# Fall 2016

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

2)

Have look at eq (2), p. 427 in book.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

3)

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

2)

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

3)

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

2)

Superlinear, and converges as:

Et billede, der indeholder tekst, person, skærmbillede

Automatisk genereret beskrivelse

You need to differentiate the gradient-function:

x0 = 120;%starting guess

n = 4; %number of iterations

w = (2\*pi/360);

x\_holder = zeros(numel(n),1);

for i=1:n

f = -40.49\*w\*cos(w\*x0)-143.16\*w\*sin(w\*x0); %define function

df = 40.49\*w^2\*sin(w\*x0)-143.16\*w^2\*cos(w\*x0);

if abs(df)<eps %smaller than machine epsilon to prevent underflow

error("small derivative");

end

d = f/df; %calculate fraction

x0 = x0-d; %calculate new root

x\_holder(i,1) = x0;

end

found\_roots = sprintf('%.0f\n',x\_holder)

Yielding

'176

164

164

164

Et billede, der indeholder bord

Automatisk genereret beskrivelse

2.1)

Are the formal 2x2 normal equations.   
One can construct the matrix formulation as follows, using the monomialbase

Giving the normal-equation:

The system matrix is

Which is the same as in the formal normal-equations.   
The right hand side becomes:

Both values are easily calculated by

lhs = transpose(A)\*A

rhs = transpose(A)\*y

**2.2)**

A=[exp(0),exp(0);exp(-0.5),exp(-1);exp(-1),exp(-2);exp(-1.5),exp(-3);exp(-2),exp(-4)];

y = [2.24,0.80,0.38,0.09,0.06];

y = y';

c = (A'\*A)\(A'\*y)

Giving

c = [a;b]=[0.0751;2.1572]

2.3)

The error is

A=[exp(0),exp(0);exp(-0.5),exp(-1);exp(-1),exp(-2);exp(-1.5),exp(-3);exp(-2),exp(-4)];

y = [2.24,0.80,0.38,0.09,0.06];

y = y';

c = (A'\*A)\(A'\*y);

xx = linspace(0,2,5);

yy = (c(1).\*exp(-xx)+c(2).\*exp(-2.\*xx))'

error = abs(y-yy)

comb\_error = sum((y-yy).^2); %%this is squared-error

This yields the output:



Which in significant decimals can be written as:

With 2 significant decimal digits.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

The only difference between the two methods are the coefficients. In Heun:

In RK variant:

Et billede, der indeholder tekst, skærmbillede

Automatisk genereret beskrivelse

Page 313:

The only term needed to change is thus alpha:

Which only implies changing in line 23 to:

And we have the variant.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

We need to fullfill:

Second condition is not fulfilled, if one only looks at equation 9.   
Lets have a look at eq. 7 vs. eq 8:

>>shows that we do miss-hit the first -term.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Using the same two equations:

We know, that the function is independent on t as , which implies that , we can substitute by :

In our situation giving:

Which is completely the same as the taylor-expansion if is set to 0:

Thus, they match up to and including .

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Same exercise:

We know, that the function is independent on x as , which implies that , we can substitute by :

Which obviously only is a valid approximation up to the ’th term, as there is a discrepancy in the second term, as the Taylor expansion is:

Thus, we would undershoot a bit if we were to use higher order terms.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

End-point is given as 5.

dxdt =@(t,x)(1/2)\*x; %%HUSK TO PARAMETRE SELVOM FUNKTIONEN IKKE ER FUNKTION AF TO PARAMETRE I OPGAVEN

f\_analytical = @(t)exp(t/2);

tspan = [0 5];

x0 = 1;

n = [10,20,40,80,160];

x\_sol = zeros(numel(n),1);

x\_real = f\_analytical(5);

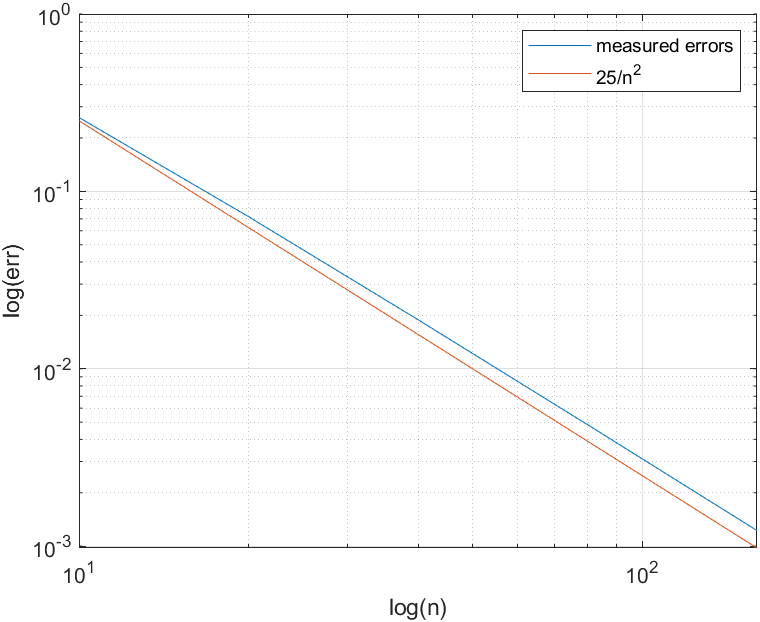
for i = 1:5

[t,xf]=RKvariant(dxdt,tspan,x0,n(i));

x\_sol(i) = xf(end);

end

Q1absErr = [n;abs(x\_real-x\_sol)']

Opg 2  


Når den er lineaær og aftagende, så kan vi i hvert fald se, at der er tale om .

Man kunne også vise det dovent ved at

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Hvilket er \*nogenlunde\* konstant, hvilket viser at konvergensen er bundet af

Et billede, der indeholder tekst

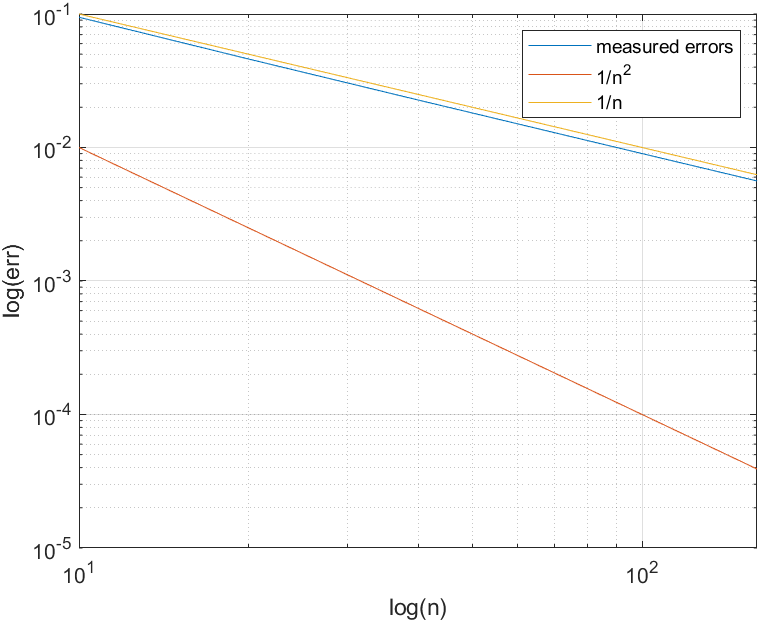
Automatisk genereret beskrivelse

Q1absErr =

10.0000 20.0000 40.0000 80.0000 160.0000

0.0945 0.0460 0.0227 0.0113 0.0056





convergence = Q1absErr(2,:).\*n

convergence =

0.9454 0.9204 0.9079 0.9017 0.8985

Almost constant – thus converges as .

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

The vector-2-norm is defined as the Euclidean distance. We can use the worst-case scenario given:

This exemplifies the general vector-norm-property:



We have the general matrix-vector property regarding the norm:

We have the system:

We also have for the perturbation in **:**

We can write the results together, thus obtaining the definition of the relative error:

Using the definition of the condition-number:

Thus, the upper bound numerically becomes:

**Assignment about factorizations**

**Et billede, der indeholder tekst

Automatisk genereret beskrivelse**

**1)**

**Jacobian:**

The matlab-code thus becomes

function [f,dF] = funFdF(X)

%%input are variables, output is function matrix and jacobian matrix for

%%system described in 23.1

f = [X(1)^2+2\*X(2)-2;X(1)+4\*X(2)^2-4];

dF = [2\*X(1),2;1,8\*X(2)];

end

**2)   
The Newton-iteration for finding the roots may be written as:**

Finding the inverse of the jacobian is numerically very unstable, and also a very slow operation. Thus, we rewrite, as the Taylor-expansion gives us:

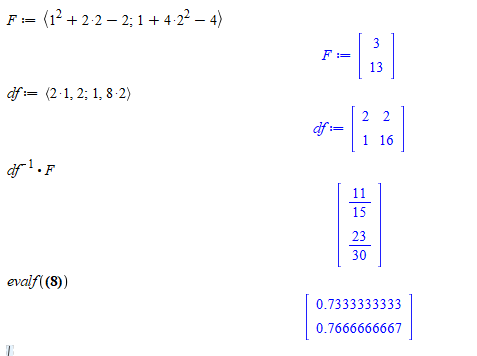
Which means that we can find the solution to the Jacobian Linear System:

Which is the same as finding the solution to the system **, ie.**

This yields the inverse of the Jacobian with numerical stability. This is implemented using mldivide \. The final iteration thus becomes (by substitution of H):

