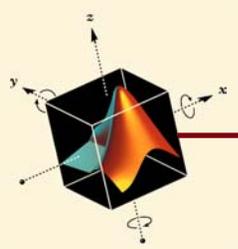
```
>> svdshow
```



Eigenvalues vs. Singular values

Eigenvalues

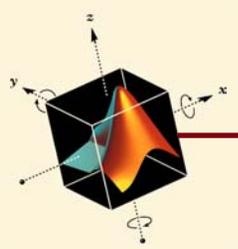
- ◆ A maps n -dimensional space onto itself
- ◆ Try to diagonalize with one change of basis
- ◆ Might be complex, even when A is real
- ◆ Might not exist; Jordan Canonical Form
- ◆ Analyze systems of o.d.e.'s



Eigenvalues vs. Singular values

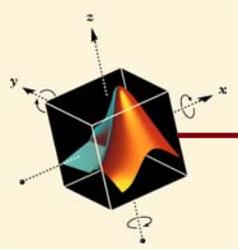
Singular values

- ◆ A maps n -space onto m -space
- ◆ Separate change of basis in domain and range
- ◆ Always real when A is real
- ◆ Always exists
- ◆ Transforming matrices are *orthogonal*
- ◆ Analyze systems of simultaneous linear eqns



Eigenvalues vs. Singular values

If A is square, symmetric and positive definite, then the eigenvalue decomposition and the singular value decomposition are the same.



Exercise

Compare

```
>> [X,D] = eig(A)
```

and

```
>> [U,S,V] = svd(A)
```

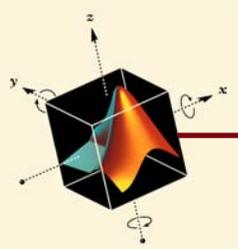
for various matrices, including

```
>> A = pascal(6)
```

```
>> A = gallery(3)
```

```
>> A = gallery(5)
```

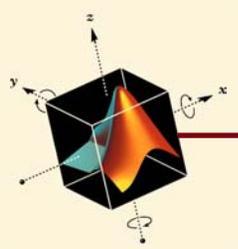
```
>> A = rosser
```



Ordinary Differential Equations

One step methods (Runge-Kutta)

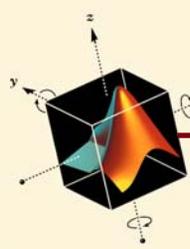
- ◆ Match Taylor series through specified order
- ◆ Single fixed formula
- ◆ Vary only step size
- ◆ Several function evaluations per step
- ◆ Minimal memory requirements



Ordinary Differential Equations

Multistep methods (Predictor-corrector)

- ◆ Polynomial approximations of various orders
- ◆ Family of formulas
- ◆ Vary order and step size each step
- ◆ Fewer function evaluations per step
- ◆ More memory required



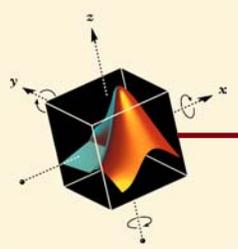
Ordinary Differential Equations

Explicit methods

- ◆ Explicit formula for each component
- ◆ Appropriate for *nonstiff* equations

Implicit methods

- ◆ Solve system of simultaneous equations at each time step
- ◆ Involves the *Jacobian*, $\partial F/\partial y$
- ◆ Appropriate for *stiff* equations

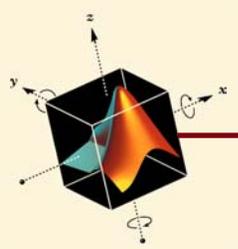


Stiff problems

Solutions *can* change on a time scale that is very short compared to the interval of integration, but the solution of interest changes on a much longer time scale.

Practical definition:

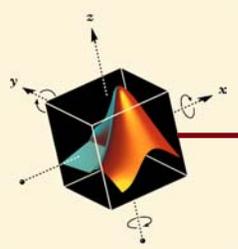
`ode45` uses too many time steps



Ordinary Differential Equations

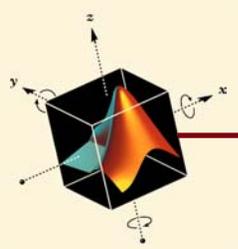
Modern methods

- ◆ Estimate error
- ◆ Compute appropriate step size
- ◆ Provide *interpolants* which allow the solution to be computed anywhere
- ◆ Provide *zero finding* and *event handling*



