P. Arbenz
S. Pauli

Fall semester 2012

Numerical Methods for CSE

ETH Zürich
D-INF

Examination

January 29th, 2013

Duration of examination: 180 minutes. **Total points: 180**

Problem 1 Sparse matrix

In this problem we use the sparse matrix

```
A = delsq(numgrid('C',n));
```

(1a) [10 points] Write a Matlab script

```
function fillin()
```

which measures the operator complexity (sum of the nonzeros of all factors of a given factorization divided by the number of nonzeros of A) of different decompositions, and plots them for $n = 2^1, \ldots, 2^8$. Measure the operator complexity of the following things:

- Cholesky decomposition of `A`
- Incomplete Cholesky decomposition of `A` (see `ichol`)
- propose and implement an alternative way to reduce the fill in of the Cholesky decomposition.

Is is possible to measure an operator complexity number below 1? Explain why.

HINT: You may use the Matlab function `nnz`, and `ichol`.

Solution: Yes, a operator complexity number below 1 is possible, since the matrix $\mathbf{A}$ is symmetric with a lower and an upper triangular part. The Cholesky factor is only the lower triangular part. The operator complexity of the Cholesky factorization may therefore be as small as 0.5. This happens if the matrix is dense.

You may use reordering. The ones presented in the lecture are `symrcm`, `amd`, `symamd` and `colamd`. (e.g. ilu is not a good solution)

($n = 2^2, \ldots, 2^8$ is OK as well)

Listing 1 and Figure 1 (The solution is with `cholinc`, as `ichol` did not work with my matlab version.)

Listing 1: `fillin()`

```
function fillin()
clear all; close all;
k = 2.^[1:8];

fill = inf*ones(length(k), 3);
n = inf*ones(length(k), 1);

for l = 1:length(k)

  A = delsq(numgrid('C',k(l)));
  n(l) = size(A,1);
```

```
12
13    % Fill in measurements
14     fill(l,1) = nnz(A);
15     fill(l,2) = nnz(chol(A));
16     fill(l,3) = nnz(cholinc(A,10^-1));
17     p = amd(A);
18     fill(l,4) = nnz(chol(A(p,p)));
19  end
20
21  loglog(n, fill(:,1:4))
22  legend('A', 'chol', 'cholinc', 'amd')
23
24  print -deps2c 'fillin.eps'
```

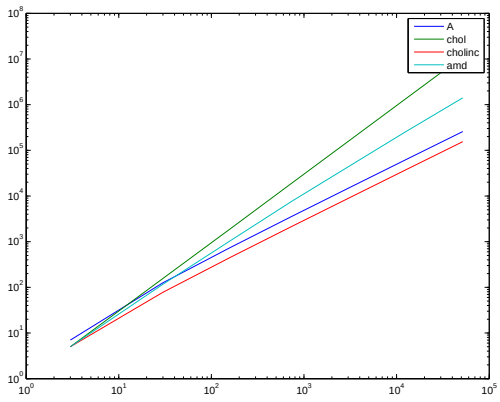

Figure 1: Convergence plot

(1b) [15 points] Plot the runtime (time to compute the solution $\mathbf{x}$, e.g. factorization and backward/forward substitution) and the measured error (norm of the residual) for $n = 2^1, \ldots, 2^8$ of the three different approaches (of (1a)) to solve $\mathbf{Ax} = \mathbf{b}$. Use a random vector $\mathbf{b}$. Implement this functionality in the Matlab script

`function conf()`

Comment on the obtained results.

HINT: You may use `tic`, `toc`, `rand`, `norm` and `sort`.

HINT: Read (1c) before starting this sub problem

HINT: As you measure the runtime call `maxNumCompThreads(1)` to insure that Matlab is running in single core mode.

Solution: Together with (1c)

(1c) [10 points] Propose and implement (in (1b)) an alternative method which takes advantage of the sparsity of $\mathbf{A}$, and solves $\mathbf{Ax} = \mathbf{b}$ not exactly but with a residual norm of 10^{-3}.

