Num. Ana. - Übung 2

Felix Dreßler (k12105003)
Elisabeth Köberle (k12110408)
Ricardo Holzapfel (k11942080)

8. Februar 2023

1 gedämpftes Newton-Verfahren

Im folgenden die Implementierung des gedämpften Newton-Verfahrens. Die Wahl von $maxIter$ basiert auf den unten angeführten Tests und gibt eine realistische obere Schranke für Konvergenz des Verfahrens. (genaueres zu F2 in den Resultaten) Der Wert mu wird im Abbruchskriterium verwendet. Es muss gelten $mu \in [0, 1]$. Die Wahl von $mu = 0.1$ basiert auf einer Empfehlung des Skripts.

```
function [x, exitflag, iter, f_eval] = solveEq(fun, x0)
    x = x0;
    maxIter = 30;
    [F,DF] = fun(x);
    f_eval = 1;
    mu = 0.1; % Wert aus Skript
    exitflag = 1;
    iter = 0;

    % falls bereits nahe genug an 0
    if(norm(F) <= 10e-16) % verfahren erfolgreich, weil nahe genug an 0
        exitflag = 0;
        return;
    end

    for iter = 1 : maxIter
       p = linsolve(DF,-F);

       a = 1;
       [F_x_k,DF_x_k] = fun(x + a  * p);
       f_eval = f_eval + 1;

       while( norm(F_x_k) > (1- mu  * a) * norm(F))

          if(a >= 0.25)
             a = a/ 2;
          elseif(a > 1/3)
              a = 1/3;
          elseif(a > 0.1)
              a = 0.1;
          else
              a = a / 10;
          end

          [F_x_k,DF_x_k] = fun(x + a  * p); % berechnet F(xk + ak*pk)
          f_eval = f_eval + 1;

       end

       F = F_x_k;
       DF = DF_x_k;
       x = x + a * p;

       if(norm(F) <= 10e-16) % verfahren erfolgreich, weil nahe genug an 0
           exitflag = 0;
           return;
       end

    end
 end
```

2 Testfunktion 1

Wir testen nun die Funktion F1 aus der Angabe mit Startvektor $x0 = \begin{pmatrix} 10 \\ 10 \end{pmatrix}$

```
>> [x, exitflag, iter, f_eval] = solveEq(@F1, x0)

x =

-2.772012301967051
2.039147038513721


exitflag =

0


iter =

16


f_eval =

18
```

Im Vergleich mit fsolve ergibt sich:

```
>> fsolve(@F1, [10,10]')
ans =

-2.7720
2.0391
```

Die beiden Lösungen liegen also sehr Nahe beieinander.

3 Testfunktion2

Wir testen nun die Funktion F2 aus der Angabe mit Startvektor $x0 = \begin{pmatrix} 10 \\ 10 \\ 10 \\ 10 \end{pmatrix}$

Im Vergleich mit fsolve ergibt sich:

```
>> fsolve(@F2, [10, 10, 10, 10]')
ans =

4.8482
5.2704
23.5149
3.4195
```

4 Testfunktion3

Nun testen wir die Funktion $f(\begin{pmatrix} x \\ y \end{pmatrix}) = \begin{pmatrix} sin(x) \\ cos(y) \end{pmatrix}$ Dazu wurde folgende MatLab-Funktion verwendet:

```
function [F, DF] = testFun(x)
% n = 2
F= zeros(2,1);
F(1) = sin(x(1));
F(2) = cos(x(2));
DF = zeros(2,2);
DF(1,1)=cos(x(1));
DF(1,2)= 0;
DF(2,2)=-sin(x(2));
DF(2,1)=0;

end
```

Im folgenden noch die Tests:

```
>> [x, exitflag, iter, f_eval] = solveEq(@testFun, x0)

x =

9.424777960769379
10.995574287564276


exitflag =

0


iter =

5


f_eval =

6
```

Im Vergleich mit fsolve ergibt sich:

```
>> fsolve(@testFun, [10, 10]')
ans =

9.4248
10.9956
```

Wir erhalten also einen sehr ähnlichen Wert.

