Prof. R. Hiptmair
G. Alberti,
F. Leonardi

AS 2015

Numerical Methods for CSE

ETH Zürich
D-MATH

Problem Sheet 5

Problem 1. Modified Newton method (core problem)

The following problem consists in EIGEN implementation of a modified version of the Newton method (in one dimension [1, Section 2.3.2.1] and many dimensions [1, Section 2.4]) for the solution of a nonlinear system. Refresh yourself on stopping criteria for iterative methods [1, Section 2.1.2].

For the solution of the non-linear system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ (with $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$), the following iterative method can be used:

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})\,, \\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{y}^{(k)})\,, \end{aligned} \tag{16}$$

where $D\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{n,n}$ is the Jacobian matrix of $\mathbf{F}$ evaluated in the point $\mathbf{x}$.

(1a) ⚁ Show that the iteration (16) is consistent with $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ in the sense of [1, Def. 2.2.1], that is, show that $\mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every $k \in \mathbb{N}$, if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$ and $D\mathbf{F}(\mathbf{x}^{(0)})$ is regular.

Solution: If $\mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{0}$ then $\mathbf{y}^{(k)} = \mathbf{x}^{(k)} + \mathbf{0} = \mathbf{x}^{(k)}$ and $\mathbf{x}^{(k+1)} = \mathbf{y}^{(k)} - \mathbf{0} = \mathbf{x}^{(k)}$.
So, by induction, if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$ then $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every k.

Conversely, if $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$, then, by the recursion of the Newton method:

$$\begin{aligned} \mathbf{F}(x^{(k)}) = \mathbf{F}(x^{(k)} + D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})) \\ \Rightarrow D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)}) = 0 \\ \Rightarrow \mathbf{F}(\mathbf{x}^{(k)}) = 0 \end{aligned}$$

(1b) ⚁ Implement a C++ function

```
template <typename arg, class func, class jac>
void mod_newt_step(const arg & x, arg & x_next,
                   func&& f, jac&& df);
```

that computes a step of the method (16) for a *scalar* function $\mathbf{F}$, that is, for the case $n = 1$.

Here, `f` is a lambda function for the function $F : \mathbb{R} \mapsto \mathbb{R}$ and `df` a lambda to its derivative $F' : \mathbb{R} \mapsto \mathbb{R}$.

HINT: Your lambda functions will likely be:

```
auto f = [ /* captured vars. */ ] (double x)
       { /* fun. body */ };
auto df = [ /* captured vars. */ ] (double x)
        { /* fun. body */ };
```

Notice that here `auto` is `std::function<double(double)>`.

HINT: Have a look at:

- http://en.cppreference.com/w/cpp/language/lambda
- https://msdn.microsoft.com/en-us/library/dd293608.aspx

for more information on lambda functions.

Solution: See `modnewt.cpp` and `general_nonlinear_solver.h`.

(1c) ⚁ What is the order of convergence of the method?
To investigate it, write a C++function `void mod_newt_ord()` that:

- uses the function `mod_newt_step` from subtask (1b) in order to apply (16) to the following scalar equation

$$\arctan(x) - 0.123 = 0 \; ;$$

- determines empirically the order of convergence, in the sense of [1, Rem. 2.1.19] of the course slides;

- implements meaningful stopping criteria ([1, Section 2.1.2]).

Use $x_0 = 5$ as initial guess.

HINT: the exact solution is $x = \tan(a) = 0.123624065869274\ldots$

HINT: remember that $\arctan'(x) = \frac{1}{1+x^2}$.

Solution: See `modnewt.cpp` and `general_nonlinear_solver.h`. The order of convergence is approximately 3:

sol.	err.	order
5	4.87638	
0.560692	4.87638	
0.155197	0.437068	
0.123644	0.0315725	1.08944
0.123624	2.00333e-05	2.80183
0.123624	5.19029e-15	2.99809

(1d) ⚃ Write a C++ function `void mod_newt_sys()` that provides an efficient implementation of the method (16) for the non-linear system

$$\mathbf{F}(\mathbf{x}) := \mathbf{A}\mathbf{x} + \begin{pmatrix} c_1 e^{x_1} \\ \vdots \\ c_n e^{x_n} \end{pmatrix} = \mathbf{0}\,,$$

where $\mathbf{A} \in \mathbb{R}^{n,n}$ is symmetric positive definite, and $c_i \geq 0$, $i = 1, \ldots, n$. Stop the iteration when the Euclidean norm of the increment $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ relative to the norm of $\mathbf{x}^{(k+1)}$ is smaller than the tolerance passed in `tol`. Use the zero vector as initial guess.