Comment on the obtained results (may be together with (1b)).

Solution: You should use CG or preconditioned CG.

The runtime of `amd` is the best option, with short runtime and small error. The error for ichol is large as expected since C in this case is only an approximate Cholesky factor of A and hence not useful. The runtime of `chol` seams to grow stronger than `pgc`.

($n = 2^2, \ldots, 2^8$ is OK as well)

Listing 2, Figure 3 and Figure 2

Listing 2: `conf()`

```
function conf()
clear all; close all;
%%%%%% Prevent Matlab from using multiple cores %%%%%%%
maxNumCompThreads(1)

MyTol = 10^-3;
k = 2.^[2:7];
t = inf*ones(length(k), 5);
err = inf*ones(length(k), 5);
n = inf*ones(length(k), 1);

for l = 1:length(k)
  A = delsq(numgrid('C',k(l)));
  n(l) = size(A,1);
  b = rand(size(A,1),1);
  MaxIt = min(1000,size(A,1));

  for j=1:3
    % time + error for cg:
    tic;
    x = pcg(A,b,MyTol,MaxIt);
    t(l, 1) = min(t(l, 1), toc);
    err(l, 1) = norm(A*x-b);

    % time + error for pcg:
    tic;
    M = cholinc(A,10^-1);
    x = pcg(A,b,MyTol,MaxIt,M',M);
    t(l, 2) = min(t(l, 2), toc);
    err(l, 2) = norm(A*x-b);

    % time + error for chol:
    tic;
    C = chol(A);
    x = C\(C'\b);
    t(l, 3) = min(t(l, 3), toc);
    err(l, 3) = norm(A*x-b);

    % time + error for cholinc:
    tic;
    C = cholinc(A,10^-1);
    x = C\(C'\b);
    t(l, 4) = min(t(l, 3), toc);
    err(l, 4) = norm(A*x-b);

    % time + error for amd:
    tic;
    p = amd(A);
    C = chol(A(p,p));
    x = C\(C'\b(p));
    [temp1, p1] = sort(p);
    x = x(p1);
    t(l, 5) = min(t(l, 3), toc);
    err(l, 5) = norm(A*x-b);

  end
end

loglog(n, t(:,1:5), '-*', n, 1e-6*n, 'k-.', n, 1e-8*n.^2, 'k--')
legend('cg', 'pcg', 'chol', 'cholinc', 'amd')

print -deps2c 'time.eps'
```

```
63
64 figure
65
66 loglog(n, err(:,1:5), '-*')
67 legend('cg', 'pcg', 'chol', 'cholinc', 'amd')
68 title('error')
69
70 print -deps2c 'conf.eps'
```

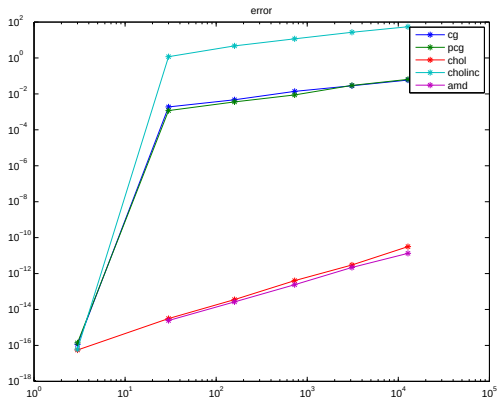

Figure 2: Error plot

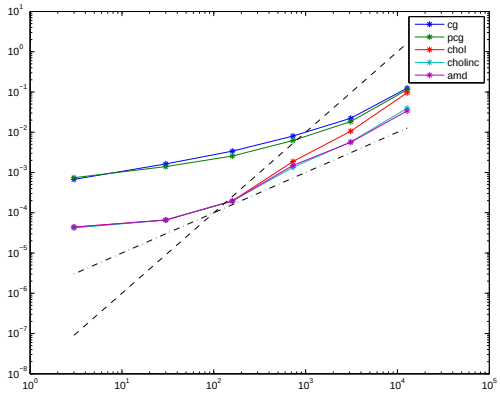

Figure 3: Convergence plot

Problem 2 Linear least squares

Let two vectors $\mathbf{z}, \mathbf{c} \in \mathbb{R}^n$, $n \in \mathbb{N}$ of measured data be given. The two numbers α^* and β^* are defined as