**3) Implementing the iteration:**

1. Finding goal values of iteration 1:



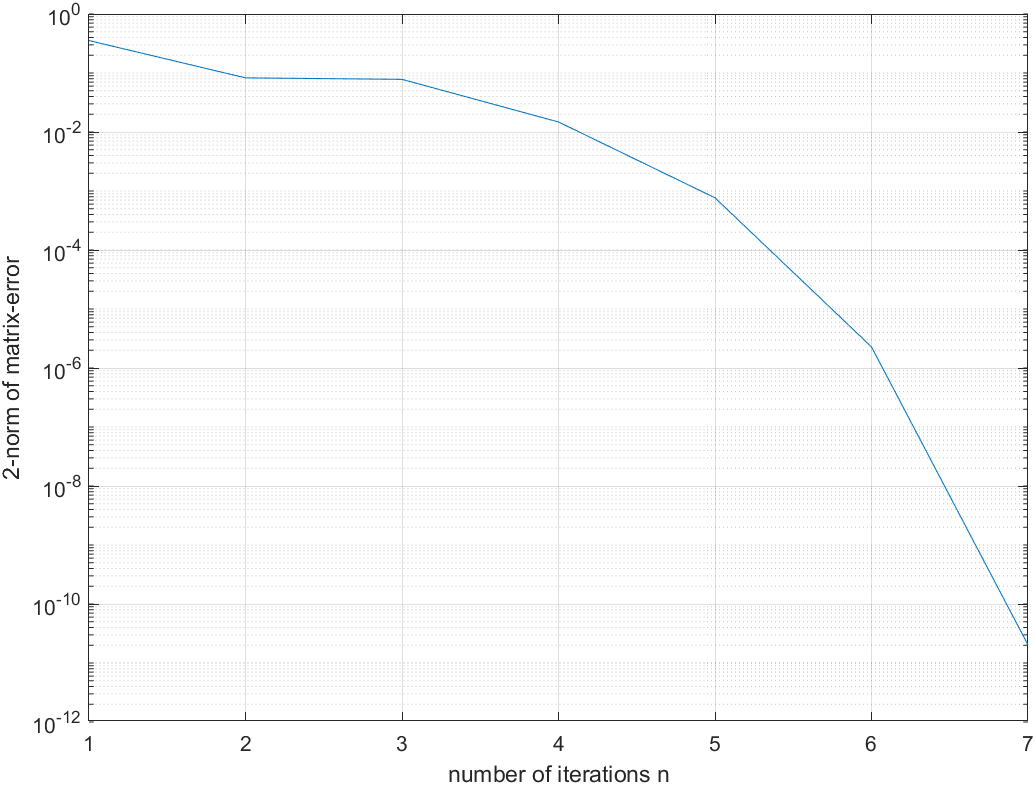
This implementation seems fit:



Giving the results:

4)   
Including the errors in the implementation. It is important to remember, that we are working with vector-norms, ie no abs error.





Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

1)   
It must be completely real, symmetric, and positive definite. Symmetric means . Positive definite means that all eigenvalues are positive – but more properties do describe positive definitiveness.

2)   
The Cholesky factorization itself uses for the outer loop:   
For each er der multiplikationer (da ikke giver nogen multiplication), og . Det giver samlet set:

Multiplikationer og divisioner i yderløkken (du kan beregne den variable sum med maple, EZ-PZ). Endvidere er der rødder og subtraktioner.   
I inderløkken har du principielt set samme arbejdsbyrde, men det skal ganges med yderløkken:

Multiplikationer:

Og det samme for subtraktionerne. ved jeg ikke rigtigt, hvor kommer fra.   
Det giver dog samlet set den dominante term

Hvilket tidsmæssigt er .

Strategi 1 ville så have:

Strategi 2 ville kun kræve i tillæg for at beregne , da det er en lower-triangular, hvilket vil sige at den maksimalt har elementer:

Hvilket er en del mindre. De er dog begge .

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

It is given that it is symmetric, and thus we may apply LU-factorization.

|  |  |
| --- | --- |
|  | (1) |
|  | (2) |
|  | (2) |

Now, we wish to solve:

First we find (for using fwd-sub)

Next row:

Thus, the factorization becomes:

Which shows, that it is a valid factorization. Unlimited many other factorizations exist aswell, though.

**2) Check if you can use LDL factorization** is symmetric and positive definitive (eigenvalues 3,1). Thus, a -factorization must exist.

Sadly, I made a Crout-factorization before for doing LU. LU factorization using Doolittle:

Applying LDLT-factorization using the unit lower triangular:

Extract diagonal elements from U and putting into D and writing transpose L:

(Which is the same mathematically as finding the solution to):

Thus, we have:

Which shows, that has a -factorization.

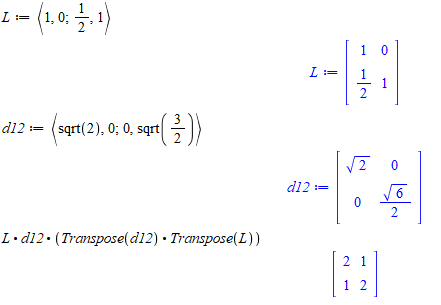


We know it is positive definitive, and symmetric. Thus, it must have a Cholesky factorization.

Thus, we simply need take the squareroot of earlier found D:

Giving us:

With .



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

from the definition of the Euclidean norm:

We know that the maximal error is , and thus this 2-norm is our upper-bound:

We have the value times, and we know that . Thus we may write

The upper bound of the relative error may be found using the newly identified upper bound for the norm of the measurement-matrix. We want to figure out:

We know from properties of vector-norms that:

Which means we may write:

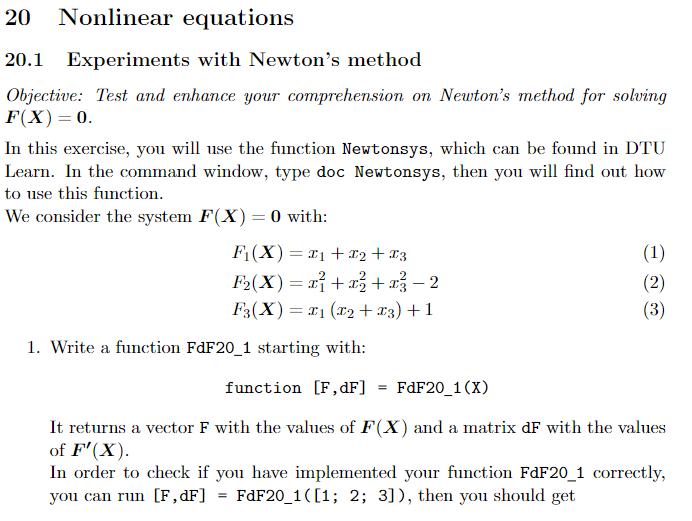
Yielding

Introducing the perturbation in b:

Thus we can write the relative error:

Thus, we have all information for finding a value for the upper bound of the relative error:

With two significant digits.



X0 = [3/4;1/2;-1/2];

n = 10;

X = X0;

Xiter = zeros(n,numel(X));

for i=1:n %%newton method

[F,dF] = FdF20\_1(X);

H = dF\F;

X = X-H;

Xiter(i,:) = X;

end

plot(Xiter)

function [F,dF] = FdF20\_1(X)

F = [X(1)+X(2)+X(3);X(1)^2+X(2)^2+X(3)^2-2;X(1)\*(X(2)+X(3))+1];

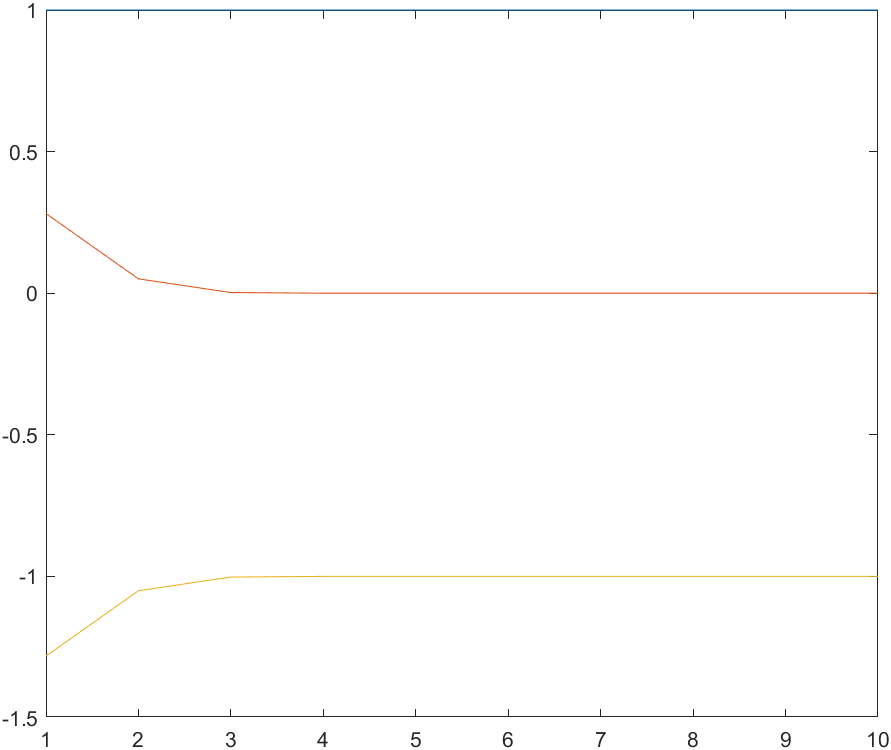
dF = [1,1,1;2\*X(1),2\*X(2),2\*X(3);X(2)+X(3),X(1),X(1)];

end

is the function.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse



Converges to



We get

And

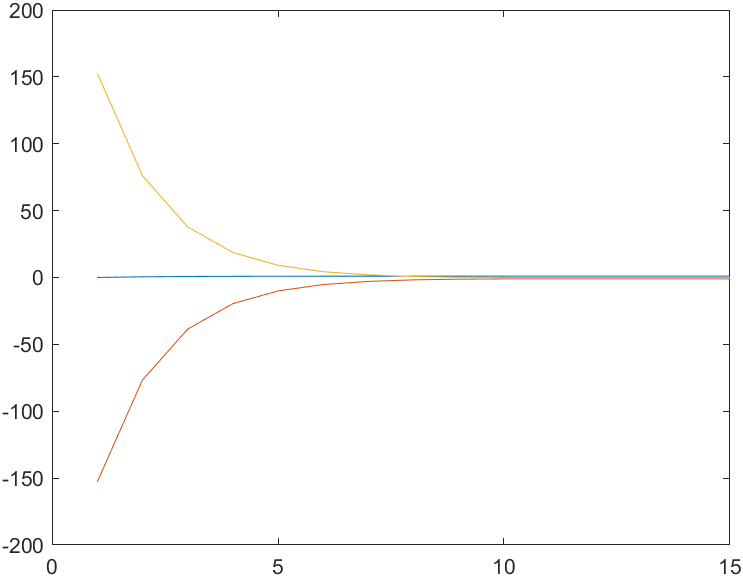
Finding for the first iteration:

Solving this system gives us problems. Why? Because the Jacobian is singular:

Which means that it does not have an inverse matrix (this would be the same as dividing by zero). Thus, Newtons method fails for this system, as it is based on the Jacobian being non-singular.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse



Converges towards:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Building upon earlier code:

%%error-assesment

%error-assesment

err = zeros(n,1);

for k=1:n %%newton method

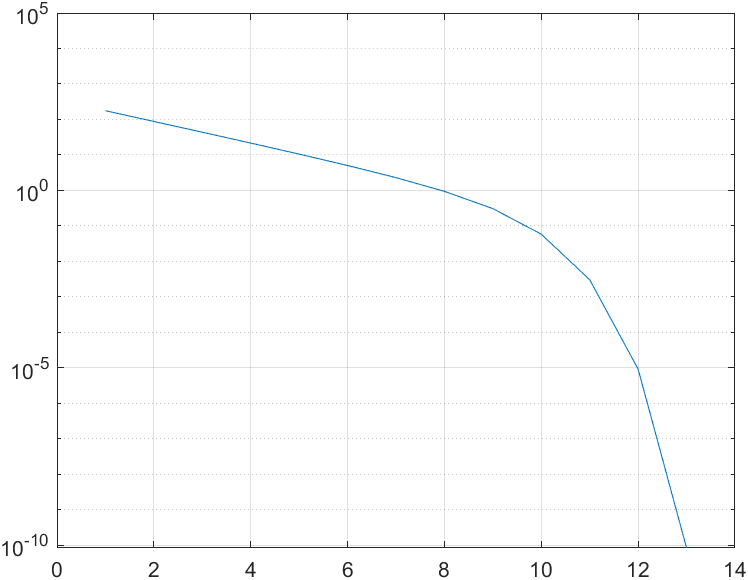
err(k) = max(abs(Xiterations(:,end)-Xiterations(:,k)));

end

figure(2)

semilogy(1:n-1,err(1:end-1));

grid('on')



Shows that we dip beyond 10e-6 at 13 iterations.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Finding the Jacobian using Maple as it is quite large. Notice it is only a function of xy

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Debugger ved at sidetjekke med Maple og bruge subs.



Det spiller.

2)

X0 = [2,2]';

FdF = @(X) FdF20\_2(X,0);

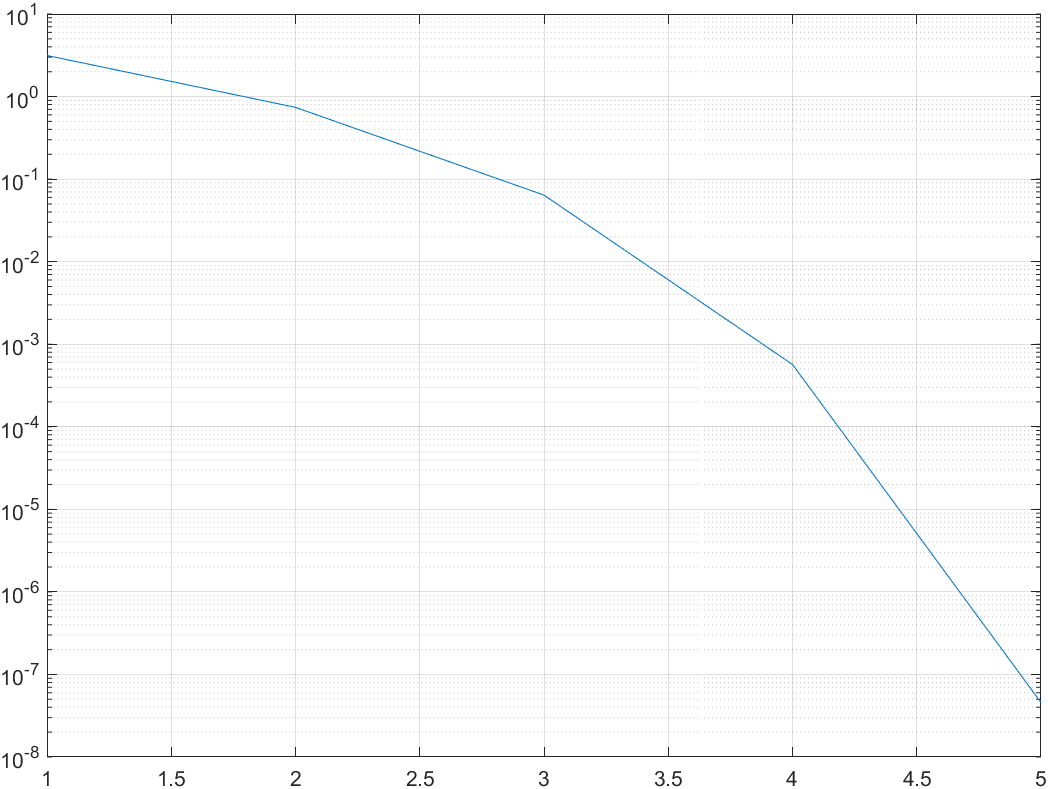
Xiter = Newtonsys\_facit(FdF,X0,5);

for i = 1:5

err(i) = norm(4-Xiter(:,i));

end

Which shows us, that the implementation is fine, we get and need 4 iterations to dip under distance from .



3)

The following code is implemented:

X0 = [2,2]';

FdF = @(X) FdF20\_2(X,0);

Xiter = Newtonsys\_facit(FdF,X0,5);

%Q3

omega = linspace(0,2\*pi,361);

X0 = Xiter(:,4); %numerical solution to omega = 0 ie first point with error smaller than 10e-3. Thus very nice start-guess -> only one iteration needed for every point

position = zeros(2,numel(omega)-1); %placeholder for rest of solutions

position(:,1) = X0; %%first point has been calculated

for i=2:numel(omega) %thus no need for omega(1)=0.

FdF = @(X) FdF20\_2(X,omega(i)); %generate function-handle for new root-guess and omega-value

k = 2;

Xiter = Newtonsys\_facit(FdF,position(:,i-1),k); %save results in 2x361 matrix with format [x;y]. Use last found position

position(:,i) = Xiter(:,end);

%%X = position(:,i); %overwrite root-guess

end

plot(position(1,:),position(2,:)) %parametrized plot for P3 = (x,y)

title('P\_3(\omega)')

xlabel('x-val')

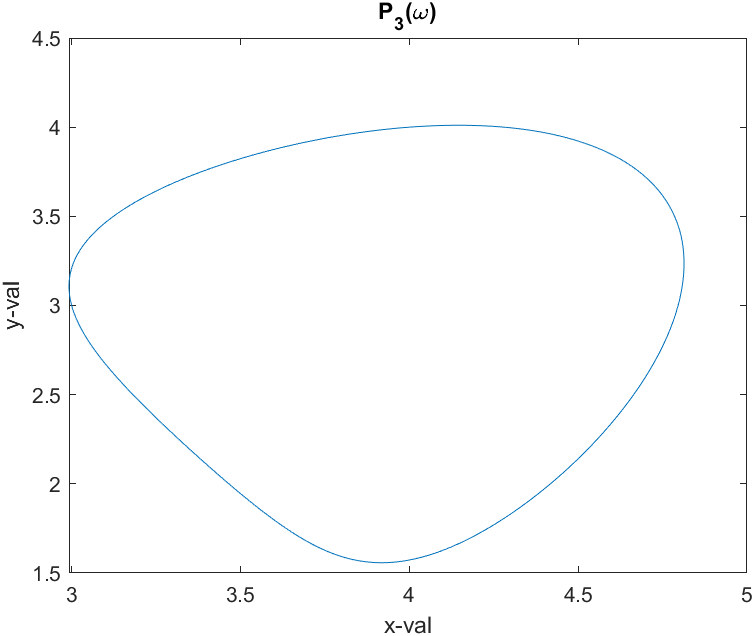
ylabel('y-val')

%Q4 distance calculation

Xanalytical = [4;4];

dist = norm(Xanalytical-position(:,end));

Which gives us:



4)

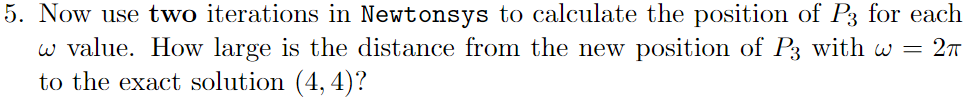
The distance is easily found:

%Q4 distance calculation

Xanalytical = [4;4];

dist = norm(Xanalytical-position(:,end));

Giving .



In the grand scheme the plots look the same. The distance, though, is only ie way better.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

2) which is the same as solving for :

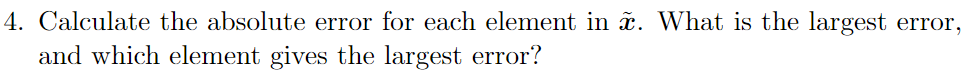
A = [3,12,10;12,0,20;0,2,30];

b = [72.3;99.5;56.6];

weights = A\b

Which gives us

3) System with perturbation gives

****

A = [3,12,10;12,0,20;0,2,30];

b = [72.3;99.5;56.6];

b\_err = [73.3;98.4;57.1];

weights\_err = A\b\_err

weights = A\b;

abs\_err = abs(weights-weights\_err)

[val,worst\_element] = max(abs\_err)

Gives us abserr:

With worst error-contributor being screws.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

rel\_rhs = norm(b\_err-b)/norm(b)

rel\_lhs = norm(weights\_err-weights)/norm(weights)

kappa = cond(A)

rel\_rhs =

1.1584e-02

rel\_lhs =

2.2536e-02

kappa =

3.9080e+00

Theoretical upper bound for error is thus:

Thus, the actual measured relative error:

Is almost exactly half the size of the upper bound.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

We know for Matrix-norms:

We also have

Thus, the fraction becomes:

Remember that these are norms, thus numbers and no longer matrices. Thus, we have commutative multiplication:

The division between B’s is only possible as

Which makes this an upper bound to the whole error-propagation.

Which we were requested to show.   
It needs to be noted that we use one specific property for :

Which makes us able to do the division.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

We wish the upper bound to be 0.02.   
We can easily find condition-number of our matrix A. Thus, we simply isolate:

Which seems a bit large.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

We make random matrix and designate nE=norm(E). Thus, we now know:

Thus, we need to rescale our random noisy-matrix:

Thus, we scale by this factor, and then add :

X2 = A\(B+E)/A;

figure(2)

imagesc(X2), axis image off, colormap gray

rel\_error = norm(X-X2)/norm(X);

Which gives us the relative error , which is way smaller than the upper bound.