HINT: You can proceed as in the previous task: define function handles as lambda functions and create a "stepping" function, which you the iterate inside a for loop.

HINT: Try and be as efficient as possible. Reuse the matrix factorization when possible.

Solution: See `modnewt.cpp` and `general_nonlinear_solver.h`.

Problem 2. Solving a quasi-linear system (core problem)

In [1, § 2.4.14] we studied Newton's method for a so-called quasi-linear system of equations, see [1, Eq. (2.4.15)]. In [1, Ex. 2.4.19] we then dealt with concrete quasi-linear

system of equations and in this problem we will supplement the theoretical considerations from class by implementation in EIGEN. We will also learn about a simple fixed point iteration for that system, see [1, Section 2.2]. Refresh yourself about the relevant parts of the lecture. You should also try to recall the Sherman-Morrison-Woodbury formula [1, Lemma 1.6.112].

Consider the *nonlinear* (quasi-linear) system:

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}\,,$$

as in [1, Ex. 2.4.19]. Here, $\mathbf{A} : \mathbb{R}^n \to \mathbb{R}^{n,n}$ is a matrix-valued function:

$$\mathbf{A}(\mathbf{x}) := \begin{bmatrix} \gamma(\mathbf{x}) & 1 & & & & \\ 1 & \gamma(\mathbf{x}) & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & \gamma(\mathbf{x}) & 1 \\ & & & & 1 & \gamma(\mathbf{x}) \end{bmatrix}, \quad \gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2$$

where $\|\cdot\|_2$ is the Euclidean norm.

(2a) ⚀ A fixed point iteration fro (Problem 2.) can be obtained by the "frozen argument technique"; in a step we take the argument to the matrix valued function from the previous step and just solve a linear system for the next iterate. State the defining recursion and iteration function for the resulting fixed point iteration.

(2b) ⚁ We consider the fixed point iteration derived in sub-problem (2a). Implement a function computing the iterate $\mathbf{x}^{(k+1)}$ from $\mathbf{x}^{(k)}$ in EIGEN.

HINT: (Optional) This is classical example where *lambda* C++11 functions may become handy.

Write the iteration function as:

```
template <class func, class Vector>
void fixed_point_step(func&& A, const Vector & b, const
    Vector & x, Vector & x_new);
```

where `func` type will be that of a *lambda* function implementing $\mathbf{A}$. The vector `b` will be an input random r.h.s. vector. The vector `x` will be the input $\mathbf{x}^{(k)}$ and `x_new` the output $\mathbf{x}^{(k+1)}$.

Then define a *lambda* function:

```
auto A = [ /* TODO */ ] (const Eigen::VectorXd & x) ->
    Eigen::SparseMatrix<double> & { /* TODO */ };
```

returning $\mathbf{A}(\mathbf{x})$ for an input $\mathbf{x}$ (*capture* the appropriate variables). You can then call your stepping function with:

```
fixed_point_step(A, b, x, x_new);
```

Include `#include <functional>` to use `std::function`.

Solution: See `quasilin.cpp`.

(2c) ⊡ Write a routine that finds the solution $\mathbf{x}^*$ with the fixed point method applied to the previous quasi-linear system. Use $\mathbf{x}^{(0)} = \mathbf{b}$ as initial guess. Supply it with a suitable correction based stopping criterion as discussed in [1, Section 2.1.2] and pass absolute and relative tolerance as arguments.

Solution: See `quasilin.cpp` .

(2d) ⊡ Let $\mathbf{b} \in \mathbb{R}^n$ be given. Write the recursion formula for the solution of

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}$$

with the Newton method.

Solution: The Newton iteration, as provided in the lecture notes, reads:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2}\right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}). \tag{17}$$

(2e) ⊡ The matrix $\mathbf{A}(\mathbf{x})$, being symmetric and tri-diagonal, is cheap to invert. Rewrite the previous iteration efficiently, exploiting, the Sherman-Morrison-Woodbury inversion formula for rank-one modifications [1, Lemma 1.6.112].

Solution: We replace the inversion with the SMW formula:

$$
\begin{aligned}
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2}\right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1}\left(\mathbf{I} - \frac{1}{\|\mathbf{x}^{(k)}\|_2}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}}{1 + \frac{(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|_2}}\right)(\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b} + \mathbf{A}(\mathbf{x}^{(k)})^{-1}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}(\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1}\left(\mathbf{b} + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top(\mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b})}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}\right)
\end{aligned}
$$

(2f) ⚃ Implement the above step of Newton method in EIGEN.

HINT: If you didn't manage to solve subproblem (2e) use directly the formula from (2d).

