Ü 2 Zweite Übungseinheit

Inhalt der zweiten Übungseinheit:

- Zeilen- und Spaltenvektoren
- Matrizen, Abbildungen
- Funktionen
  - Funktionen als Funktions-M-Datei programmieren
  - Fixpunkte und Nullstellen von Funktionen
  - Befehle `fzero, roots`
  - Funktions-Einzeiler (*anonymous functions*)
- Kontrollstrukturen
  - Schleifen
  - Verzweigungen
- Fixpunkt-Iteration, ein- und mehrdimensional

Ü 2.1 Zeilen- und Spaltenvektoren

Sie haben in der vorigen Einheit bereits mit Vektoren gearbeitet: Beim Zeichnen von Funktionsgraphen haben Sie Bereiche für x- und y-Werte als Zeilenvektoren erzeugt. MATLAB unterscheidet zwischen Zeilen- und Spaltenvektoren.

Hier folgt eine kurze Zusammenfassung zum Erzeugen von und Rechnen mit Vektoren. Klicken Sie sich durch diese Anleitung, dann lernen Sie die wichtigsten Befehle dazu kennen. Überlegen Sie sich bei jedem Ergebnis, welche Rechenregel MATLAB dabei verwendet hat!

- Standard-Operationen der Vektorrechnung (Addition `+` , Subtraktion `-` , Multiplikation `*` mit Skalar, inneres Produkt `*` von Zeilen- mit Spaltenvektor, Vektor transponieren `'`)
- elementweise Multiplikation `.*` und Division `./` bei Vektoren gleicher Form. Ergebnis ist ein Vektor gleicher Form; die entsprechenden Elemente werden multipliziert bzw. dividiert.
- Wenn ein Operand Zeilen- und der andere Spaltenvektor ist, erzeugt Addition oder Subtraktion eine Matrix nach dem Muster „jedes Element mit jedem“.
- Bei der Multiplikation kommt es drauf an: linker Partner Zeile, rechter Partner Spalte berechnet inneres Produkt; linker Partner Spalte, rechter Partner Zeile berechnet eine Matrix nach dem Muster „jedes Element mit jedem“ (heißt auch äußeres, Kronecker- oder Tensor-Produkt - da ist sich die Fachwelt nicht einig).
- Bei der Division `/` zweier Zeilen- bzw. zweier Spaltenvektoren passieren völlig bizarre Dinge – was MATLAB da berechnet, versuchen wir hier erst gar nicht zu erklären.

Vektoren erstellen

```
>> x=[1 2 3]
x =
     1     2     3
```

x ist ein Zeilenvektor mit drei Elementen.
Vektoren sind von eckigen Klammern `[ ]` begrenzt.

```
>> x=[1, 2, 3]
x =
     1     2     3
```

So geht 's auch: Die einzelnen Komponenten in einer Zeile können Sie durch Leerzeichen (so wie gerade vorher) oder durch Beistriche (so wie hier) trennen.

```
>> y=[2;1;5]
y =
     2
     1
     5
```

y ist ein Spaltenvektor mit drei Elementen. Die Strichpunkte ; trennen die Zeilen im Vektor.

```
>> y = [2
1
5]
y =
     2
     1
     5
```

Statt des Strichpunkten können Sie im *command window* oder in Dateien auch eine neue Zeile beginnen.

Rechenoperationen

```
>> z=[2 1 0];
>> a=x+z

a =

     3     3     3
```

Sie können zwei Zeilen- oder zwei Spalten-Vektoren addieren oder subtrahieren – das entspricht den Standardregeln der Vektorrechnung

```
>> b=2*a

b =

     4     4     0
```

Skalar mal Vektor, das ist eine Standard-Regel

```
>> x/2
ans =
    0.5000    1.0000    1.5000
```

Division Vektor durch Skalar, ebenfalls Standard

```
>> 2/x
Error using  /
Matrix dimensions must agree.
```

in dieser Reihenfolge nicht möglich

```
>> 2./x
ans =
    2.0000    1.0000    0.6667
```

Das geht: Elementweise Division

```
>> b=x+y
b =
     3     4     5
     2     3     4
     6     7     8
```

Addition von Zeilen- plus Spaltenvektor ergibt eine Matrix – Das ist nicht gerade Standard, sondern MATLABs kreative Erweiterung des Plus-Operators.

Achtung bei älteren MATLAB-Versionen! Versionen vor R2016b erlauben nicht, Zeilen- zu Spaltenvektoren zu addieren. Obiger Befehl ergibt eine Fehlermeldung

```
» b=x+y
Error using +
Matrix dimensions must agree.
```

Die kreative Interpretation der Standard-Regeln in den neueren Versionen gilt für viele arithmetische und logische Operatoren. Regel: Wenn es irgendwie sinnvoll erscheint, erweitert MAT-

LAB die Operanden so, dass eine elementweise Operation möglich wird[14].

MATLAB führte dieses automatische Erweitern 2016 mit der Begründung: [15] ein: *"MATLAB has a long history of inventing notation that became widely accepted"* – die Kommentare im Blog sind eher kontroversiell. Nicht alle Anwender freut so einen lockerer und origineller Umgang mit den Rechenregeln.

> Vorsicht bei Rechenoperationen mit Vektoren! Wenn die Formate von Zeilen- und Spaltenvektoren nicht den Regeln der Standard-Vektor- und Matrizenrechnung entsprechen, findet MATLAB oft eine kreative, aber meist ungewollte Interpretation der Anweisungen.

```
>> a=x.*z
a =
     2     2     0
```

Sie können zwei Vektoren derselben Größe elementweise multiplizieren (oder dividieren); array operator `.*` (oder `./`)

```
>> x*y
ans =
    19
```

Skalares (oder inneres) Produkt. Standardregel für Zeilen- mal Spaltenvektor

```
>> y*x
ans =
     2     4     6
     1     2     3
     5    10    15
```

Spalten- mal Zeilenvektor ergibt Matrix. Heißt äusseres, Tensor- oder Kronecker-Produkt; auch das ist eine Standard-Rechenoperation.

```
>> x.*y
ans =
     2     4     6
     1     2     3
     5    10    15
```

Elementweise Multiplikation von Zeilen- mit Spaltenvektor ergibt dieselbe Matrix wie oben – MATLABs kreative Interpretation.

Vektoren transponieren

```
>> x
x =
     1     2     3
>> x'
ans =
     1
     2
     3
>> y
y =
     2
     1
     5
>> y'
ans =
     2     1     5
```

Höchste Zeit, dass Sie den ' Operator kennenlernen. Damit transponieren Sie Vektoren: Zeilen- wird Spaltenvektor, und umgekehrt.