Whole script:

load deblur.mat %%import matrix-data

[L,U,P] = lu(A);

X = U\(L\B/U)/L;

figure(1)

imagesc(X), axis image off, colormap gray

faxi\_condi = cond(A);

blurring\_boundary = (0.02/(faxi\_condi^2))\*norm(B);

E = randn(size(A));

nE = norm(E);

E = E\*blurring\_boundary/nE;

nE = norm(E);

X2 = A\(B+E)/A;

figure(2)

imagesc(X2), axis image off, colormap gray

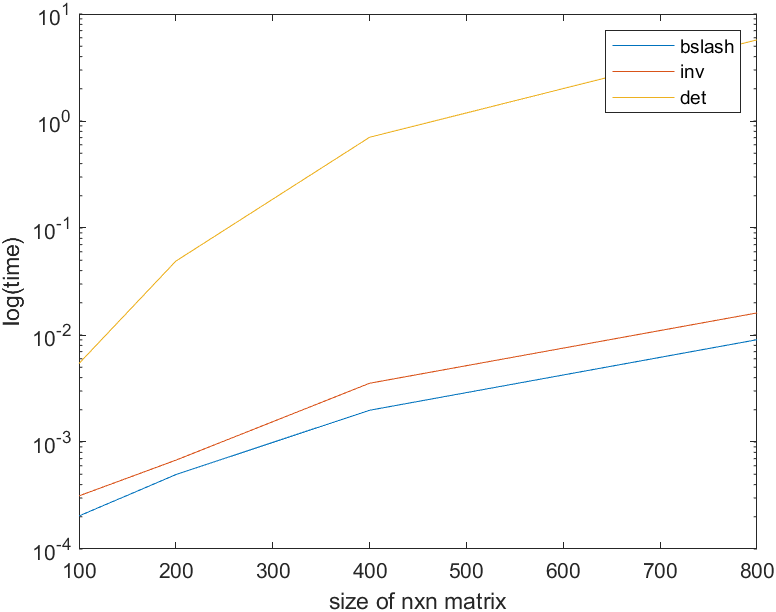
rel\_error = norm(X-X2)/norm(X);

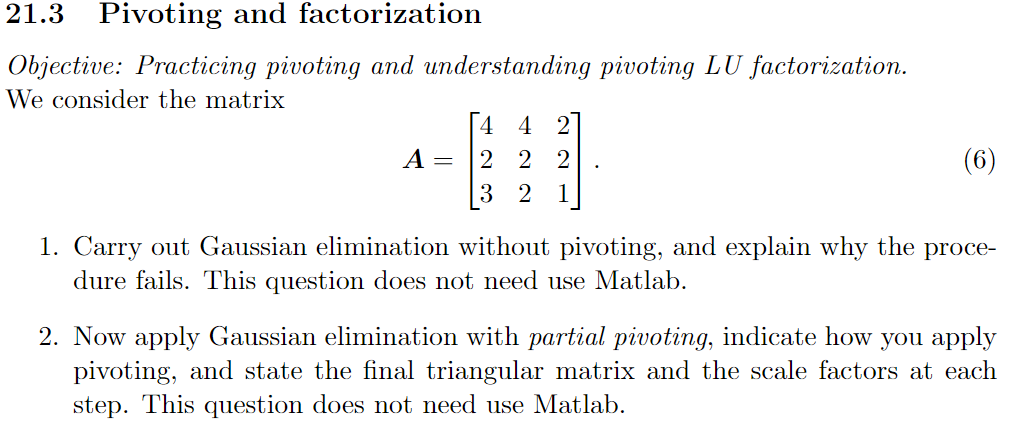
Et billede, der indeholder tekst

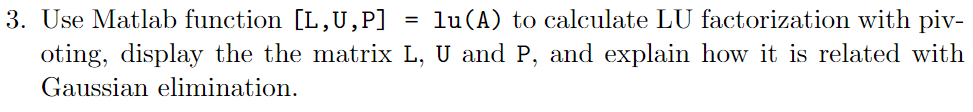
Automatisk genereret beskrivelse



Different for each run, but in general \ is a bit faster. Determinant-time is in general one to two orders of magnitude slower. At n=800, \ is almost twice as quick as inv time, and times faster than determinant-method.







1)

Which clearly shows, that we are not able to carry out the elimination without pivoting until the end. Gauss-elimination is the same as doing a Doolittle LU-factorization:

Thus, we know that:

Which means we can calculate the last row:

Hvilket ikke kan lade sig give sig, da . Derfor fejler metoden altså, og systemet kan ikke LU-faktoriseres.

We are simply unable to get a solution. This was in effort of trying to get the reduced row echelon form. I could instead try and get an upper triangular (echelon form):

Which is doable if one uses pivoting (which is the row-substitution between 2nd and 3rd row).

In LU-factorization terms this would be equivalent to finding a factorization to the system:

Which now gives:

Accordingly for the last row:

And thus the system could be factored:

Therefore, using partial pivoting makes it possible. Writing the pivoting-operation in matrix-form:

Reducing first column using

Which thus is in triangular (echelon) form. Thus, the elimination has succeeded.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

We see:

And

Thus, it is the same operation in opposite direction 😊

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst, person, skærmbillede, dokument

Automatisk genereret beskrivelse

1)   
So we should use different interpolation forms from 0-degree polynomial up to n-1-degree polynomial.

2)

n = [1:12];

for i=n

v = linspace(0,1,i);

M = vander(v);

condition\_number(i) = cond(M);

end

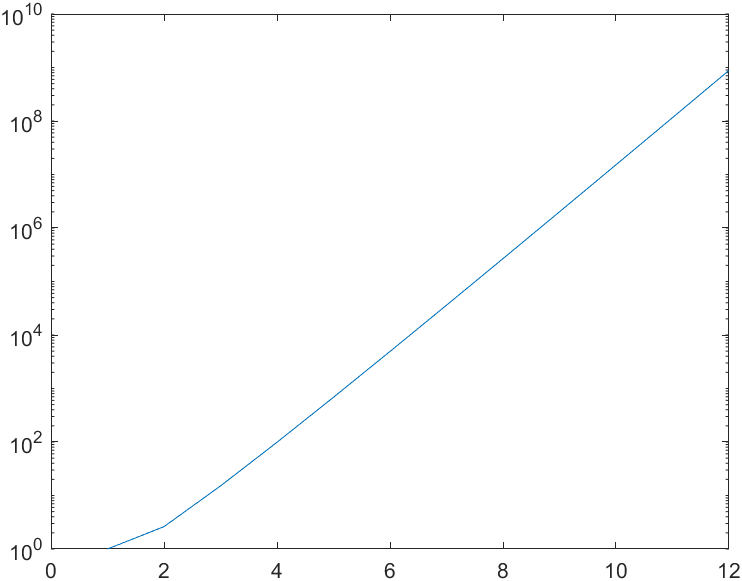
semilogy(n,condition\_number);

%%datafitting part

A = [n',ones(numel(n),1)];

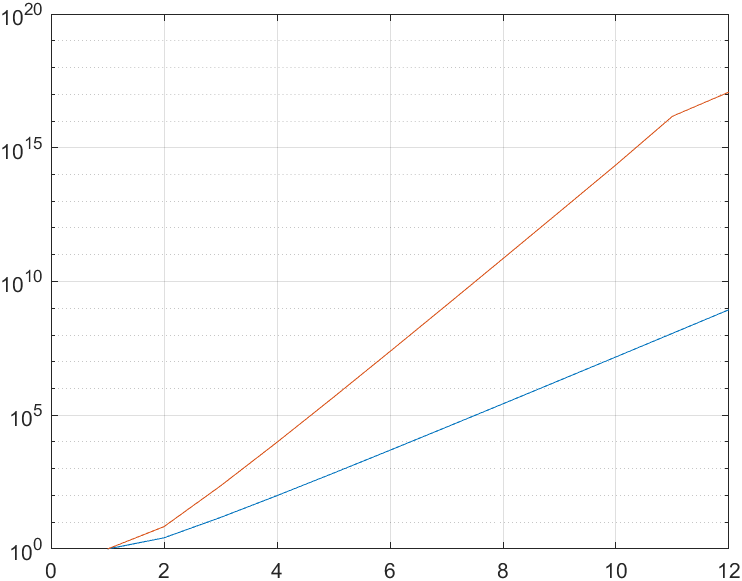
logC = log10(condition\_number');

c = (A'\*A)\(A'\*logC)



3)   
Yes, the Vandermonde matrix is defined as the coefficient matrix , and thus , which is the definition of the system matrix.

4)



It seems that

5)

1.0000e+00

2.6180e+00

1.5100e+01

9.8868e+01

6.8643e+02

4.9244e+03

3.6061e+04

2.6782e+05

2.0094e+06

1.5193e+07

1.1558e+08

8.8348e+08

Thus, we in the end expect to loose 8 significant digits -> 15-8 = 7 sig.dig. left when solving the linear system.   
In the normal equation, we expect to loose approximately digits ie we have overflow. At we only have significant digit left when doing the data-fitting (solving the normal-equation).

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

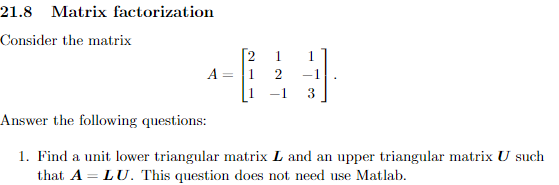
1) All eigenvalues are real and positive.

2)   
Siden at B er symmetrisk, så: , hvorved at

Hvilket betyder, at egenvektoren for de to submatricer må være ens. Siden at hver undermatrix er symmetrisk, er produktet det nødvændigvis også, og det opfylder betingelsen af at have samme linearitet som matricen , dvs. det er blot en skalering af . Herfor kommer der til at stå:

3)   
Vi ved nu, at er egenvektor til B med eigenværdi , og egenvektor til med værdi . Herved ved vi, at to-normen

Endvidere ved vi at (fra at vi viste at egenværdien er i anden i 2)). Derfor er det tydeligt at det er opfyldt.