5 Testfunktion4

Nun testen wir die eindimensionale Funktion $f(x) = e^{-x}$ Dazu wurde folgende MatLab-Funktion verwendet:

```
function [F, DF] = slowFun(x)
% n = 1
F = exp(-x);
DF = -exp(-x);
end
```

Im folgenden noch die Tests:

```
>> [x, exitflag, iter, f_eval] = solveEq(@slowFun, 10)

x =

35


exitflag =

0


iter =

25


f_eval =

26
```

Im Vergleich mit fsolve ergibt sich:

```
>> fsolve(@slowFun, 10)
ans = 12.000
```

hier erhalten wir unterschiedliche Nullstellen (obwohl die Funktion natürlich keine Nullstelle hat). Das liegt vermutlich an einer anderen Genauigkeit der fsolve Funktion im Vergleich zu unserer Implementierung

6 Testfunktion5

Nun testen wir die eindimensionale Funktion $f(x) = 0$ Dazu wurde folgende MatLab-Funktion verwendet:

```
function [ F, DF ] = zero( x )
% n = 1
F = 0;
DF = 0;
end
```

Diese Funktion ist interessant weil wir natürlich keine Iteration brauchen, wenn unser Ausgangswert bereits eine Nullstelle ist.

Im folgenden noch die Tests:

```
>> [x, exitflag, iter, f_eval] = solveEq(@zero, 10)

```

```
x =

10


exitflag =

0


iter =

0


f_eval =

1
```

Im Vergleich mit fsolve ergibt sich:

```
>> fsolve(@zero, 10)
ans = 10
```

Klarerweise geben beide Funktionen wieder x0 zurück.

7 Testfunktion6

Nun testen wir die eindimensionale Funktion $f(x) = x^2 + 1$ Dazu wurde folgende MatLab-Funktion verwendet:

```
function [ F, DF ] = noRoot( x )
% n = 1
F = x.*x + 1;
DF = 2.*x;
end
```

Diese Funktion besitzt keine Nullstelle, deshalb sollten wir hier nach der maximalen Iterationszahl auch abbrechen.

Im folgenden noch die Tests:

```
>> [x, exitflag, iter, f_eval] = solveEq(@noRoot, 10)

x =

6.101667877118361e-09


exitflag =

1


iter =

30
```

```


f_eval =

425
```

Im Vergleich mit fsolve ergibt sich:

```
>> fsolve(@noRoot, 10)
ans = -2.0678e-05
```

Obwohl die Funktion keine Nullstelle besitzt erhalten wir bereits gut Approximationen für das Minimum ($x = 0$) der Funktion.

8 Resultate

Testfunktion 6 gibt Grund zur Vermutung, dass das Verfahren in manchen Fällen gegen ein Minnimum der Funktion konvergiert, falls keine Nullstelle vorhanden ist.

Versucht man verschiedene Werte für mu wird die Eingabefunktion fun weniger oft aufgerufen. Das führt bei nicht-Konvergenz dann auch zu einem schnelleren Abbruch. Im folgenden zwei Versuche mit unterschiedlichen mu:

$mu = 0.1$

```
[x, exitflag, iter, f_eval] = solveEq(@F1, transpose(x0))

x =

-2.7720
2.0391


exitflag =

0


iter =

16


f_eval =

18



>> [x, exitflag, iter, f_eval] = solveEq(@noRoot, 10)

x =

6.1017e-09


exitflag =
```

```
1


iter =

30


f_eval =

425
```

$mu = 0.5$

```
>> [x, exitflag, iter, f_eval] = solveEq(@F1, transpose(x0))

x =

-2.7720
2.0391


exitflag =

0


iter =

16


f_eval =

18

>> [x, exitflag, iter, f_eval] = solveEq(@noRoot, 10)

x =

1.1243e-07


exitflag =

1


iter =

30


f_eval =

287
```