[14] Für eine genauere Erklärung suchen Sie in der MATLAB Dokumentation nach *Compatible Array Sizes for Basic Operations*

[15] `https://blogs.mathworks.com/loren/2016/10/24/matlab-arithmetic-expands-in-r2016b/`

Im Matlab-Desktop sehen Sie übrigens die Registerkarte „Workspace“ links in der Mitte, und wenn Sie draufklicken, darüber ein Fenster. (Möglicherweise ist das Fenster schon offen, ohne dass Sie extra angeklickt haben.) Alle bisher verwendeten Variablen sind dort aufgelistet. Das Matrix-Symbol neben dem Variablennamen erinnert daran, dass MATLAB alle Variablen als Matrizen interpretiert - Skalare als 1×1-Matrizen, Zeilenvektoren mit n Elementen als $1 \times n$- und Spaltenvektoren mit m Einträgen als $m \times 1$-Matrizen. Doppelklick auf eine Zeile dieser Liste öffnet ein Fenster, den „Array Editor“, das die Werte der Variablen in Tabellenform anzeigt. Die Werte lassen sich (Doppelklick auf die entsprechende Zelle im Tabellenblatt) auch ändern.

Zeilenvektoren mit regelmäßigen Einträgen

Zum Erzeugen von Zeilenvektoren noch einige Beispiele:

```
>> x=linspace(0,10,5)

x =

         0    2.5000
5.0000    7.5000   10.0000
```

Dieser Befehl erzeugt einen Zeilenvektor der Länge 5, dessen Elemte äquidistant im Intervall [0,10] liegen. Ohne drittes Argument entsteht immer ein Vektor der Länge 100.

```
>> x=0:2.5:10
x =
         0    2.5000
5.0000    7.5000   10.0000
```

Auch mit dem Doppelpunkt-Operator lässt sich derselbe Zeilenvektor erzeugen nach dem Muster `Anfangswert:Schrittweite:Endwert`
Bei gewünschter Vektor-Länge ist `linspace` einfacher; wenn die Schrittweite vorgegeben ist, bietet sich die Doppelpunkt-Variante an.

```
>> x=1:6
x =
     1     2     3     4
5     6
```

Bei Schrittweite 1 ist die Doppelpunkt-Anweisung besonders einfach

Ü 2.2 Matrizen

Der Name MATLAB steht für *matrix laboratory*. Das signalisiert Ihnen: Das Arbeiten mit Matrizen ist in dieser Rechenumgebung ein ganz zentrales Thema. Hier eine kurze Einführung ins Erstellen von und in die Grundrechnungsarten für Matrizen.

Hoffentlich sind Ihnen die Rechenregeln der Matrizenrechnung noch in Erinnerung. Bevor Sie weiterarbeiten – ist Ihnen klar, wir die Multiplikation von Matrizen abläuft? Sie sollten mit Stift auf Papier nachrechnen können:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 0 \end{bmatrix} = \begin{bmatrix} 21 & 24 & \dots \\ 47 & \dots & ? \end{bmatrix}$$

Matrizen erstellen

```
>> A=[1 2; 3 4]
A =
     1     2
     3     4
>> B = [ 5, 6, 7
8, 9, 0]
B =
     5     6     7
     8     9     0
```

Eingabe: Leerzeichen oder Komma trennt Komponenten in einer Zeile; Strichpunkt oder neue Zeile trennt Zeilen.

Rechenoperationen, Zugriff auf Elemente

MATLAB bietet ähnlich wie für Vektoren folgende Operationen an:

`+ , -`	elementweise Addition, Subtraktion
`*`	multipliziert nach den Regeln der linearen Algebra
`.* , ./`	elementweise Multiplikation, Division
`A'`	Matrix `A` transponieren

Bei Rechenoperationen zwischen Matrizen ist wichtig, dass die Matrix-Dimensionen zueinander passen. Versuchen Sie für die gerade erstellten Matrizen `A` und `B`

```
>> C=A*B
C =
    21    24     7
    47    54    21
```

Matrix-Multiplikation nach den Regeln der linearen Algebra.

```
>> C=B*A
Error using  *
Incorrect dimensions ...
```

Innere Dimensionen müssen übereinstimmen.

Es is A eine 2×2-, B eine 2×3-Matrix. Bei $A \cdot B$ ist in $(2 \times 2) \cdot (2 \times 3)$ die innere Dimension 2. Bei $B \cdot A$ mit $(2 \times 3) \cdot (2 \times 2)$ passen die inneren Dimensionen nicht: $2 \neq 3$.

Elementweise Operationen lassen sich nur auf Matrizen gleicher Größe anwenden.

```
>> C=A+B
Arrays have incompatible sizes for this operation.
```

```
>> A(1,2)
ans =
     2
```

Zugriff auf Element a_{12}

```
>> B(1:2, 2:3)
ans =
     6     7
     9     0
```

Zugriff auf Elemente in Zeilen 1 bis 2, Spalten 2 bis 3

Ü 2.3 Der Doppelpunkt-Operator

Der Doppelpunkt (englisch: *colon*) ist einer der nützlichsten Operatoren im Umgang mit Matrizen und Vektoren. Er erzeugt Vektoren, Index-Bereiche und Iterationsindizes. Als Index-Ausdruck kann er aus Matrizen einzelne Zeilen oder Spalten herausgreifen.

In der MATLAB-Hilfe finden Sie Informationen dazu unter dem Stichwort *colon*.

Syntax-Beispiele

Erzeugen einfacher Zahlenfolgen

`x=3:8` erzeugt den Zeilenvektor `[3, 4, 5, 6, 7, 8]`

`x=0:2:6` erzeugt den Zeilenvektor `[0, 2, 4, 6]`

`x=10:-1:0` erzeugt den Countdown-Vektor `[10, 9, ..., 3, 2, 1, 0]`

Diese Syntax werden wir für Zählschleifen im Kapitel Ü 2.6 verwenden.

Index-Ausdrücke, Zugriff auf Vektor- und Matrixelemente

Als Matrix- oder Vektorindex bedeutet der Doppelpunkt soviel wie „alle Zeilen“ oder „alle Spalten“.