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

1)

Giving:

2) Having the LU-factorization, this is quite easy:



Samme omgang piss om sidst, blot hvor du starter med en unitary diagonal i U.

Ender med at blive:

Et smartere træk havde været at bemærke, at er en symmetrisk matrix, hvorved at . Thus I can just use earlier result.



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

**1. Finding all derivatives:**

Thus, finding:

**Order 1: Eulers method**

With initial-condition:

Order 2:

With initial-condition:

Order 3:

Order 4:

7.1.13)   
Suppose that a diffeq is solved numerically on interval a,b and the local truncation error is ch^p. Show that if all truncation errors have the same sign (worst possible case), then the total truncation error is (b-a)ch^(p-1).

We know, that the local truncation is

And we have steps between . Thus, we will have to multiply by :

We can do some mathemagic and expand the power factor:

7.1.9)   
Consider the problem x’=x. If the initial condition is x(0)=c, then the solution is x(t)=c\*exp(t). If a roundoff error of epsilon occurs in reading the value of c into the computer, what effect is there on the solution at the point t = 10? At t=20? Do the same for x’=-x

The absolute error will look like:

Thus, the accumulation will result in errors:

Thus they increase as a power of . The relative error though:

Is a constant <3

In the other case, we have the solution :

Thus, the error here actually decreases as a power:

But the relative error again is the same.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

function [t,x] = MyTaylorOrder2(diffeq1,diffeq2,tspan,x0,n)

b = tspan(2);

a = tspan(1);

h = (b-a)/n;

x = zeros(1,n+1);

x(1,1) = x0;

t = (linspace(a,b,n+1));

for i=1:n

dval1 = diffeq1(t(i),x(1,i));

dval2 = diffeq2(t(i),x(1,i));

x(1,i+1) = x(1,i)+h\*(dval1+(1/2)\*h\*dval2);

end

end

This boy does well.   
Shoots a bit under the analytical solution, but is very close.

Is called by using:

X1 = @(t,x)t+x;

X2 = @(t,x)1+t+x;

n = 1;

t = [0,1/2];

x0 = 1;

[tvals,xvals] = MyTaylorOrder2(X1,X2,t,x0,n)

Thus, we need to know both derivatives for using this method, ie. 1 differentiation before starting and two function evaluations in every iteration.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder bord

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

So for each doubling of , we expect the global error to be divided by . Lets have a look:

holder = err\_abs/(2^2)

holder =

0.00084415

0.00021608

5.4658e-05

1.3745e-05

3.4464e-06

It almost fits perfectly. Also if you plot it it is beautiful.

X1 = @(t,x)t+x;

X2 = @(t,x)1+t+x;

n = [16,32,64,128,256,512];

t = [0,1];

x0 = 1;

error\_holder = zeros(numel(n),1);

for i=1:numel(n)

[tvals,xvals] = MyTaylorOrder2(X1,X2,t,x0,n(i));

error\_holder(i) = xvals(end);

end

fanalytical = -2+2\*exp(1);

err\_abs = abs(fanalytical-error\_holder);

expected\_reduction = err\_abs/(2^2);

expected\_reduction\_matrix = [n(2:end);expected\_reduction(1:end-1)']

semilogy(n,err\_abs)

hold on

semilogy(n(2:end),expected\_reduction(1:end-1))

legend('absolute error','expected error reduction')

Et billede, der indeholder tekst

Automatisk genereret beskrivelse



It seems the relative error is quite constant, as expected. But if one looks closely, we observe a small linear trend. As expected, the absolute error increases as a function of (that is why it is linear in semiology).   
The growing relative error must be caused by a combination of local truncation and rounding.



3) with n=10000



We see that the error acts in the same way (the curves have the EXACT same shape). We see that the rounding introduced in the relative error has been minimized a bit. Also, the absolute error grows less quickly. This also explains the vertical translation of the error-functions between the two cases.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

1) function becomes:

function [t,x] = MyRK4(dxdt,tspan,x0,n)

a = tspan(1);

b = tspan(2);

h = (b-a)/n;

%%x = zeros(1,n+1);

x = x0;

%%t = zeros(1,n+1);

hhalve = h/2;

x = zeros(1,n+1);

t = linspace(a,b,n+1);

x(1,1) = x0;

for i=1:n

K1 = h\*dxdt(t(i),x(1,i));

K2 = h\*dxdt(t(i)+hhalve,x(1,i)+(1/2)\*K1);

K3 = h\*dxdt(t(i)+hhalve,x(1,i)+(1/2)\*K2);

K4 = h\*dxdt(t(i)+h,x(1,i)+K3);

x(1,i+1) = x(1,i) + (1/6)\*(K1+2\*K2+2\*K3+K4);

end

2)   
tspan = [0,1];

dxdt = @(t,x)t+x;

x0 = 1;

fanalytical = @(t)-t-1+2\*exp(t);

n = 2.^[4:9];

teo = fanalytical(tspan(2));

error\_holder = zeros(3,numel(n));

for i = 1:numel(n)

[t\_rk4,x\_rk4] = MyRK4(dxdt,tspan,x0,n(i));

[t\_heun,x\_heun] = MyHeun(dxdt,tspan,x0,n(i));

[t\_euler,x\_euler] = MyEuler(dxdt,tspan,x0,n(i));

error\_holder(1,i) = abs(teo-x\_rk4(end));

error\_holder(2,i) = abs(teo-x\_heun(end));

error\_holder(3,i) = abs(teo-x\_euler(end));

end

semilogy(n,error\_holder)

legend('rk4','heun','euler')

xlabel('number of steps n')

ylabel('log(abs(err))')

figure(2)

loglog(n,error\_holder)

%%convergence for rk\_4: check if following gives appx constant

disp('convergence of RK4\n')

error\_holder(1,:).\*n.^4

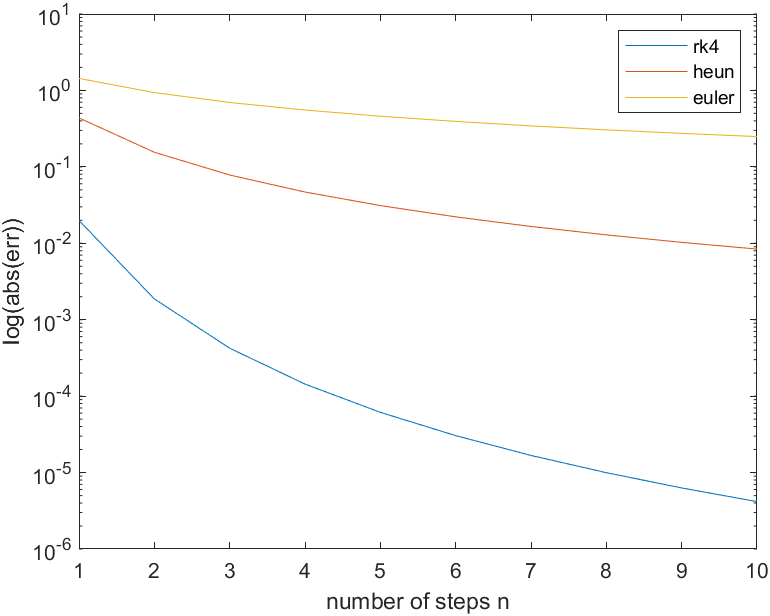
disp('convergence of heun\n')

error\_holder(2,:).\*n.^2

disp('convergence of euler\n')

error\_holder(3,:).\*n.^1

3)



For checking convergence, you just check if following is constant:

So if it gives a constant, you have that type of convergence. Also, the error fullfills . Thus, after you have found the convergence, you can say what the error is appx.   
You find that the convergence for euler is , heun and RK4 .

By running the script you get the appx value of .

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Here is an initial-value problemthat has two solutions: , . Verify that the two solutions are x1(t)=0 and x2(t) = (t\*2/3)^(3/2) for t>=0. What happens if you use Taylor series method?

To verify, I just put in the solution.   
Case 1:

Which obviously is true and fullfills .   
Case 2:

Now, we check if this is the same as :

Taking also seems true, so the two functions must be a solution.

If one were to apply the taylor series method of degree 2:

With

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

**(MULTIVARIATE CHAIN RULE)**

****

We would get:

Which is undefined. **Thus, Taylors method would fail.**

If one used RK4:

Thus we end up with:

Ie RK4 will not fail, but find the zero-solution.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse



Thus, the solution to this IVP is:





function [dF] = odefun(t,x)

dF = [-x(1)+x(2)+sin(t);-x(1)-x(2)];

end

Et billede, der indeholder tekst

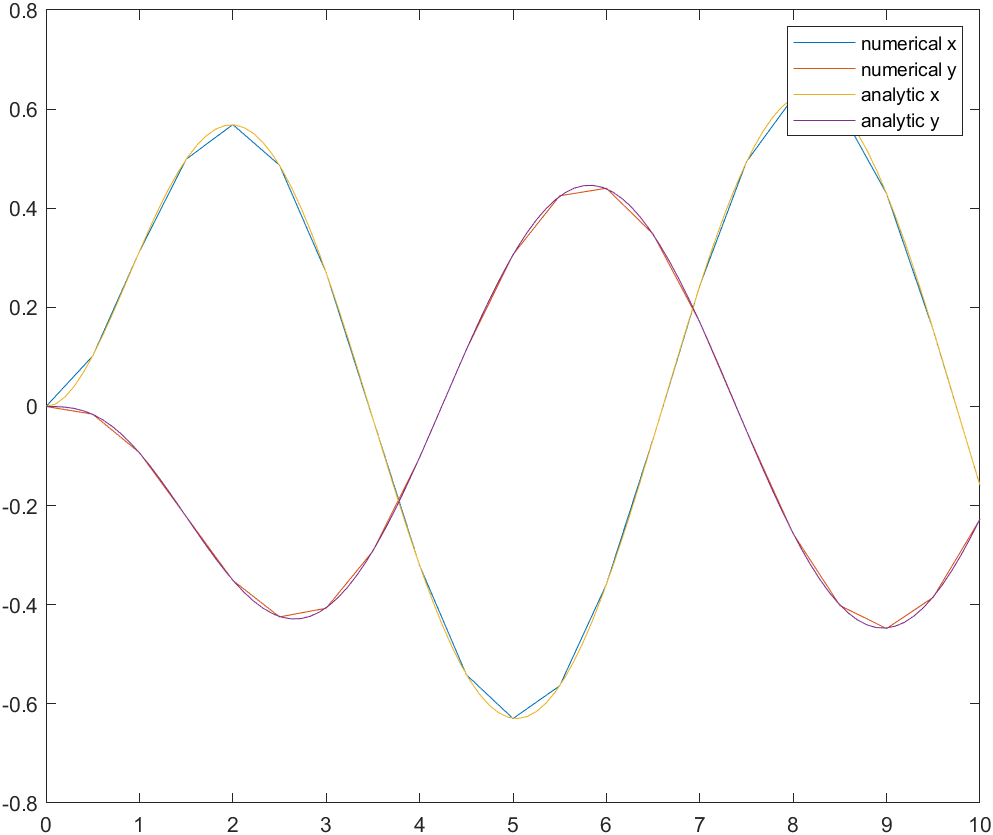
Automatisk genereret beskrivelse

It knows how many diffeqs there are from the size of the inputvector x0.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse











The errors are quite small. I think this is definitely satisfactory.