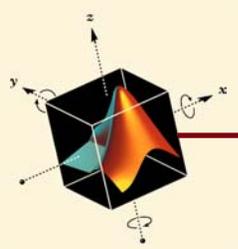
Ordinary Differential Equations

- ◆ Nonstiff
 - ◆ ode23
 - ◆ ode45
 - ◆ ode113
- ◆ Stiff
 - ◆ ode23s
 - ◆ ode15s



ODE Suite Methods

- ◆ **ode45** **Dormand-Prince formula**
(4,5) Runge-Kutta triple, order 5
- ◆ **ode23** **Bogacki-Shampine formula**
(2,3) Runge-Kutta triple, order 3
- ◆ **ode113** **Adams-Bashforth-Moulton, PECE**
based on ODE/STEP, INTRP, orders 1-13
- ◆ **ode15s** **Numerical Differentiation Formulas**
NDF family of Klopfenstein-Shampine, orders 1-5
- ◆ **ode23s** **modified Rosenbrock (2,3) triple**
W-method of Shampine-Reichelt, order 2



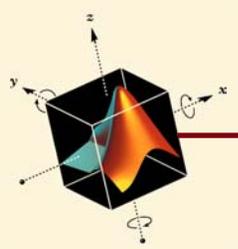
Time Step Selection

ode45, ode23, ode113

Choose steps small enough to resolve the fastest possible behavior

ode15s, ode23s

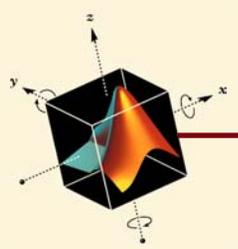
Choose steps just small enough to resolve the *actual* behavior



Nonstiff ODE methods

The best choice is *likely* to be

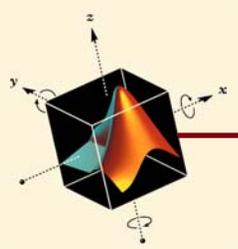
- ◆ **ode45** for most problems
- ◆ **ode23** if you want less accuracy or if $F(t,y)$ is not smooth
- ◆ **ode113** if you want much more accuracy or if $F(t,y)$ is very expensive



Stiff ODE methods

The best choice is *likely* to be

- ◆ **ode15s** for most problems
- ◆ **ode23s** if you want less accuracy
or if $F(t,y)$ is not smooth
or if $\partial F/\partial y$ is very cheap

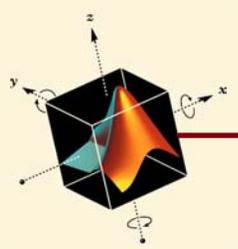


Using the stiff solvers

You can use `ode15s` and `ode23s` exactly like the codes for non-stiff problems, since $\partial F/\partial y$ is generated numerically by default.

But you may be able to speed up the computation greatly:

- ◆ vectorize the evaluation of $F(t,y)$
- ◆ provide the sparsity pattern of $\partial F/\partial y$
- ◆ evaluation of $\partial F/\partial y$ for large problems *requires* sparse matrix technology.