`A(:,4)` alle Zeilen, Spalte 4; die gesamte vierte Spalte von A

`A(3,:)` Zeile 3, alle Spalten; die gesamte dritte Zeile von A

`x(3:7)` verwendet den Vektor `3:7 = [3, 4, 5, 6, 7]` als Zugriffs-Index; greift den Teil-Vektor bestehend aus den x-Komponenten 3 bis 7 heraus

`A(2:4,3:5)` die Untermatrix bestehend aus den Zeilen 2 bis 4 und Spalten 3 bis 5 von A

Achtung, verwechseln Sie nicht `x(3:7)`, `x(3,7)` und `x([3,7])` ! Der erster Ausdruck ist ein Teilvektor von `x` (siehe oben); der zweite Ausdruck bezeichnet das Matrixelement x_{37}; der dritte verwendet den Vektor `[3, 7]` als Zugriffs-Index, greift also die 3- und 7- Komponente des Vektors x heraus. Das ergibt einen Vektor der Länge 2 mit den Komponenten `[x(3), x(7)]`.

Ü 2.4 Matrizen und Abbildungen

Die Matrix ist dazu gedacht,
dass sie aus einem Vektor einen and'ren macht.

Die linearen Abbildungen zwischen Vektorräumen $\mathbb{R}^m \to \mathbb{R}^n$ entsprechen genau den Matrix-Vektor-Multiplikationen $\mathbf{y} = A \cdot \mathbf{x}$ mit $n \times m$-Matrizen A.

Lineare Beziehungen zwischen Ein- und Ausgangsgrößen sind Grundbausteine für jede Art von Datenanalyse und -modellierung. Spannend und herausfordernd kann das für hochdimensionale Vektorräume werden. Im 2- und 3-dimensionalen Raum hingegen lassen sich Abbildungen noch gut visualisieren. Wenn Sie die Beispiele hier durchgearbeitet haben, kann die Verallgemeinerung auf mehrdimensionale Räume auch kein großer Schritt mehr sein. Laden Sie dazu die Datei `cat.dat` von der Übungs-Homepage in Ihr Arbeitsverzeichnis.

```
>> X=readmatrix('cat.dat');
>> size(X)
ans =
   119     2
>> X=X';
>> size(X)
ans =
     2   119
```

Sie lesen aus einer Textdatei einen Datensatz als Matrix ein. X hat 119 Zeilen und 2 Spalten. Wir vertauschen fürs weitere Arbeiten Zeilen und Spalten. X hat also nun 2 Zeilen und 119 Spalten.

> Für MATLAB-Versionen vor R2019a: Den Befehl `X=readmatrix('cat.dat')` gibt es noch nicht. Laden Sie die Skript-Datei datenX.m herunter und führen Sie diese aus. Danach ist die Matrix `X` im workspace geladen und Sie können mit den Befehlen `size(X)` und `X=X';` weiterarbeiten.

Die Spaltenvektoren in `X` entsprechen 119 Punkten im 2-dimensionalen Raum. Zeichnen Sie diese Punkte, dann können Sie sich die Daten besser vorstellen.

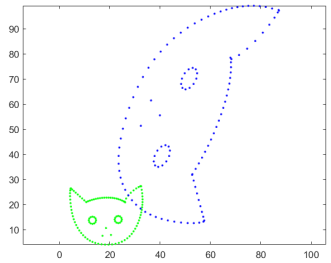
```
>> plot(X(1,:),X(2,:),'b.')
>> axis equal
```

Arnolds Katze[16] blickt Sie an. Gegeben sei nun die Matrix

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix} .$$

Wie wirkt diese Matrix auf Punkte des $\mathbb{R}^2$? Sie können die 2×2-Matrix A auf die 119 Spaltenvektoren, die in der Matrix X stehen, einfach durch Multiplikation anwenden: $Y = A \cdot X$ ist eine 2×119-Matrix, deren Spaltenvektoren jeweils die mit A multiplizierten Spaltenvektoren von X sind. Lassen Sie sich Ausgangs- und Bildpunkte zeichnen:

```
A=[ 1 2; 3 0];
Y=A*X;
plot(X(1,:),X(2,:),'g.',Y(1,:),Y(2,:),'b.')
axis equal
```



Sie sehen: A verdreht, vergrößert und verzerrt die Katze. Sie transformiert Ausgangsvektoren in Bildvektoren, die (außer in Sonderfällen) andere Länge und Richtung haben. (Spätestens jetzt sollten Sie das Merksprüchlein vom Anfang des Abschnittes verstehen!)

In diesem Beispiel werden alle Vektoren mehr oder weniger verlängert, aber nicht beliebig groß. Der linke Ohrzipfel, Punkt $\approx$ [32; 27], wird auf [86; 96] abgebildet; das entspricht einer Verlängerung um den Faktor $\approx 3{,}1$.

[16] Der Mathematiker Wladimir Igorewitsch Arnold (1937–2010) illustrierte damit in Lehrbüchern der klassischen Mechanik die Eigenschaften von Transformationen.

```
>> x=[32; 27];
>> y=A*x
y =
    86
    96
>> norm(y)/norm(x)
ans =
    3.0784
```

So können Sie Verlängerungs-Faktoren berechnen (in der 2-Norm).

Versuchen Sie andere Werte für x. Für welchen Vektor finden Sie den größten Verlängerungsfaktor? Umgekehrt: finden Sie auch einen Vektor mit besonders kleinem Verlängerungsfaktor?

Die Norm einer Matrix liefert den maximalen Verlängerungsfaktor

(Das gilt für 1-, 2- oder ∞-Norm; je nachdem, in welcher Norm gemessen wird, können die Werte leicht unterschiedlich sein. Für die Matrix A aus diesem Beispiel ist $\|A\|_2 = 3{,}5266$. Weil sich dieser nicht so einfach aus den Matrixelementen berechnen lässt, arbeitet man auch mit der 1-Norm (maximale Spaltenbetragssumme: 4, der Vektor [1; 0] verlängert sich um diesen Wert in der 1-Norm) oder der ∞-Norm (maximale Zeilenbetragssumme: 3, der Vektor [1;1] verlängert sich um diesen Wert in der ∞-Norm)).

```
>> [norm(A) norm(A,1) norm(A,'inf')]
ans =
    3.2566    4.0000    3.0000
```

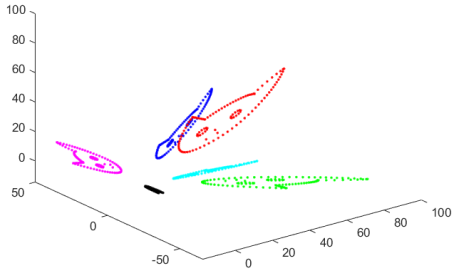
Das sind drei Matrix-Normen von A

Matrizen können aber auch lineare Abbildungen zwischen Vektorräumen unterschiedlicher Dimension vermitteln. Gegeben seien nun die Matrizen

$$B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \quad , \quad C = \frac{1}{15} \begin{bmatrix} -2 & 2 & 14 \\ -2 & -13 & -16 \\ 14 & 16 & 7 \end{bmatrix} .$$