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

The crash-test gives huge errors. This is because we are effectively undersampling, and the functions change quickly compared to the samplerate.

We get satisfying results using appx 150 sample-points. We have this many periods

And sample times during these periods, giving us

times per period of .

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

154 successful steps = 154 iterations. The accuracy is quite bad. This is probably because we did not specify a tolerance range (which we could do). Ode45 is practical because we don’t need to know beforehand how many n we need for a nice precision.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

How would a program be used to solve:

We would linearize by introducing new variables. This will be the dictionary for the problem:

|  |  |  |  |
| --- | --- | --- | --- |
| Old variable | New variable | Initial Value | Differential equation |
|  |  | 3 |  |
|  |  | -7 |  |
|  |  | 4 |  |

Thus, we have a first order system:

**With initial-conditions:**

The right hand side thus would simply be:

How would the system be solved if we did not have an initial-condition for but only for ?

Well, in that case it is not a standard IVP. But we can find our unknown using the boundary-conditions:

With initialconditions:

Which thus would give:

and we can use the same dictionary as above.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Starter med at omskrive systemet med følgende dictionary:

|  |  |  |  |
| --- | --- | --- | --- |
| Old variable | New variable | Initial Value | Differential equation |
|  |  | 5 |  |
|  |  | -0.8 / -0.9 |  |

Implementeres I matlab:

y\_guess\_1 = [5;-0.8];

y\_guess\_2 = [5;-0.9];

tspan = [0,20];

[x\_shot\_1,y\_shot\_1] = ode45(@odefun,tspan,y\_guess\_1);

[x\_shot\_2,y\_shot\_2] = ode45(@odefun,tspan,y\_guess\_2);

endVals = [y\_shot\_1(end,:),y\_shot\_2(end,:)] %%first part of solution is gonna be end-point of y (trajectory), second part is gonna be end-point of y' (velocity)

endpoints = [endVals(:,1:2:3)] %%only take trajectory-solutions

plot(x\_shot\_1,y\_shot\_1(:,1))

hold on

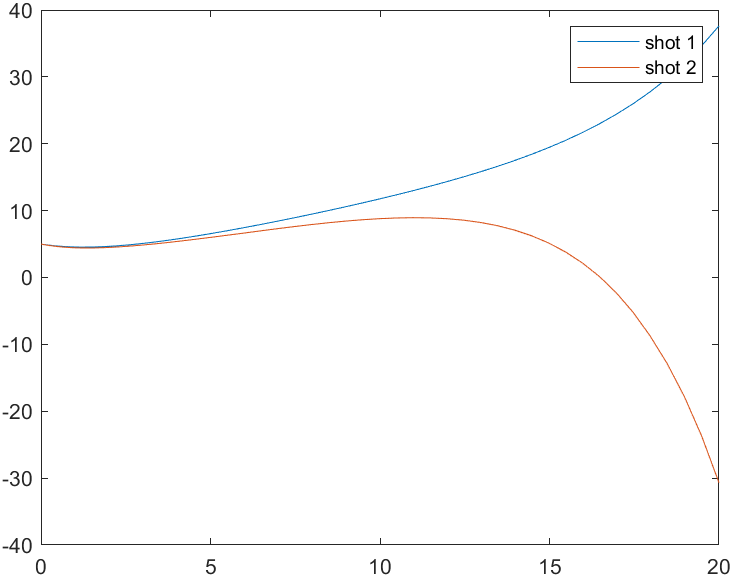
plot(x\_shot\_1,y\_shot\_2(:,1))

legend('shot 1','shot 2')

function [X] = odefun(x,y)

[X] = [y(2);(y(1)-x-y(2))\*1/7];

end



skal altså ligge et sted imellem og . Vi har endepunkts-værdispændet med vores to gæt.



guess\_interval = linspace(-0.9,-0.8,1000);

phi = zeros(numel(guess\_interval),1); %for efficiency

for i=1:numel(guess\_interval)

[x\_tmp,y\_tmp] = ode45(@odefun,tspan,[5;guess\_interval(i)]);

phi(i) = y\_tmp(end,1);

end

figure(2)

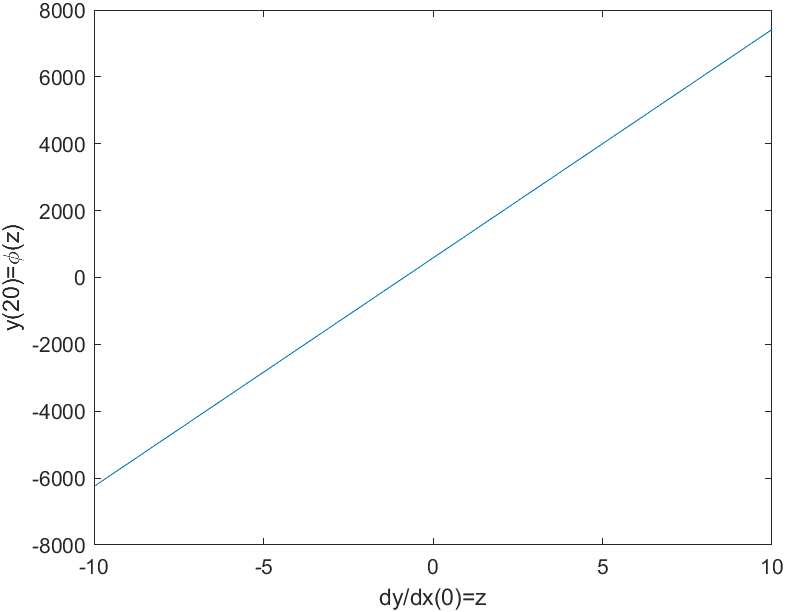
plot(guess\_interval,phi)

xlabel('dy/dx(0)=z')

ylabel('y(20)=\phi(z)')

Et billede, der indeholder tekst

Automatisk genereret beskrivelse



It is a linear function. Thus, we would use the secant method to find the root. We would only need one iteration.

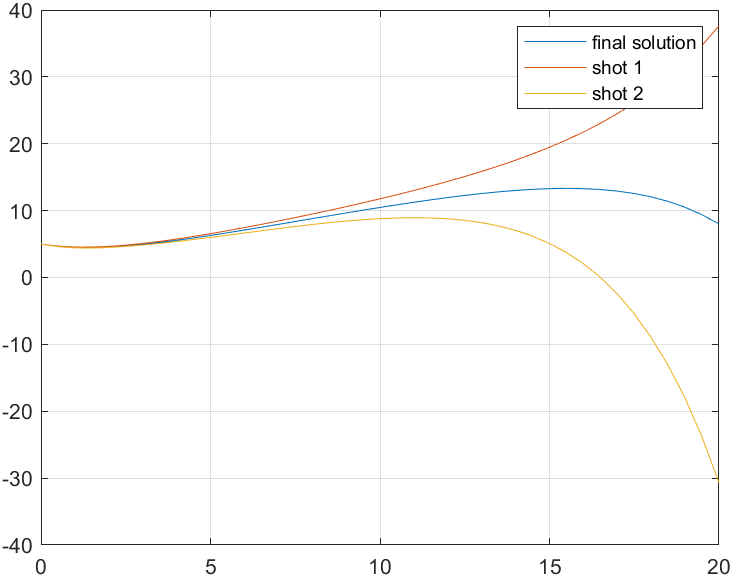
Et billede, der indeholder tekst

Automatisk genereret beskrivelse

We use the secant-method with initial-point:

Which gives us the root-value

5)



Tada. Vi opfylder boundary-value problemet.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

A1:

(as it is a second order runge kutta)

A2)   
Globally, we multiply by n:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Six dif.eqs.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse



RK4 has a local truncation error of . Globally, this results in a truncation error of .

Thus, we know:

And thus:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Den her laver du lige i morgen

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Made by hand, look at block.

Et billede, der indeholder tekst

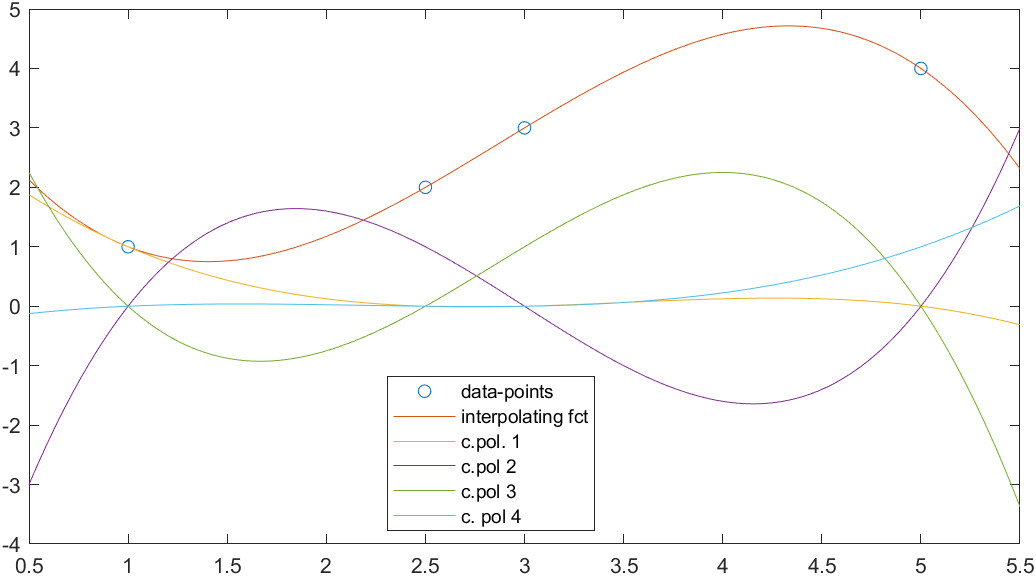
Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse



Seems very nice.