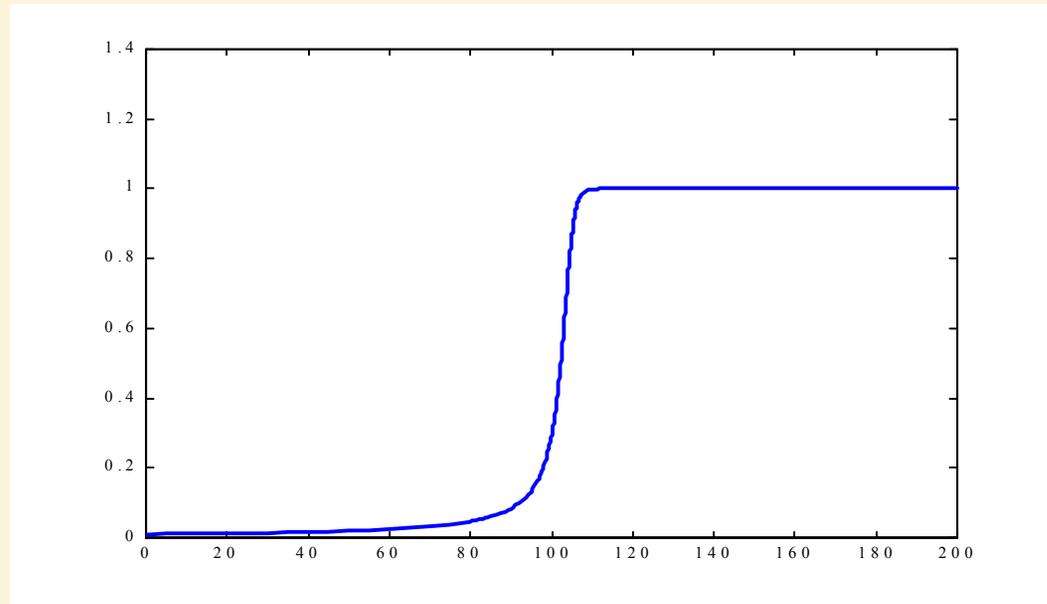


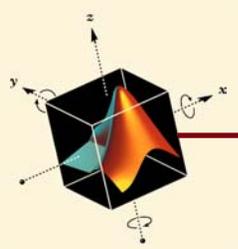
Flame propagation example

$$y' = y^2(1-y)$$

$$y(0) = 0.01$$

$$0 \leq t \leq 200$$

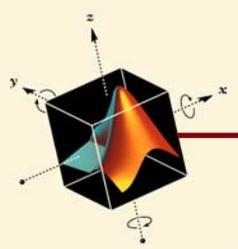




Exercise

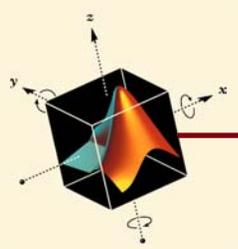
Try different ode solvers. Vary parameters.

- » `flame = inline('y.^2.*(1-y)', 't', 'y')`
- » `y0 = 1.e-4`
- » `tspan = [0 2/y0]`
- » `opts = odeset('reltol',1.e-5, ...
 'abstol',1.e-4)`
- » `ode23s(flame,tspan,y0,opts)`
- » `zoom on`



Zero finding

- ◆ Polynomials
 - ◆ `roots`
- ◆ Nonlinear functions
 - ◆ `fzero`
- ◆ Minimization
 - ◆ `fmin`



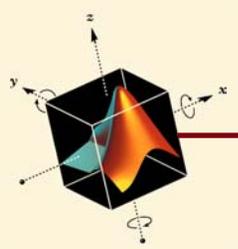
Companion matrix

The *roots* of the polynomial of degree n

$$p(x) = x^n + c_1x^{n-1} + \dots + c_n$$

are the *eigenvalues* of the n -by- n matrix

$$\begin{array}{ccccc} -c_1 & -c_2 & \dots & -c_{n-1} & -c_n \\ 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ & & & \cdot & \\ & & & \cdot & \\ 0 & 0 & \dots & 1 & 0 \end{array}$$



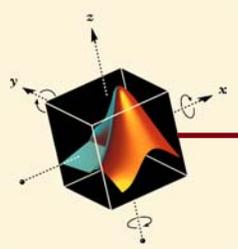
Polynomial roots

- » dbtype roots
- » dbstop 39 roots
- » `p = poly(1:5)`
- » `roots(p)`

```
39      r = [r; eig(a)];
```

```
K» a
```

```
K» dbcont
```



Multiple roots

What are the roots of

$$(x - a)^m = 0 ?$$

What are the roots of

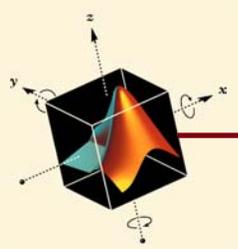
$$(x - a)^m = \text{eps} ?$$

» `p = poly(ones(1,16))`

» `z = roots(p)`

» `plot(z, 'o')`

» `axis equal`



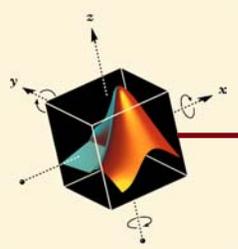
Exercise

J. H. Wilkinson's famous example

- » `p = poly(1:20)`
- » `delta = 2^(-23)`
- » `p(2) = -210 - delta`
- » `z = roots(p)`
- » `plot(z, 'o')`

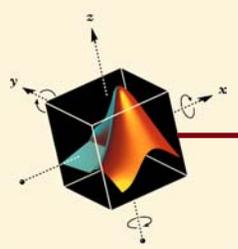
Make `delta` small enough to get real roots.

Which roots are most sensitive?