Wenden Sie auf die Daten-Matrix X zuerst B und dann einige Male C an und sehen Sie Arnolds Katze durch den Raum wandern...

```
Z=B*X;
plot3(Z(1,:),Z(2,:),Z(3,:),'b.')
hold on
Z=C*Z;
plot3(Z(1,:),Z(2,:),Z(3,:),'r.')
Z=C*Z;
...
plot3(Z(1,:),Z(2,:),Z(3,:),'k.')
hold off
```



Aufgabe 11:

Gegeben sind die 2×119 Datenmatrix X wie oben und Matrizen

$$D = \begin{bmatrix} 1 & 1 \\ -\frac{1}{4} & \frac{3}{4} \end{bmatrix} \quad , \quad E = \begin{bmatrix} 1 & 2 \\ -\frac{1}{4} & \frac{3}{4} \end{bmatrix} \quad , \quad F = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ -1 & \frac{1}{2} \end{bmatrix} \quad , \quad G = \begin{bmatrix} \frac{1}{2} & 0 \\ -\frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

Wenden Sie die entsprechenden Abbildungen jeweils mehrere Male auf die Datenmatrix an und zeichnen Sie die Bildpunkte. Für welche Matrizen

- konvergiert die Fixpunkt-Iteration?

- liegt eine kontrahierende Abbildung vor?
  Eine kontrahierende Abbildung garantiert automatisch Konvergenz der Fixpunkt-Iteration. Aber auch für manche nicht-kontrahierende Abbildungen kann Fixpunktiteration konvergieren.
- liegt eine symplektische Abbildung vor?
  Das sind Abbildungen, die zwar Flächen verzerren, aber den Flächeninhalt unverändert lassen. Das können Sie aus den graphischen Darstellungen natürlich nur annähernd schätzen. Vielleicht finden Sie aber auch heraus: neben der Norm, die den Verlängerungsfaktor misst, gibt es eine weitere wichtige Matrix-Größe, die den Flächen- (oder allgemeiner: Volums-) Vergrößerungs-Faktor determiniert.

  Symplektische Abbildungen sind ein wichtiges Thema in der theoretischen Mechanik und für österreichische Radrennfahrerinnen[17].

Ü 2.5 Funktionen

Sie können in MATLAB eigene Funktionen definieren und in sogenannten Funktions-M-Dateien (*function M-files*) speichern. Bei einfache Funktionstermen ist es auch möglich, Funktions-Einzeiler, sogenannte *anonymous functions* zu verwenden.

Dieser Abschnitt erklärt folgende Punkte:

- Eine einfache Funktionsdatei
- Nullstellen mit `fzero` und `roots`
- Funktions-Einzeiler (*anonymous functions*)

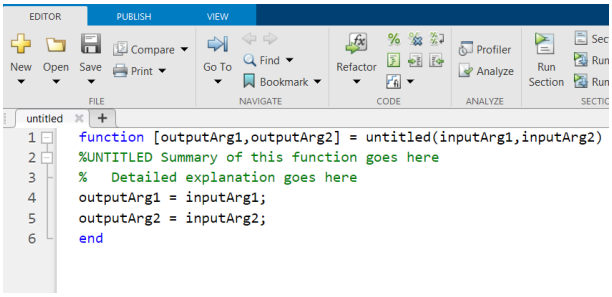
Ü 2.5.1 Eine einfache Funktionsdatei

Gegeben sei die Funktion

$$f : x \mapsto y = 3\cos x - \log x \ .$$

Sie ist ihnen in der Vorlesung und im Skriptum bereits begegnet.

Wählen Sie im MATLAB-Fenster links oben „New–Function“. Sie öffnen damit ein Fenster des MATLAB-Editors, in dem auch schon eine Muster-Funktion drinsteht.



Ergänzen Sie und erstellen Sie einen Quelltext. Orientieren Sie sich am folgenden Muster:

```
function [ y ] = meinf(x)
%MEINF Mein erster Versuch, eine Funktion zu programmieren
%   Funktion aus dem Skript, Abbildung 2
%
% Uebungen NM1, C.B. Feb.2014
y=3*cos(x)-log(x);
end
```

Die erste Zeile einer Funktionsdatei legt die Ein- und Ausgabeparameter sowie den Namen

[17] Anna Kiesenhofer, Integrable systems on b-symplectic manifolds. (Thesis, 2016)

einer Funktion fest. In diesem Fall heißt die Funktion `meinf`, hat eine Variable x als Argument (Input) und liefert einen Funktionswert y (Output). Die Argumente x und y können Skalare, aber auch Felder (Vektoren, Matrizen) sein. Sie können eigene Funktionsnamen verwenden, aber bitte nur zulässige Namen (keine Umlaute und Sonderzeichen, keine Ziffern zu Beginn)

Speichern Sie Ihre Funktion ab. Achten Sie darauf, dass die Datei unter dem Namen `meinf.m` gespeichert wird. Am besten legen Sie sich für die Matlab-Dateien, die Sie in diesen Übungen erstellen, ein eigenes Unterverzeichnis an.

Je nachdem, in welchem Verzeichnis sie gespeichert haben, kann für Sie eine kleine Zusatzarbeit anfallen:

> Skript- und Funktionsdateien lassen sich im *command window* nur aufrufen, wenn sie sich im aktuellen Ordner befinden.
> Wechseln Sie in der MATLAB-Oberfläche, linkes Fenster „Current Folder“ in das Verzeichnis, in dem Sie Ihre Funktion abgespeichert haben

Tipp: Schneller geht es, wenn Sie im Editor-Fenster, nachdem Sie die Funktion gespeichert haben, auf das grüne „Run“-Pfeil-Symbol klicken. Wenn eine Dialog-Box erscheint mit einem Text der Art *File D:/work/meinf.m is not found in the current folder blabla blabla*, dann klicken Sie einfach auf „Change Folder“ und ignorieren Sie weitere Meldungen (*mein.f requires more input...*) wurscht–das Verzeichnis ist gewechselt, mehr wollten wir nicht.

> Die Option *add to path* ist nicht zu empfehlen. Bei Namensgleichheit von Dateien verlieren Sie sonst die Übersicht, welche Datei nun tatsächlich ausgeführt wird.

Wechseln Sie wieder in die MATLAB *Command Window*. Stellen Sie zuerst gleich einmal die Anzeige von mehr Dezimalstellen ein und zusätzliche Leerzeilen ab:

```
>> format long g
>> format compact
>>
```

Andere Möglichkeiten:
short g gibt weniger Stellen.
short oder long e gibt Exponentialschreibweise.
format loose schiebt Leerzeilen ein.