$$(\alpha^*, \beta^*) = \operatorname*{argmin}_{\alpha,\beta\in\mathbb{R}} \|\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c}\|_2, \tag{1}$$

with the tridiagonal matrix

$$\mathbf{T}_{\alpha,\beta} = \begin{pmatrix} \alpha & \beta & 0 & \dots & 0 \\ \beta & \alpha & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \alpha & \beta \\ 0 & \dots & 0 & \beta & \alpha \end{pmatrix} \in \mathbb{R}^{n,n}.$$

(2a) [8 points] Reformulate (1) as a linear least squares problem in the usual form

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^k} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2$$

with suitable $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{b} \in \mathbb{R}^m$, $m, k \in \mathbb{N}$.

Solution: The vector whose norm has to be minimized can be written as

$$\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} \overset{(1)}{=} \begin{pmatrix} \alpha z_1 + \beta z_2 - c_1 \\ \alpha z_2 + \beta(z_1 + z_3) - c_2 \\ \dots \\ \alpha z_{n-1} + \beta(z_{n-2} + z_n) - c_{n-1} \\ \alpha z_n + \beta z_{n-1} - c_n \end{pmatrix} \overset{(2)}{=} \begin{pmatrix} z_1 & z_2 \\ z_2 & z_1 + z_3 \\ \vdots & \vdots \\ z_{n-1} & z_{n-2} + z_n \\ z_n & z_{n-1} \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} - \mathbf{c} \overset{(3)}{=} \mathbf{A}\mathbf{x} - \mathbf{b}.$$

In this way, we have moved the unknown α and β into the vector $\mathbf{x}$, while the data $\mathbf{z}$ and $\mathbf{c}$ are moved into the matrix $\mathbf{A}$ and the hand side vector $\mathbf{b} = \mathbf{c}$ (that is not affected by the transformation). The number of unknown k is equal to 2 and the number of equations m is equal to n.

(2b) [13 points] Write a Matlab function

```
[alpha, beta] = lsqest(z,c)
```

that computes the values of the optimal parameter α^* and β^* according to (1) from the data vectors `z` and `c` (i.e., $\mathbf{z}$ and $\mathbf{c}$). Use the QR-decomposition to solve the linear least squares problem.

HINT: For $\mathbf{z} = (1, 2, \dots, 10)^T$ and $\mathbf{c} = (10, 9, \dots, 1)^T$ you should get $\alpha^* \approx -0.4211$ and $\beta^* \approx 0.5789$.

Solution:

Listing 3: Linear least squares for the problem (1).

```
function lsqest
z = (1:10)';
c = (10:-1:1)';
[alpha, beta] = lsqest(z,c)

% z and c are column vectors
function [alpha, beta] = lsqest(z,c)
A = [z, [z(2:end);0] + [0;z(1:end-1)]];
[Q,R] = qr(A);
c = Q'*c;
v = R\c;
alpha = v(1);
beta = v(2);
```

Problem 3 Bézier semi-circle

(3a) [12 points] Write a Matlab function

```
function plot_bezcurv(d)
```

which draws the Bézier curve, the control points $\mathbf{d}$, and the convex hull defined by the control points $\mathbf{d}$ ($2 \times n$ matrix).