HINT: (Optional) Feel free to exploit lambda functions (as above), writing a function:

```
template <class func, class Vector>
void newton_step(func&& A, const Vector & b, const
    Vector & x, Vector & x_new);
```

Solution: See `quasilin.cpp`.

(2g) ⚀ Repeat subproblem (2c) for the Newton method. As initial guess use $\mathbf{x}^{(0)} = \mathbf{b}$.

Solution: See `quasilin.cpp`.

Problem 3. Nonlinear electric circuit

In previous exercises we have discussed electric circuits with elements that give rise to linear voltage–current dependence, see [1, Ex. 1.6.3] and [1, Ex. 1.8.1]. The principles of nodal analysis were explained in these cases.

However, the electrical circuits encountered in practise usually feature elements with a *non-linear* current-voltage characteristic. Then nodal analysis leads to non-linear systems of equations as was elaborated in [1, Ex. 2.0.1]. Please note that transformation to frequency domain is not possible for non-linear circuits so that we will always study the direct current (DC) situation.

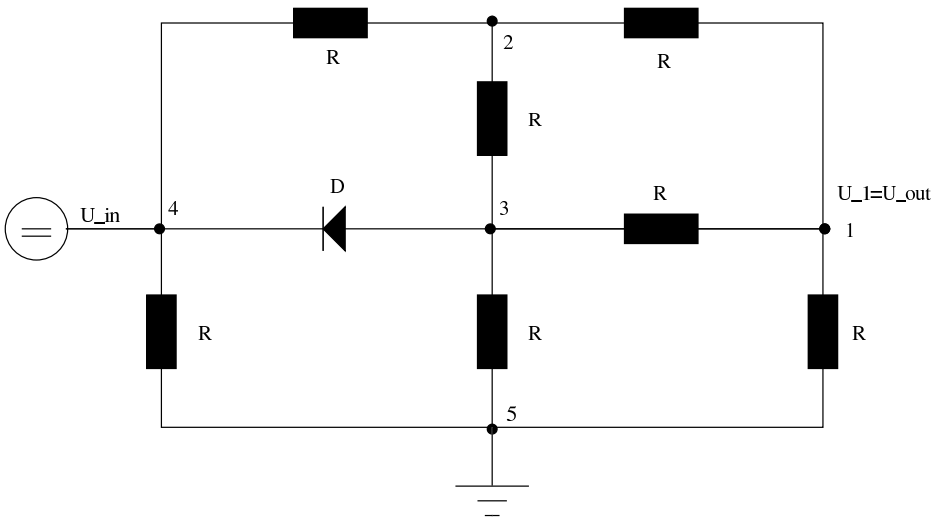

Figure 11: non-linear circuit for Problem 1

In this problem we deal with a very simple non-linear circuit element, a diode. The current through a diode as a function of the applied voltage can be modelled by the relationship

$$I_{kj} = \alpha\left(e^{\beta\frac{U_k - U_j}{U_T}} - 1\right),$$

with suitable parameters α, β and the thermal voltage U_T.

Now we consider the circuit depicted in Fig. 11 and assume that all resistors have resistance $R = 1$.

(3a) ⚀ Carry out the nodal analysis of the electric circuit and derive the corresponding non-linear system of equations $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ for the voltages in nodes 1,2, and 3, *cf.* [1, Eq. (2.0.2)]. Note that the voltages in nodes 4 and 5 are known (input voltage and ground voltage 0).

Solution: We consider Kirchhoff's law $\sum_{k,j} I_{kj} = 0$ for every node k. I_{kj} contributes to the sum if node j is connected to node k trough one element (R,C,L,D).

(1) $3U_1 - 0 - U_2 - U_3 = 0$

(2) $3U_2 - U_1 - U_3 - U = 0$

(3) $3U_3 - 0 - U_1 - U_2 + \alpha\left(e^{\beta\left(\frac{U_3 - U}{U_T}\right)} - 1\right) = 0$

Thus the nonlinear system of equations reads

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 3U_1 - U_2 - U_3 \\ 3U_2 - U_1 - U_3 - U \\ 3U_3 - U_1 - U_2 + \alpha e^{-\frac{\beta}{U_T}U} \cdot e^{\frac{\beta}{U_T}U_3} - \alpha \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \qquad \text{for } \mathbf{u} = \begin{pmatrix} U_1 \\ U_2 \\ U_3 \end{pmatrix}.$$

(3b) Write an EIGEN function

```
void circuit(const double & alpha, const double & beta, const VectorXd & Uin, VectorXd & Uout)
```

that computes the output voltages `Uout` (at node 1 in Fig. 11) for a *sorted* vector of input voltages `Uin` (at node 4) for a thermal voltage $U_T = 0.5$. The parameters `alpha`, `beta` pass the (non-dimensional) diode parameters.