Testen Sie nun, ob sich die Funktion aufrufen lässt:

```
>> y=meinf(1)
y =
   1.62090691760442
```

Sie haben die Funktion an der Stelle 1 ausgewertet.

Wenn Sie brav Kommentare eingegeben haben, gibt es nun eine Belohnung: Geben Sie ein:

```
>> help meinf
 meinf Mein erster Versuch, eine Funktion zu programmieren
    Funktion aus dem Skript, Abbildung 2

  Uebungen NM1, C.B Feb.2014
```

Und noch etwas ist toll an dieser Funktion: sie liefert nicht nur für einen skalaren Wert ein Ergebnis, sondern automatisch auch für eine ganze Werteliste. Erinnern Sie sich: Die Anweisung *Startwert*:*Schrittweite*:*Endwert* ist (alternativ zu `linspace`) eine zweite Möglichkeit, Zeilenvektoren zu erzeugen.

```
>> x=0:1:10

x =

     0     1     2     3
4     5     6     7     8
9    10
```

Werte von 0 bis zehn. Auch x=0:10 hätte funktioniert (Wenn Schrittweite fehlt, nimmt MATLAB automatisch 1)

Die Funktion `meinf` lässt sich elementweise auf den Zeilenvektor x anwenden.

```
>> meinf(x)
ans =   Inf    1.62090691760442
-1.94158769020137 ...
```

Kleiner Schönheitsfehler: $f(x) = 3\cos x - \log x$ ist für $x = 0$ nicht definiert. Der Funktionswert $f(0)$ wird als `Inf` gespeichert.

Ü 2.5.2 Nullstellen mit dem Newton-Verfahren

Dazu müssen Sie auch die Ableitung der Funktion f als Funktions-Datei programmieren. Ausnahmsweise helfen Ihnen die Unterlagen hier noch beim Differenzieren:

$$f'(x) = -3\sin x - \frac{1}{x}$$

```
function y = meindf(x)
%MEINDF Ableitung der Funktion meinf
% es ist so fad, Kommentare
% zu schreiben, aber es zahlt
% sich aus, wenn man eine Woche
% spaeter versucht, zu verstehen,
% was das hier sein soll
y=-3*sin(x)-1./x;
end
```

Öffnen Sie ein neues M-file, programmieren Sie die Ableitung, und speichern Sie unter dem Namen `meindf.m` ab.

```
>> meindf(1)
ans =
  -3.52441295442369
>> x=1:3
x =
     1     2     3
>> meindf(x)
ans =
  -3.52441295442369
-3.22789228047704
-0.75669335751293
```

Prüfen Sie, ob sich die Ableitung korrekt auswerten lässt

Ein kleines, aber wichtiges Detail: Wenn Sie die Division als `1/x` programmieren, funktioniert die Auswertung für *skalare* x problemlos, nicht aber, wenn x ein Vektor ist! Anfangs ist es oft schwer zu durchblicken, wann MATLAB Multiplikationen und Divisionen von sich aus elementweise interpretiert, und wann man explizit den Punkt setzen muss.

> **Regel:** (War schon in der vorigen Einheit da[18])
> In Funktionsdateien immer die elementweisen Operatoren `.* ./ .^` verwenden.

[18] *repetitio est mater studiorum*, auf Deutsch „Wiederholung ist die Mutter der Langeweile“, oder so ähnlich...

Dann hätten wir allerdings oben gemäß der Regel auch die erste Multiplikation mit `.*` schreiben sollen:

```
y=-3.*sin(x)-1./x;
```

In diesem Fall ist es wurscht, ob man `.*` oder `*` schreibt: wenn einer der beiden Operanden ein Skalar ist, wird automatisch elementweise gerechnet.

Zurück zum Newton-Verfahren. Informieren Sie sich notfalls im Skriptum, Kapitel 1.10, über die Rechenvorschrift.

```
>> x=1
x =
     1
>> x=x-meinf(x)/meindf(x)
x =
   1.45990834177644
```

Beginnen Sie mit Startwert 1 und rechnen Sie einen Schritt des Newton-Verfahrens.

Wiederholen Sie den Newton-Schritt, bis sich der Wert der Nullstelle nicht mehr ändert!

Vergleichen Sie ihre Werte mit folgender Tabelle, und beachten Sie das Konvergenzverhalten:

	korrekte Stellen:
1	1
1.45990834177644	1.4
1.44725583798192	1.44725
1.44725861727779	1.447258617277
1.44725861727790	alle

Die Anzahl der richtigen Stellen ist in jedem Schritt mindestens doppelt so groß wie im vorhergehenden. Diese außerordentlich rasche Konvergenz (quadratische Konvergenz) ist charakteristisch für das Newton-Verfahren.

Ü 2.5.3 Nullstellen mit der Sekantenmethode

```
>> xalt = 2
xalt =
     2
>> x = 1
x =
     1
>> xneu = x - meinf(x)*(x-xalt)/(meinf(x)-meinf(xalt))
xneu =
   1.45499210414322
```

Wenn Sie nicht wissen, was das hier soll, ziehen Sie das Skriptum, Kapitel 1.9 oder Vorlesungsunterlagen zu Rate.

```
>> xalt = x; x = xneu;
```

Mit diesen Befehlen setzen Sie die beiden Startwerte der Sekantenmethode neu auf die beiden zuletzt berechneten Werte. Jetzt können Sie die Formel erneut auswerten (aber nicht neu eintippen, Pfeiltaste verwenden!)

```
>> xneu = x - meinf(x)*(x-xalt)/(meinf(x)-meinf(xalt))
xneu =
   1.44716725175157
```

Wiederholen Sie die Auswertung der Sekantenmethode, bis sich der Wert der Nullstelle nicht mehr ändert!

Vergleichen Sie ihre Werte mit folgender Tabelle, und beachten Sie das Konvergenzverhalten:

	korrekte Stellen:
1	1
1.45499210414322	1.4
1.44716725175157	1.447
1.44725862822699	1.4472586
1.44725861727792	1.4472586172779
1.44725861727790	alle

Die Anzahl richtiger Stellen, sagt die Theorie, ist die Summe der richtigen Stellen der beiden vorherigen Näherungen. Das würde typischer Weise 0, 1, 1, 2, 3, 5, 8, 13, ... genaue Stellen bedeuten (kommt ihnen diese Folge bekannt vor?) Dem entsprechend sollte sich pro Schritt die Anzahl der richtigen Stellen um etwa 60% erhöhen.

Tatsächlich verdoppelt sich die Anzahl bei den ersten Schritten (1-2-4-8), und erhöht sich schliesslich immer noch um 75% (von 8 auf 14 Stellen). Die Sekantenmethode konvergiert hier schneller, als sie es den Regeln der Theorie entsprechend müsste.