HINT: You may use `convhull` and `fill`

Solution: Listing 4

Listing 4: `plot_bezcurv()`

```
function plot_bezcurv(d)
n = size(d,2)-1;                    %number of nodes
k = convhull(d(1,:),d(2,:));        %k =indices of re-ordered vertices of the
    convex hull
fill(d(1,k),d(2,k),[0.8 1 0.8]);    %draw the c.h. using reordered vertices
hold on;
plot(d(1,:),d(2,:),'b-*');          %draw the nodes

```

```
x = 0:0.001:1;
V = bernstein(n,x);
bc = d*V;                                   %compute the Bezier curve
plot(bc(1,:),bc(2,:),'r-','LineWidth',2);          %plot the Bezier curve

xlabel('x');
ylabel('y');
legend('Convex Hull','Control points','Bezier curve');
hold off;
end

function B = bernstein(n,t)
B = [ones(1,length(t)); zeros(n,length(t))];
for j=1:n
  for k=j+1:-1:2
    B(k,:) = t.*B(k-1,:) + (1-t).*B(k,:);
  end
  B(1,:) = (1-t).*B(1,:);
end
end
```

(3b) [12 points] Write a Matlab function

```
len = function  bezLength(d)
```

which approximately computes the length of the Bézier curve by some numerical quadrature.

Solution: Listing 5

Listing 5: `bezLength()`

```
function d = bezLength(d)
n = size(d,2)-1;                            %number of nodes
x = 0:0.001:1;
V = bernstein(n,x);
bc = d*V;                                   %compute the Bezier curve

% Add up the segments of a bezier curve. Estimate the length of each
% segment with the Pythagorean theorem.
d = sqrt(sum((bc(:,1:end-1) - bc(:,2:end)).^2));
d = sum(d);
end

function B = bernstein(n,t)
B = [ones(1,length(t)); zeros(n,length(t))];
for j=1:n
  for k=j+1:-1:2
    B(k,:) = t.*B(k-1,:) + (1-t).*B(k,:);
  end
  B(1,:) = (1-t).*B(1,:);
end
end
```

(3c) [16 points] We want to approximate a unit half circle by (one segment of) a Bézier curve with 5 Bézier control points. The Bézier control points shall be such that

- the Bézier curve is symmetric.
- the Bézier curve passes through the points $\binom{-1}{0}$, $\binom{0}{1}$ and $\binom{1}{0}$.
- the tangent at the two end points $\binom{-1}{0}$ and $\binom{1}{0}$ corresponds to the tangent of the semi circle.

- the length of the Bézier curve corresponds to the length of a semi circle (π).

Write a Matlab function

```
function halfcircle(d)
```

where you compute the Bézier control points and plot the resulting Bézier curve.

HINT: You may use the following 5 Bézier control points: $\binom{-1}{0}$, $\binom{-1}{\alpha}$, $\binom{0}{\beta}$, $\binom{1}{\alpha}$ and $\binom{1}{0}$

HINT: Find a linear (analytical) relation between α and β such that the Bézier curve passes through the point $\binom{0}{1}$. Then only one independent parameter is left. Then use a **simple** numerical method from the lecture to determine a good value for this parameter such that the Bézier curve has length π

HINT: If you are unable to complete task (3a) and/or (3b) use the functions `plot_bezcurv_p(d)` and/or `len = bezLength_p(d)` instead.

Solution: The Bezier curve is given by

$$P(t) = \sum_{i=0}^{4} P_i B_{i,4}(t)$$

with

$$B_{i,4}(t) = \binom{4}{i}(1-t)^{4-i}t^i.$$

With the 5 Bezier points the Bezier curve yields

$$P(t) = \binom{-1}{0}(1-t)^4 + 4\binom{-1}{\alpha}(1-t)^3t^1 + 6\binom{0}{\beta}(1-t)^2t^2 + 4\binom{1}{\alpha}(1-t)^1t^3 + \binom{1}{0}t^4.$$

To determine α and β we use that $P(t)$ for $t = 0.5$ should lie on the half unit circle. $P(.5)$ is in the middle of the half circle, hence

$$P(.5) = \begin{pmatrix} 0 \\ (8\alpha + 6\beta)0.5^4 \end{pmatrix} = \binom{0}{1},$$

leading to $\beta = \frac{8}{3} - \frac{4}{3}\alpha$.