Use Newton's method to solve $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ with a tolerance of $\tau = 10^{-6}$.

Solution: An iteration of the multivariate Newton method is:

$$\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} - \left(D\mathbf{F}(\mathbf{u}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{u}^{(k)})$$

The components of the Jacobian $D\mathbf{F}$ are defined as $D\mathbf{F}(\mathbf{u})_{ij} := \frac{\partial F_i(\mathbf{u})}{\partial u_j}$. For the function obtained in a) we get

$$D\mathbf{F}(\mathbf{u}) = \mathbf{J} = \begin{pmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 + \frac{\alpha\beta}{U_T} e^{\beta\frac{U_3 - U}{U_T}} \end{pmatrix}.$$

For the implementation in Eigen see file `nonlinear_circuit.cpp`.

(3c) We are interested in the nonlinear effects introduced by the diode. Calculate $U_{\text{out}} = U_{\text{out}}(U_{\text{in}})$ as a function of the variable input voltage $U_{\text{in}} \in [0, 20]$ (for non-dimensional parameters $\alpha = 8$, $\beta = 1$ and for a thermal voltage $U_T = 0.5$) and infer the nonlinear effects from the results.

Solution: The nonlinear effects can be observed by calculating the differences between the solutions U_{out}, see file `nonlinear_circuit.cpp`.

Problem 4. Julia set

Julia sets are famous fractal shapes in the complex plane. They are constructed from the basins of attraction of zeros of complex functions when the Newton method is applied to find them.

In the space $\mathbb{C}$ of complex numbers the equation

$$z^3 = 1 \tag{18}$$

has three solutions: $z_1 = 1$, $z_2 = -\frac{1}{2} + \frac{1}{2}\sqrt{3}i$, $z_3 = -\frac{1}{2} - \frac{1}{2}\sqrt{3}i$ (the cubic roots of unity).

(4a) As you know from the analysis course, the complex plane $\mathbb{C}$ can be identified with $\mathbb{R}^2$ via $(x, y) \mapsto z = x + iy$. Using this identification, convert equation (18) into a system of equations $\mathbf{F}(x, y) = \mathbf{0}$ for a suitable function $\mathbf{F} : \mathbb{R}^2 \mapsto \mathbb{R}^2$.

Solution: We have

$$z^3 - 1 = (x + iy)^3 - 1 = x^3 + 3ix^2y - 3xy^2 - iy^3 - 1 = x^3 - 3xy^2 - 1 + i(3x^2y - y^3) = 0.$$

Thus, equation (18) is equivalent to $\mathbf{F}(x, y) = \begin{pmatrix} x^3 - 3xy^2 - 1 \\ 3x^2y - y^3 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$.

(4b) Formulate the Newton iteration [1, Eq. (2.4.1)] for the non-linear equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ with $\mathbf{x} = (x, y)^T$ and $\mathbf{F}$ from the previous sub-problem.

Solution: The iteration of Newton's method for multiple variables reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(D\mathbf{F}(\mathbf{x}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{x}^{(k)}),$$

where $D\mathbf{F}$ is the Jacobian $D\mathbf{F}(\mathbf{x}) = \begin{pmatrix} 3x^2 - 3y^2 & -6xy \\ 6xy & 3x^2 - 3y^2 \end{pmatrix}$.

(4c) Denote by $\mathbf{x}^{(k)}$ the iterates produced by the Newton method from the previous sub-problem with some initial vector $\mathbf{x}^{(0)} \in \mathbb{R}^2$. Depending on $\mathbf{x}^{(0)}$, the sequence $\mathbf{x}^{(k)}$ will either diverge or converge to one of the three cubic roots of unity.

Analyze the behavior of the Newton iterates using the following procedure:

- use equally spaced points on the domain $[-2, 2]^2 \subset \mathbb{R}^2$ as starting points of the Newton iterations,
- color the starting points differently depending on which of the three roots is the limit of the sequence $\mathbf{x}^{(k)}$.

HINT:: useful MATLABcommands: `pcolor`, `colormap`, `shading`, `caxis`. You may stop the iteration once you are closer in distance to one of the third roots of unity than 10^{-4}.

The three (non connected) sets of points whose iterations are converging to the different z_i are called Fatou domains, their boundaries are the Julia sets.