Ü 2.5.4 Nullstellen mit fzero

MATLAB hat *numerische* Verfahren zur Nullstellensuche eingebaut, eine Kombination aus Intervallhalbierung, Sekantenmethode und inverser quadratischer Interpolation. Die Funktion `fzero` ruft diese Verfahren auf.

Für unser Beispiel $f(x) = 3\cos(x) - \log(x)$ lautet der Aufruf:

```
>> fzero(@meinf,1)
ans =
   1.44725861727790
>>
```

fzero steht für „find zero“

Wichtig: Dem Funktionsnamen (hier: `meinf`) müssen Sie den „Funktionshenkel“ `@` voranstellen (MATLAB nennt `@` *function handle*). Funktionsnamen mit Henkel `@` davor sind ein eigener Datentyp, damit lassen sich Funktionen an andere Funktionen übergeben. Stellen Sie sich vor, `@` sei der Henkel, mit dem Sie ein Kaffeehäferl weitergeben.

Aber `fzero` hat seine Tücken: es findet einen Punkt so nah wie möglich bei einem Vorzeichenwechsel der Funktion. Für stetige Funktionen (Zwischenwertsatz!) ist das zugleich ein Wert nahe einer Nullstelle. Für unstetige Funktionen kann `fzero` Werte liefern, die zu Singularitäten der Funktion gehören. Beispiel: der Tangens bei $\pi/2$

```
>> fzero(@tan,1)
ans =
   1.57079632679490
>>
```

Das ist keine Nullstelle der Tangensfunktion

Mehrfache Nullstellen (gerader Ordnung), bei denen die Funktion die x-Achse berührt, aber nicht schneidet, kann `fzero` auch nicht finden.

Funktions-Henkel können noch mehr. Damit lässt sich ohne Skript, als Einzeiler, eine Funktion definieren. Das erklärt der nächste Abschnitt.

Ü 2.5.5 Funktions-Einzeiler, *Anonymous Functions*

Bei einem einzeiligen Funktionsterm ist es nicht nötig, eigene Funktionsdateien zu schreiben. Unsere Beispiel-Funktion f mit Funktionsterm $f(x) = 3\cos(x) - \log(x)$ und ihre Ableitung $f'(x) = -3\sin x - \frac{1}{x}$ lassen sich auch als sogenannte *anonymous functions* definieren. Das sind Funktionen, die nicht in einer Programmdatei gespeichert sind, sondern mit einer Variablen vom Typ *function handle* (erkennbar am „Funktionshenkel“ `@` mit Funktionsargument in Klammer) verknüpft sind. Für unsere Beispiele:

```
>> f = @(x) 3*cos(x) -log(x)
f =
  function_handle with value:
    @(x)3*cos(x)-log(x)

>> df = @(x) -3*sin(x)-1./x
df =
  function_handle with value:
    @(x)-3*sin(x)-1./x
```

Sie können diese Einzeiler-Funktionen auswerten

```
>> f(1)
ans =
    1.6209
```

oder direkt `fzero` anwenden. Unbedingt ausprobieren! Achtung, weil `f` bereits vom Typ „Funktionshenkel“ ist, ist beim Argument von `fzero` *kein* Henkel `@` mehr erlaubt.

```
>> fzero(f,1)
ans =
    1.4473
>> fzero(f,10)
ans =
   11.9702
```

Die Schritte eines Newton-Verfahrens im *command window* könnten dann beispielsweise so aussehen:

```
>> x=1;
>> x=x-f(x)/df(x)
x =
   1.459908341776445
>> x=x-f(x)/df(x)
x =
   1.447255837981919
>> x=x-f(x)/df(x)
x =
   1.447258617277790
>> x=x-f(x)/df(x)
x =
   1.447258617277903
```

> Funktions-Einzeiler (*anonymous functions*) sind praktisch, wenn der Funktionsterm aus einem einzigen ausführbaren Befehl besteht.
> Die Bezeichnung eines Funktions-Einzeiler ist eine Variable vom Typ „Funktions-Henkel“, sie bezieht sich *nicht* auf eine Funktionsdatei

Ü 2.5.6 Nullstellen mit roots

Lösungen (man sagt auch Wurzeln) *polynomialer* Gleichungen sind für MATLAB leichter zu finden. Betrachten wir als Beispiel

$$p(x) = x^3 - 2x - 5$$

Ein Polynom ist durch die Angabe seiner Koeffizienten bestimmt. In diesem Beispiel lauten sie

$$1; 0; -2; -5$$

weil $p(x) = 1 \cdot x^3 + 0 \cdot x^2 - 2 \cdot x - 5 \cdot x^0$. Sie übergeben `roots` die Liste der Koeffizienten als Vektor `[1 0 -2 -5]`, und roots liefert *alle* (auch die komplexen) Nullstellen des Polynoms.

```
>> roots([1 0 -2 -5])
ans =
   2.09455148154233
  -1.04727574077116 + 1.13593988908893i
  -1.04727574077116 - 1.13593988908893i
>>
```

Übungsbeispiele

Aufgabe 12:

Wie groß ist das Molvolumen von Stickstoff bei 20 C und 1 bar nach der Van der Waals-Gleichung?

Die Zustandsgleichung

$$\left(p + \frac{a}{V_{mol}^2}\right)(V_{mol} - b) = RT$$

beschreibt den Zusammenhang zwischen Druck p, Molvolumen V_{mol} und Temperatur T. Die Konstanten a und b haben für Stickstoff die Werte

$$a = 0{,}129\,\mathrm{Pa\,m^6/mol^2}, \quad b = 38{,}6 \times 10^{-6}\,\mathrm{m^3/mol}.$$

Die molare Gaskonstante ist $R = 8{,}3145\,\mathrm{J/molK}$. Nach Einsetzen der Zahlenwerte verbleibt als Gleichung für V_{mol}:

$$\left(100000 + \frac{0,129}{V_{mol}^2}\right)(V_{mol} - 0,0000386) = 2437,4$$

Lösen Sie diese Aufgabe mit der Sekanten- und der Newtonmethode und mittels `fzero`. Die Gleichung lässt sich auch auf polynomiale Form umformen. Verwenden Sie `roots` zur Lösung. Das Skriptum beschreibt in Kapitel 1.6 eine Umformung als Fixpunkt-Gleichung. Lösen Sie die Aufgabe auch mit Fixpunkt-Iteration!