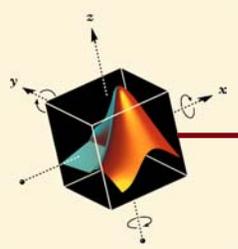
Zeros of scalar functions

fzero



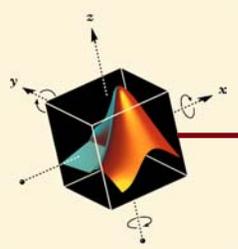
Fourier Transforms

- ◆ Transform matrix
 - ◆ `dftmtx`
- ◆ Fast Fourier Transform
 - ◆ `fft`, `ifft`, `fft2`, `ifft2`
- ◆ Convolution
 - ◆ `conv`



Finite Fourier Transform

- » $n = 8$
- » $\omega = \exp(-2 \cdot \pi \cdot i / n)$
- » $[k, j] = \text{meshgrid}(0:n-1)$
- » $F = \omega.^{(k.*j)}$
- » $\text{dftmtx}(8)$
- » $x = \text{randn}(n, 1)$
- » $F * x$
- » $\text{fft}(x)$

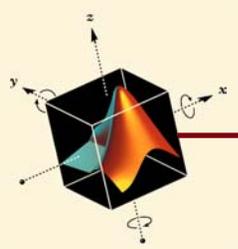


Fast Finite Fourier Transform

- » $I = \text{eye}(n, n)$
- » $P = I(:, [1:2:n \ 2:2:n])$
- » $H = \text{dftmtx}(n/2)$
- » $D = \text{diag}(\text{omega}.^{(0:n/2-1)})$

Theorem:

» $F == [H \ D*H; \ H \ -D*H]*P'$



Fast Finite Fourier Transform

```
function y = fft(x)
```

```
%FFT Fast Finite Fourier Transform.
```

```
% FFT(X) computes the same Fourier transform as FFT(X).
```

```
% The code uses a recursive divide and conquer algorithm
```

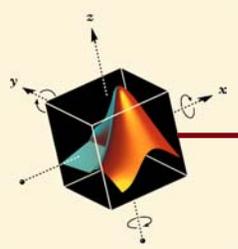
```
% for even order and matrix-vector multiplication for odd  
% order. If length(X) is n*m where n is even and m is  
% odd,
```

```
% the computational complexity of this approach is
```

```
%  $O(n \cdot \log(n)) \cdot O(m^2)$ .
```

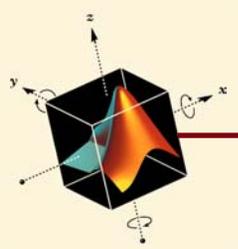
```
n = length(x);
```

```
omega = exp(-2*pi*i/n);
```



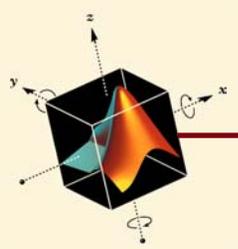
Fast Finite Fourier Transform

```
if rem(n,2) == 0
    % Recursive divide and conquer
    k = (0:n/2-1)';
    w = omega .^ k;
    u = fft(x(1:2:n-1));
    v = w.*fft(x(2:2:n));
    y = [u+v; u-v];
else
    % The Fourier matrix.
    j = 0:n-1;
    k = j';
    F = omega .^ (k*j);
    y = F*x;
end
```



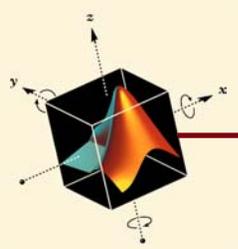
Random numbers

- ◆ Uniform distribution
 - ◆ `rand`
- ◆ Normal distribution
 - ◆ `randn`
- ◆ Other distributions
 - ◆ `help stats`



Sparse Matrices

- ◆ Creating sparse matrices
 - ◆ `sparse`, `spdiags`
- ◆ Linear equations
 - ◆ `\`, `symrcm`, `symmmd`, `colmmd`
- ◆ Iterative methods
 - ◆ `pcm`, `minres`, `bicgstab`, ...
- ◆ Eigenvalues
 - ◆ `eigs`, `svds`



Symbolic Computation

- ◆ Computer algebra systems
 - ◆ **Maple V, ...**
- ◆ Symbolic objects
 - ◆ **sym, syms**
- ◆ Calculus example
- ◆ Matrix example
- ◆ Function calculator
 - ◆ **funtool**