Solution: For each starting point at most `N_it` iterations are accomplished. For a given starting point $\mathbf{x}^{(0)}$, as soon as the condition $|x_1^{(k)} + ix_2^{(k)} - z_i| < 10^{-4}$, $i \in \{1, 2, 3\}$ is reached, we assign a color depending on the root z_i and on the number of iterations done. Each attractor is associated to red, green or blue; lighter colors correspond to the points with faster convergence. The points that are not converging in `N_it` iterations are white. We set the color scale using the MATLABcommands `colormap` and `caxis`.

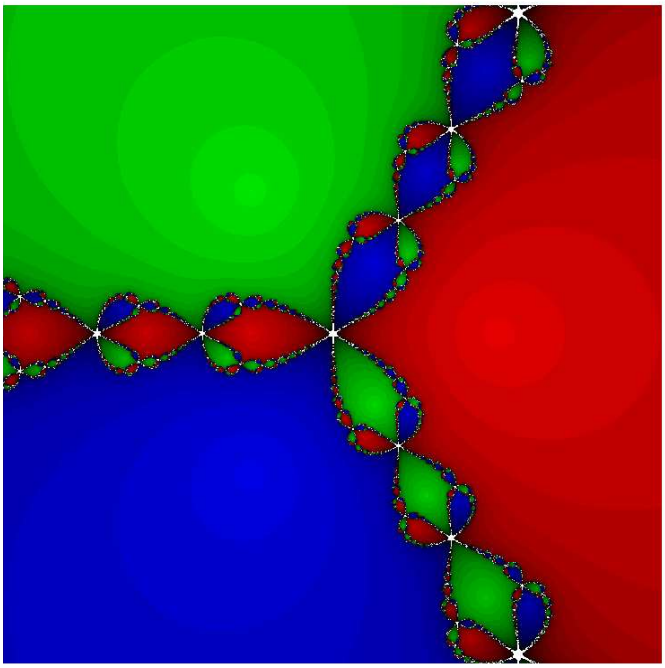

Figure 12: Julia set for $z^3 - 1 = 0$ on a mesh containing 1000×1000 points, `N_it`=20.

Listing 33: `julia_set.m`

```
function julia_set(N_it, num_grid_points)
```

```
close all;
if nargin<2;   N_it = 25;  num_grid_points = 200; end;

% initialize matrix for colors
col = ones(num_grid_points);
a = 2.0;
xx = linspace(-a,+a,num_grid_points);
yy = linspace(-a,+a,num_grid_points);

% roots of unity:
z1 = [ 1;     0];
z2 = [-1/2;  sqrt(3)/2];
z3 = [-1/2; -sqrt(3)/2];

for ny = 1:length(yy)
    for nx = 1:length(xx)
        % for each starting point in the grid
        v = [xx(nx); yy(ny)];

        for k=1:N_it;
            F  = [v(1)^3-3*v(1)*v(2)^2-1;
               3*v(1)^2*v(2)-v(2)^3];
            DF = [3*v(1)^2-3*v(2)^2, -6*v(1)*v(2);
                  6*v(1)*v(2)       , 3*v(1)^2-3*v(2)^2];
            v = v - DF\F;            % Newton update

            if      ((norm(v-z1)) < 1e-4)
                col(ny,nx) = 1 + k;
                   break;
            elseif  ((norm(v-z2)) < 1e-4)
                col(ny,nx) = 1 + k + N_it;
                   break;
            elseif  ((norm(v-z3)) < 1e-4)
                col(ny,nx) = 1 + k + 2*N_it;
                   break;
            end
        end
    end
```

```
end
st = 1/N_it;          % build a RGB colormap,
% 1s at the beginning to recognize slowly-converging points (white)
mycolormap = [ [1,1-st:-st:0, zeros(1,2*N_it)];
               [1,zeros(1,N_it), 1-st:-st:0, zeros(1,N_it)];
               [1,zeros(1,2*N_it), 1-st:-st:0] ]';
% mycolormap = 1 - mycolormap;     % cmy, pyjamas version...
% mycolormap = [ [1,1-st:-st:0, st:st:1,    st:st:1,];
%                [1,st:st:1,    1-st:-st:0, st:st:1,];
%                [1,st:st:1,    st:st:1,    1-st:-st:0] ]';
colormap(mycolormap);             % built in: colormap jet(256);
% this is the command that creates the plot:
pcolor(col);
caxis([1,3*N_it+1]); shading flat; axis('square','equal','off');
print -depsc2 'ex_JuliaSet.eps';
```

Issue date: 15.10.2015

Hand-in: 22.10.2015 (in the boxes in front of HG G 53/54).

Version compiled on: October 23, 2015 (v. 1.0).