Aufgabe 13:

Gegeben sei das Polynom

$$p(x) = -64 + 176\,x - 188\,x^2 + 101\,x^3 - 29\,x^4 + \frac{17\,x^5}{4} - \frac{x^6}{4}$$

Schreiben Sie dazu eine Funktions-M-Datei. Denken Sie dabei daran, die elementweisen Operatoren `.^` zu verwenden. Vergessen Sie nicht den Strichpunkt am Ende der Zeile (sonst liefert das M-file mächtig viel unnötige Ausgabezeilen im *Command Window*).

Zeichnen Sie das Polynom für $0 < x < 5$. Suchen Sie Nullstellen mit dem Newton-Verfahren, mit `fzero` (Sie müssen dazu geeignete Startwerte setzen) und mit `roots` .

Warum findet `fzero` nicht alle Nullstellen?

Überprüfen Sie, wie vorher in den Übungsunterlagen, die Konvergenzgeschwindigkeit des Newton-Verfahrens für alle drei Nullstellen. Verhält sich die Konvergenzgeschwindigkeit den Regeln entsprechend?

Ändern Sie im Polynom den Koeffizienten von x^3 um 1%; 0,1%; 0,01%. Bestimmen Sie Nullstellen mit `roots` und geben sie für jede ursprüngliche reelle Nullstelle an: Gibt es diese reelle Nullstelle noch? Wenn Ja, um wieviel hat sich ihr Wert relativ geändert? Oder hat sich eine mehrfache Nullstelle in mehrere verschiedene reelle Nullstellen aufgesplittet?

Diese Aufgabe illustriert: Die numerische Berechnung mehrfacher Nullstellen ist anfällig gegenüber kleinen Änderungen des Polynoms und Rundungsfehlern. Nullstellen verschwinden, verschieben oder vermehren sich. Man spricht von einem *schlecht konditionierten Problem*.

> **Schlecht konditioniertes Problem:** Kleine Änderungen in den Daten und/oder Rundungsfehler bewirken starke Änderungen im Ergebnis.

Ü 2.6 Kontrollstrukturen

Sie haben in den Aufgaben zur vorigen Übungseinheit Fixpunkt-Iterationen oder Iterationen des Newtonverfahrens oder der Sekantenmethode gleichsam „im Handbetrieb“ im *Command Window* eingegeben und, wenn die Ergebnisse sich nicht mehr geändert haben, das Verfahren beendet. Sie können diesen Ablauf auch als Programm formulieren. MATLAB bietet die üblichen Kontrollstrukturen (ähnlich wie in Java oder C++). Für den Anfang reichen `for`-Schleifen und `if`-Verzweigungen, wie sie dieser Abschnitt vorstellt. Mehr dazu, auch über `while`-Schleifen, erzählt Ihnen die MATLAB-Hilfe.

Schleifen

Eine for-Schleife hat zumeist die Form

```
for index = startwert:endwert
    anweisung
    anweisung
    ...
end
```

Der Index durchläuft dann die Werte `startwert, startwert+1, startwert+2...endwert`; er muss nicht (wie in Java oder C++) durch `index++` erhöht werden. Allgemeinere Form des Schleifenkopfes:

```
for index = startwert:schrittweite:endwert
```

Beispiel: ein Schleifenzähler s, der mit Schrittweite $-0,1$ die Werte $1; 0,9; 0,8; \ldots 0$ durchläuft:

```
for s = 1:-0.1:0
```

Verzweigungen

Eine bedingte Verzweigung mit `if` hat die Form

```
if ausdruck
    anweisung
    anweisung
    ...
end
```

MATLAB wertet „ausdruck“ aus, und wenn das Resultat logisch `true` oder (Unterschied zu Java!) ungleich 0 ist, führt es die folgenden Anweisungen bis zum `end` aus. Geschachtelte `if` sind möglich, jede Ebene muss mit dem entsprechenden `end` abgeschlossen werden.

Während Java sehr streng darauf achtet, dass in einer `if`-Anweisung nur ein logischer Ausdruck stehen darf, erlaubt MATLAB sogar Vektoren oder Matrizen. Wenn „ausdruck“ kein Skalar ist, muss jede einzelne Komponente `true` oder $\neq 0$ sein.

Die allgemeinere Form mit `elseif` und/oder `else` hat die Form

```
if ausdruck1
    anweisungen1
elseif ausdruck2
    anweisungen2
else
    anweisungen3
end
```

Musterprogramm

Das folgende Musterprogramm zur Fixpunkt-Iteration stellt Schleifen und Verzweigungen vor. Es führt eine Fixpunktiteration gemäß der Vorschrift

$$x^{(k+1)} = \phi(x^{(k)})$$

für den Startwert $x^{(0)}$ durch. Das Programm können Sie von der Übungs-Homepage herunterladen:

```
function x = fixpunkt(phi,x0)
%FIXPUNKT: Demo-Programm zur Fixpunkt-Iteration
% Das Verfahren iteriert gemaess der Vorschrift
%       x_i+1 = phi(x_i)
% bis Aenderungen unter eine Toleranzschwelle sinken
% oder maximale Iterationszahl ueberschritten wird.
%
% Eingabe   phi... Funktion, deren Fixpunkt gesucht ist
%                  (mit Funktions-Henkel @)
%           x0 ... Startwert oder -vektor
% Ausgabe   x .... Bei Konvergenz: Fixpunkt,
%                  sonst: NaN
% Beispiel  fixpunkt(@cos,1)
%
%-----------------------------------------NMI,SS09-18 C.Brand
itmax = 100;                % maximale Iterationszahl
errlim = 1.e-9;             % Fehlerschranke
for i=1:itmax
    x = phi(x0);            % Funktionsauswertung
    if norm(x-x0)<errlim    % Abbruchkriterium: 2-Norm des Fehlers
        return
    end
    x0 = x;
end
x = NaN;
end
```

Ü 2.7 Fixpunkt-Iteration ein- und mehrdimensional

Eindimensional

In Aufgabe 5 haben Sie die Qudratwurzel aus a als Fixpunkt der Funktion

$$y = \frac{1}{2}\left(x + \frac{a}{x}\right)$$

berechnet. Wir wiederholen, verwenden Schleifen und das Fixpunkt-Musterprogramm und verallgemeinern auf den mehrdimensionalen Fall.

```
>> heron = @(x)(x+13/x)/2;
```

Definieren Sie die Iterations-Funktion als *anonymous function*. Hier soll $\sqrt{13}$ berechnet werden. Alternativ können Sie auch eine Funktions-Datei dafür schreiben.

```
>> heron(2)
ans =
    4.2500
```

Testen Sie unbedingt, ob sich die Funktion aufrufen lässt und ein passendes Ergebnis liefert.

```
>> x=1;
>> x=heron(x)
x =
     7
```

So sieht eine Fixpunkt-Iteration im „Handbetrieb“ aus: wiederholter Aufruf der Funktion in der Befehlszeile.

```
>> x=heron(x)
x =
    4.4286
>> x=heron(x)
x =
    3.6820
>> x=heron(x)
x =
    3.6063
```

```
>> fixpunkt(heron,1)
ans =
    3.6056
```

Und so berechnet das Fixpunkt-Musterprogramm dasselbe Ergebnis.