I did like this:



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

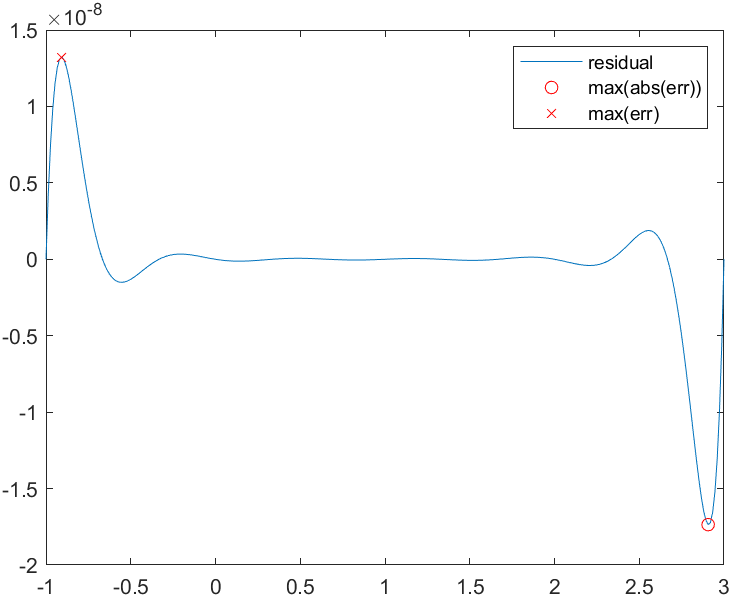
1)   
Using the second interpolation error theorem, the upper bound of the polynomial error is given by:

With , and

Thus, we derive the upper bound for the derivative (we have a very nice function):

And we have

2).



The maximal error is and occurs at

3)   
The maximal absolute error is at and has size . Thus, it is larger than the earlier found error. Both errors are smaller than the upper bound.



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

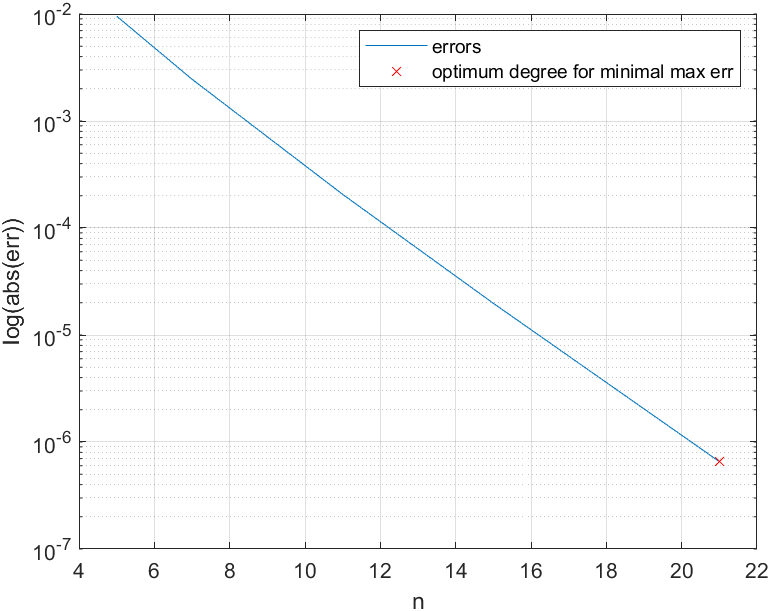






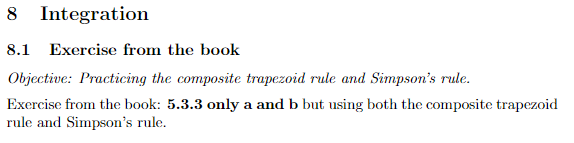
We see, that the approximation is best at appx n=7. Larger degree polynomials introduce error-oscillations.

When we make the interval way smaller:

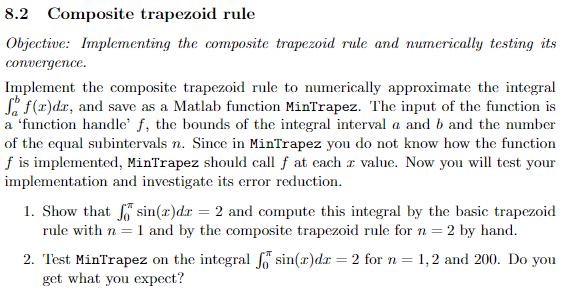


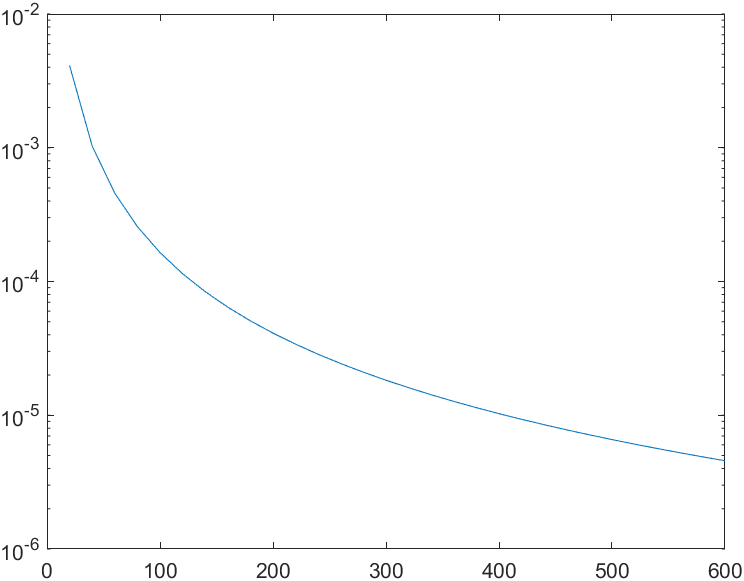
It seems the errors behaves nicely when increasing the order of the polynomial.   
But why? It has the x-axis as an asymptote. When only using -0.9..0.9 we do not need care about the asymptoticity. But when using -5..5 we do need to model an asymptote (linearity), and this linearity can not be modelled with high-order oscillating polynomials.

We see, that in general the error are biggest in the “tails” of the functions, close to the edges of the interval. This is called Runges phenomenon.



Made on paper





The error seems to be approximately linear after .

4)

%%Q4 data-fitting this boy

m = n(15:end);

Lerr = log(err(15:end));

A = [-log(m'),ones(numel(m),1)];

c = (A'\*A)\(A'\*Lerr)

Gives , .

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

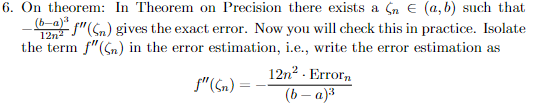
We know:

Which using our boundaries yields:

As requested.   
Solving the inequality we find that we need to bring at least terms.

Testing with the given equation (1):

Which is not the same, but kinda close to (remember, we found an UPPER bound ie worst-case).

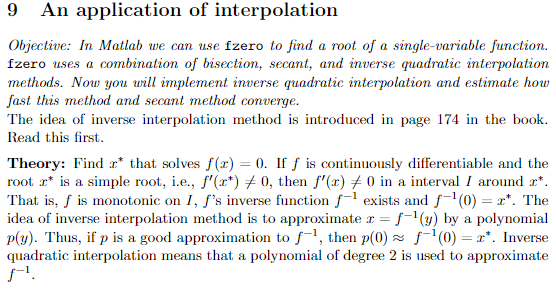




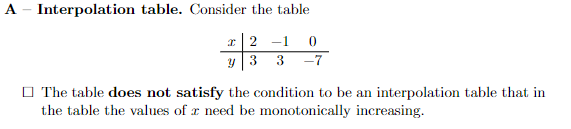
Den dobbelt-afledte er . Derfor skal udtrykket have værdier, som er imellem . Når man plotter det, er det heldigvis tilfældet. Endvidere kan vi skrive gennemsnittet:

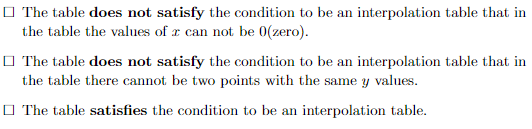
Vi kan også bestemme gennemsnitsværdien:

Som er, hvad funktionen konvergerer imod.