Listing 6

Listing 6: `halfcircle()`

```
function halfcircle
clear all; close all;
bezcurv([-1 -1 0 1 1; 0 0.873 1.5027 0.873 0]);

%bisection
bigAlpha = 1;
smallAlpha = .5;
beta = 8/3-4/3*bigAlpha;
alpha = bigAlpha
d = bezLength([-1 -1 0 1 1; 0 alpha beta alpha 0]);
i = 0;
while abs(d-pi) > 10^-10
alpha = (bigAlpha+smallAlpha)/2;
beta = 8/3-4/3*alpha;
d = bezLength([-1 -1 0 1 1; 0 alpha beta alpha 0]);

if d < pi
   smallAlpha = alpha;
else
    bigAlpha = alpha;
end
i=i+1

```

```
24 end
25 beta
26
27 end
```

Problem 4 Best rank-1 approximation

Given $\mathbf{A} \in \mathbb{R}^{n,n}$ with positive diagonal we consider the minimization problem

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} \left\| \mathbf{A} - \mathbf{x}\mathbf{x}^\top \right\|_F^2 . \tag{2}$$

(4a) [6 points] Reformulate (2) as a standard non-linear least squares problem $\frac{1}{2}\|F(\mathbf{x})\|_2^2 \to \min$ for a suitable function F.

Solution:

$$F_{i+(k-1)n} = a_{ik} - x_i x_k \quad i,k = 1,\ldots,n$$

(4b) [15 points] Derive the Jacobian and implement it in the Matlab function

```
function df = DF(x).
```

Solution:

$$J(\mathbf{x}) = \begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_{n^2}}{\partial x_1} & \cdots & \frac{\partial F_{n^2}}{\partial x_n} \end{bmatrix}$$

$$\begin{bmatrix} \frac{\partial F_1}{\partial x_1} \\ \vdots \\ \frac{\partial F_{n^2}}{\partial x_1} \end{bmatrix} = \begin{bmatrix} -2x_1 \\ -x_2 \\ \vdots \\ -x_n \\ -x_2 \\ 0 \\ \vdots \\ 0 \\ -x_3 \\ 0 \\ \vdots \\ 0 \\ -x_4 \\ \vdots \end{bmatrix} \qquad \begin{bmatrix} \frac{\partial F_1}{\partial x_i} \\ \vdots \\ \frac{\partial F_{n^2}}{\partial x_i} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -x_1 \\ 0 \\ \vdots \\ 0 \\ -x_2 \\ 0 \\ \vdots \\ -x_{i-1} \\ 0 \\ \vdots \\ 0 \\ -x_1 \\ -x_2 \\ \vdots \\ -x_{i-1} \\ -2x_i \\ x_{i+1} \\ \vdots \\ x_n \\ 0 \\ \vdots \\ 0 \\ -x_{i+1} \\ 0 \\ \vdots \end{bmatrix}$$

Listing 7

Listing 7: `DF()`

```
function df = DF( x )
% construct DF

n = length(x);
df = zeros(n*n,n);
for i = 1:n
    first = 1 + (i-1)*n;
    last  = n + (i-1)*n;
    df(first:last,i) = -x;
    ind = [1:n:n^2] + i-1;
    df(ind,i) = df(ind,i) -x;
end

end
```

(4c) [11 points] Write a Matlab code

`function x = rankoneapprox(A)`

that computes the solution of (2) by means of the Gauss-Newton iteration with initial guess $x_i^{(0)} = \sqrt{a_{ii}}, i = 1, \ldots, n$ with a tolerance of 10^{-3}.

HINT: You may use `reshape`. HINT: If you are unable to complete task 2 use the function `df = DF_p(x)` instead.

HINT: Use `test4.m` to test your function.

Solution: Listing 8

Listing 8: `rankoneapprox()`

```
function x = rankoneapprox( A )

% config
tol = 1e-6;
n = size(A,1);
x = sqrt(diag(A));

% construct F
F = @(x) reshape(A,n*n,1) - reshape(x*x',n*n,1);

% Newton correction
% s = DF(x) \ F(x);

% Newton iterations
while norm(s) > tol * norm(x)
    s = DF(x) \ F(x);
    x = x - s;
end

end
```

(4d) [8 points] Explain, why the MATLAB built in function `eig` can be used to solve (2) provided that $\mathbf{A}$ is *symmetric* (hermitian).

Solution:

$$\left\|\mathbf{A}-\mathbf{x}\mathbf{x}^{\top}\right\|_F = \sqrt{\sum_{i=1}^{n}\sigma_i^2(\mathbf{A}-\mathbf{x}\mathbf{x}^{\top})} \tag{3}$$

Choose $\mathbf{x}$ as the eigenvalue of the largest eigenvector of $\mathbf{A}$ times the square root of the largest eigenvalue. This minimizes the right hand side of (3).

(4e) [5 points] Write a MATLAB code

`function x = symrankoneapprox(A)`

that computes the solution of (2) for *symmetric* (hermitian) $\mathbf{A}$ using MATLAB's `eig`.

HINT: Use `test4.m` to test your function.

Solution:

Listing 9

Listing 9: `symrankoneapprox()`

```
function x = symrankoneapprox( A )

[V,D] = eig(A);
D = diag(D);
[d,ind] = max(D);
x = V(:,ind)*sqrt(d);

end
```

Problem 5 ODE / Runge-Kutta method

We want to solve the initial value problem $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$ by the Runge-Kutta method characterized by the following Butcher table:

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{2}{3} & \frac{2}{3} & 0 & \\ 0 & -1 & 1 & 0 \\ \hline & 0 & \frac{3}{4} & \frac{1}{4} \end{array} \tag{4}$$

(5a) [11 points]

Implement two Matlab functions

```
y1 = RK_step(odefun, t, y0, h),
```

and

```
[t, y] = RK(odefun, tspan, y0, N).
```

`RK_step(odefun, t, y0, h)` implements one Runge-Kutta step defined by (4), from `t` to `t+h` with the starting value `y0`. `odefun(t,y)` defines the right hand side of the initial value problem $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$.

`RK(odefun,tspan,y0,N)` uses `RK_step` to solve the ODE over the whole interval specified by `tspan` using $N \in \mathbb{N}$ uniform timesteps.

Solution: Listing 10 and Listing 11

Listing 10: `RK_step()`

```
function y1 = RK_step(odefun, t, y0, h)

K1 = odefun(t,           y0               );
K2 = odefun(t + h*2/3,   y0 + h*K1*2/3    );
K3 = odefun(t,           y0 + h*(K2 - K1));

y1 = y0 + h*K2*3/4 + h*K3/4;
```

Listing 11: `RK()`

```
function [t, y] = RK(odefun,tspan,y0,N)

h = diff(tspan)/N;
t = linspace(tspan(1),tspan(2),N+1);
y = [y0, zeros(length(y0),N)];

for k = 1:N
    y(:,k+1) = RK_step_p(odefun, t(k), y(:,k), h);
end

y = y.';          % change to Matlab convention
```

(5b) [5 points] Bring the following ODE into an suitable form to solve it with `RK_step(odefun, t, y0, h)`:

$$x''(t) + x(t) = \sin(t), \quad x(0) = 100, \quad x'(0) = 5. \tag{5}$$

Implement this suitable form in the Matlab function

```
out = odefun(t, y).
```

Solution: Listing 12

Listing 12: `odefun()`

```
function out = odefun(t, y)

out = [y(2); sin(t)-y(1)];
```

(5c) [12 points] Implement a Matlab function

`exRK()`,

to graphically (with a plot) determine the order of the chosen Runge-Kutta method (4) for the given ODE (5). Use the Matlab function `ode45` with relative tolerance `100 * eps` and absolute tolerance `eps` to determine a reference solution.

HINT: The error of a method could be computed by $\|x(T)-\hat{x}(T)\|_{L_2}+\|x'(T)-\hat{x}'(T)\|_{L_2}$, where x is the reference solution and $\hat{x}$ the approximate one. T is the stopping time.

HINT: You may use `norm`.

HINT: If you are unable to complete task (5a) and/or (5b) use the functions `[t, y] = RK_p(odefun,tspan,y0,N)` and/or `out = odefun_p(t, y)` instead.