Du mangler invers interpolation





Last one.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

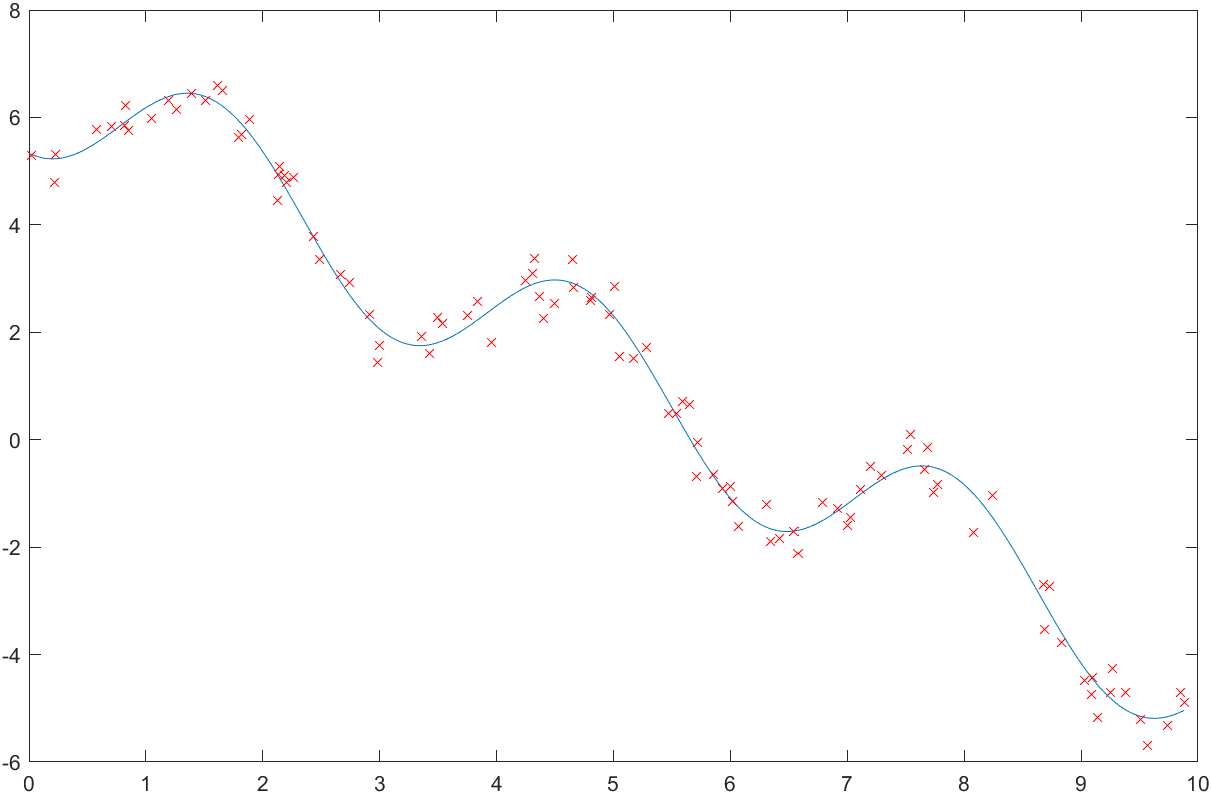
Automatisk genereret beskrivelse

Basis-functions are:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse





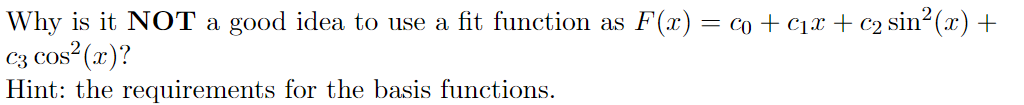
Three requirements for the basis-functions:

1. Linearly independent

2. Appropriate for the given problem

3. need to be well-conditioned for numerical work

All basis-functions are linearly independent.   
Are they appropriate for the given problem?

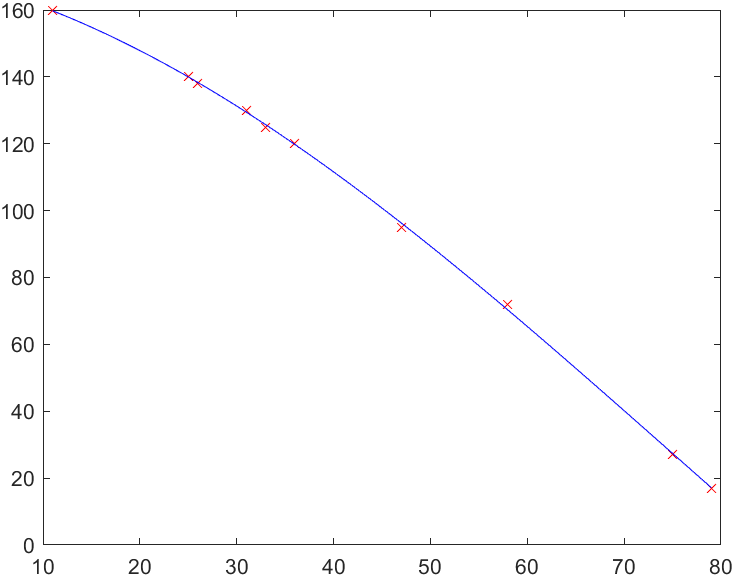


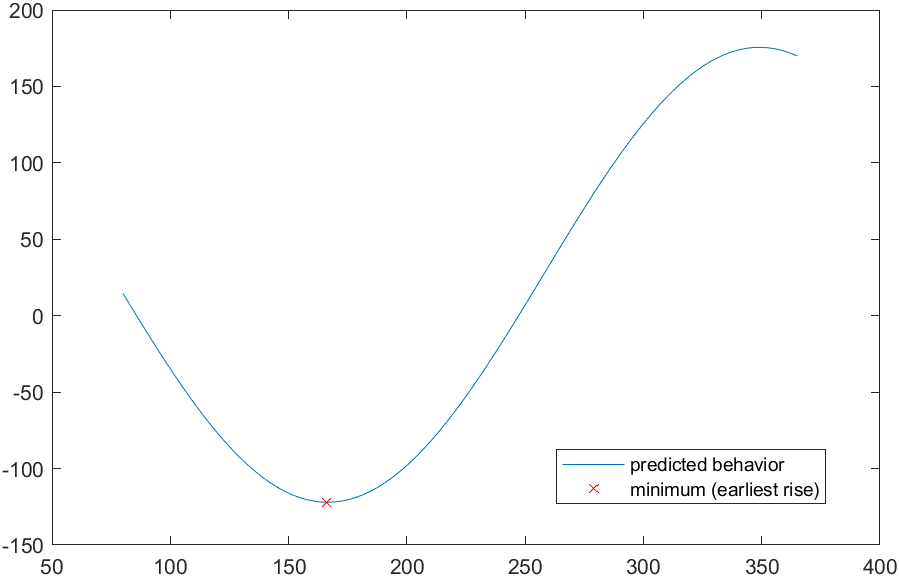


Obviously they do not describe the periodicity well. We know that , which we just scale a bit funnily. Thus we loose the oscillations.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse





The earliest rise will thus be in summer on the 166th day after jan 1st at around 6am-122min = 3.58am.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Normal equation is:

with

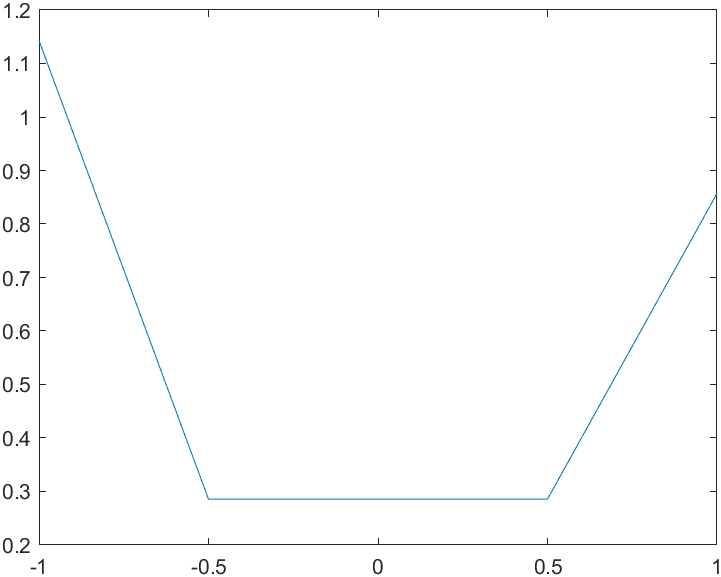
* which gives:

The normal equation for calculating the constants is:

The system matrix is and the right-hand side is

2)

3)   
Quite bad fit.



max abs Err:

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

function [x] = MyNewton(f,df,x0,nmax)

%f is function handle for function

%df is function handle for derivative

%x0 is start-guess

%nmax is number of iterations used

x = ones(nmax+1,1);

x(1) = x0;

for n=1:nmax

fx = f(x(n));

fp = df(x(n));

x(n+1) = x(n)-fx/fp;

end

end

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

We see that the converge towards root = 1.   
But how do we know if it is quadratic convergence?

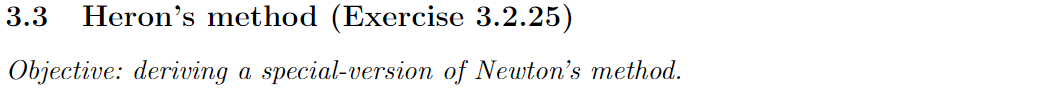
We can tell, that the error is falling, and we gain a doubling of significant digits in every iteration after iteration 3. The proper way to tell if we have quadratic convergence is to look at the ratio-table.  
When we have quadratic convergence. We see that this is the case after a couple of iterations. It converges towards 0.83.

Is this the same as the constant?

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

It absolutely seems so <3



Et billede, der indeholder tekst

Automatisk genereret beskrivelse

The newton iteration is gonna be:

Subtracting the half of a size is the same as half of the size:

Tada

f = @(x)x^2-2;

df = @(x)2\*x;

nmax = 2;

x0 = 1;

x = MyNewton(f,df,x0,nmax);

while(abs(x(end)-sqrt(2))>eps)

nmax = nmax + 1;

disp("incrementing");

x = MyNewton(f,df,x0,nmax);

end

disp("done :-) ")

sprintf("nmax becomes %d",nmax)

We need at least 5 iterations to get to machine-precision.

Using bisection method:

We have:

Multiplied they are <0. We find the midpoint:

As , we know that the root lies in , as .   
Iteration 2:

Etc.   
We would need

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

52 iterations.

->

function root = bisection\_method(f,a,b,nmax,epsilon)

fa = f(a);

fb = f(b);

if(sign(fa)==sign(fb))

disp("same sign");

return

end

error = b-a;

for n=0:nmax

error = error/2;

c = a+error;

fc = f(c);

if(abs(error)<epsilon)

disp("convergence achieved")

root = c;

return

else

if(sign(fa)~=sign(fc))

b = c;

fb = fc;

else

a = c;

fa = fc;

end

end

end

end

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

Newton-method in this case does not converge *at all.* This is because we have an unlucky start-guess. It gets stuck in a loop.   
Secant-method converges rather quickly.   
Bisection-method converges, but does so very slowly.

Newton with a good starting guess converges way quicker.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

First answer.

x = [-1,0,0.6,1]';

y = [-0.5,0.1,0.3,0.24]';

A = [ones(numel(x),1) x];

c = (A'\*A)\(A'\*y)

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

C. Look p. 131 – this is right – but that does not even matter if B is not satisfied. If we start in a flat-spot we never converge at any rate.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

To iterationer giver 0.5439.

Et billede, der indeholder tekst

Automatisk genereret beskrivelse

The two points are almost exactly equally big. Thus, we would have a horizontal secant, and divide by 0 (in practice, we would have underflow as it is not exactly 0).