Solution: Listing 13 and Figure 4

Listing 13: `exRK()`

```
function exRK
clear all; close all; tic;

% Interval and initial condition:
tspan = [0 1];
y0 = [100; 5];

% Reference solution with ode45:
opts = odeset('RelTol', 100*eps, 'AbsTol', eps);
[t, y_ex] = ode45(@odefun, tspan, y0, opts);
ex = y_ex(end,:);           % extract final value to compute error

h = 2.^(-(4:10));           % meshwidth
n = diff(tspan) ./ h;       % # steps

for l = 1:length(n)
  [t, y_rk] = RK(@odefun, tspan, y0, n(l));
  err_rk(l) = norm(ex - y_rk(end,:));
  a = y_rk(end,:);
  err_rk(l) = norm(a(1)-ex(1)) + norm(a(2)-ex(2));
end

figure(2);
loglog( h, err_rk,'r', h, 1e3*h.^3 ,'k--');
axis tight; xlabel('meshwidth_h'); ylabel('error');
legend('3-stage_RK', 'h^3', 'location','nw');
set(0,'DefaultLineLineWidth','default');
print -depsc2 'ExRKconvergence.eps';
toc
```

(5d) [13 points] Analytically determine the convergence order and the stability interval of the Runge-Kutta method in (4)

Solution: Ansatz: $y'(t)=\lambda y, \quad y(0)=1$ for negative λ the solution is known: $y(t)=e^{\lambda t}$.

For the given RK table we have:

$$k_1 = f(t,y) = \lambda_i y_j$$

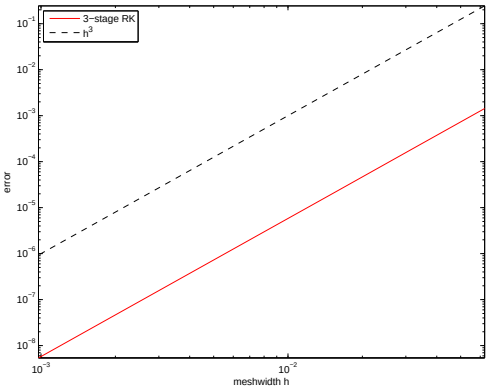

Figure 4: Convergence plot

$$k_2 = f(t + \frac{2}{3}h, y_j + \frac{2}{3}hk_1) = \lambda_i(y_j + 2/3hk_1) = (\lambda_i + \frac{2}{3}h\lambda_i^2)y_j$$

$$k_3 = f(t + 0, y_j + h(-k_1 + k_2)) = \lambda_i(y_j + h(-k_1 + k_2)) = (\lambda_i - h\lambda_i^2 + h\lambda_i^2 + \frac{2}{3}h^2\lambda_i^3)y_j = (\lambda_i + \frac{2}{3}h^2\lambda_i^3)y_j$$

$$y_{j+1} = y_j + h(\frac{3}{4}k_2 + \frac{1}{4}k_3) = (1 + \frac{3}{4}h\lambda_i + \frac{3}{4}\frac{2}{3}h^2\lambda_i^2 + \frac{1}{4}h\lambda_i + \frac{1}{4}\frac{2}{3}h^3\lambda_i^3)y_j = (1 + h\lambda_i + \frac{1}{2}h^2\lambda_i^2 + \frac{1}{6}h^3\lambda_i^3)y_j$$

This are 4 terms of the Taylor series of $e^{h\lambda}$, hence the RK method has order 3.

$$F(h\lambda) = 1 + h\lambda + \frac{1}{2}h^2\lambda^2 + \frac{1}{6}h^3\lambda^3$$

The stability interval is given by $B = \{\mu \in \mathbb{R} || F(\mu)| < 1\}$. By solving for $F(\mu) = 1$ is easy and leads to $\mu = 0$. Solving for $F(\mu) = -1$ is harder and leads to $\mu \approx -2.51$ (alternatively this result is available in the slides). This leads to the stability interval $(-2.51, 0)$.