Mehrdimensionale Fixpunkt-Iteration, vektorwertige Funktionen

Dieses Programm kann aber ebenso mehrdimensionale Fixpunkt-Iteration durchführen. In der Vorlesung wurde der Fixpunkt einer vektorwertigen Funktion $\mathbf{\Phi} : \mathbb{R}^2 \to \mathbb{R}^2$ als Beispiel behandelt:

$$\begin{aligned} x_1 &= \frac{1}{4}(x_2 - x_1 x_2 + 1) \\ x_2 &= \frac{1}{6}(x_1 - \log(x_1 x_2) + 2) \end{aligned}$$

Die Funktion $\mathbf{\Phi}(\mathbf{x})$ kann als Funktions-Datei so implementiert werden:

```
function y = phi(x)
x1=x(1);
x2=x(2);
y = [ (x2 - x1.*x2 + 1)/4
      (x1-log(x1.*x2)+2)/6 ];
end
```

(Bemerkung: Eine einzeilige *anonymous function* wäre schon auch möglich, aber bei einem etwas umfangreicheren Funktionsterm ist die Funktions-Datei die übersichtlichere Implementierung!)

Aufruf der Fixpunkt-Funktion für die Funktionsdatei (Henkel `@`+Dateiname!)

```
>> fixpunkt(@phi,[1;1])

ans =

    0.3534
    0.6400
```

Ü 2.8 Weitere Aufgaben bis zur nächsten Übungseinheit

Aufgabe 14:

Formulieren Sie die Iterationsvorschrift des Newton-Verfahrens zur Lösung von

$$\sin x = x/2 \ .$$

Programmieren Sie die entsprechende Iterations-Funktion und rufen Sie das Fixpunkt-Musterprogramm auf. Finden Sie damit alle positiven Lösungen der obigen Gleichung.

Aufgabe 15:

Suchen Sie die Nullstellen der Funktion

$$f(x) = e^x - 3x^2$$

mit Hilfe des Newton-Verfahrens und einfacher Fixpunkt-Iteration (mehrere Umformungen möglich).

Vergleichen Sie (bei denselben Startwerten) die Anzahl der Iterationen. Schätzen Sie bei den Verfahren mit linearer Konvergenz den Reduktionsfaktor C. Findet Ihre Fixpunkt-Formulierung alle Nullstellen?

Aufgabe 16:

Das Vorlesungsskriptum diskutiert die Gleichung

$$r = K\frac{q-1}{1-q^{-n}} \quad \text{für } r = 900, K = 100\,000, n = 180.$$

Schreiben Sie ein MATLAB-Programm, das allgemein für Eingabewerte r, K und n die Lösung q findet. (Annahme: $n \cdot r > K$ und daher $q > 1$) Welches Verfahren Sie verwenden, bleibt Ihnen überlassen.

Hinweise dazu:

- Der Aufzinsungsfaktor q liegt realistischer Weise im Bereich $1 < q < 1.05$, entsprechend einer monatlichen Verzinsung über 0% und unter 5%. Für $q = 1$ ist die Gleichung nicht definiert (Nenner wird 0), ein Startwert $q = 1$ ist nicht sinnvoll.
- Die Fixpunkt-Form (Auflösen nach q im Zähler) konvergiert langsam, ist aber rasch implementiert und vergleichsweise unempfindlich bezüglich der Wahl des Startwertes (sofern $q > 1$).

- Newton-Verfahren und MATLABs `fzero` brauchen gute Startwerte.

Aufgabe 17:

Formulieren Sie analog zum Fixpunkt-Musterprogramm eine Funktions-M-Datei zur Intervallhalbierung. Der Aufruf

```
IntervHalb(@igendeineFunktion, x0, x1)
```

soll, ausgehend von den Startwerten $x^{(0)}$ und $x^{(1)}$ eine Nullstelle der Funktion finden. Testen Sie das Verfahren, indem Sie Nullstellen folgender Funktion suchen:

$$f(x) = x^2 - 3\tan x + 1$$

Anfangsintervalle mit Vorzeichenwechsel sind: $[0, 1]$, $[1, 2]$, $[4.5, 4.7]$. Findet Intervallhalbierung für alle drei Intervalle eine Nullstelle? Was findet `fzero`, wenn Sie obige Intervallgrenzen als Startwerte geben?

Aufgabe 18:

Die Vorlesungsfolien zur 1. Vorlesung beschreiben die Illinois-Variante der Regula Falsi. Dieses Verfahren zeigt im Regelfall die guten Konvergenzeigenschaften der Sekantenmethode und bietet gleichzeitig den sicheren Einschluss der Nullstelle im aktuellen Intervall. Wenn Sie dieses Verfahren implementieren, haben Sie ein sehr brauchbares Allzweck-Nullstellen-Programm.

Orientieren Sie sich am Fixpunkt-Musterprogramm und schreiben Sie eine Funktions-M-Datei zur Regula Falsi in der Illinois-Variante. Der Aufruf

```
Illinois(@igendeineFunktion, x0, x1)
```

soll, ausgehend von den Startwerten $x^{(0)}$ und $x^{(1)}$ eine Nullstelle der Funktion finden. Testen Sie das Verfahren, indem Sie alle positiven Lösungen der folgenden Gleichung suchen:

$$\sin x = x/2$$

Rechnen Sie zum Vergleich auch mit `fzero` die Lösung.

Aufgabe 19:

Gegeben sei das Gleichungssystem

$$\begin{array}{rcrcrcr} 8x_1 & + & x_2 & - & x_3 & = & 8 \;, \\ 2x_1 & + & x_2 & + & 9x_3 & = & 12 \;, \\ x_1 & - & 7x_2 & + & 2x_3 & = & -4 \;. \end{array}$$

Formulieren Sie ein Fixpunktverfahren. Überlegen Sie, aus welcher Gleichung sie x_1 ausdrücken sollen, aus welcher x_2 und x_3. Hinweis: Unbekannte möglichst aus jener Gleichung ausdrücken, in der sie den stärksten Einfluss (den größten Koeffizienten) haben. Testen Sie das Verfahren.

Aufgabe 20:

(Wenn Sie die Unterlagen durchgearbeitet haben, ist diese Aufgabe eigentlich schon gelöst)

Gegeben sei ein System von zwei Gleichungen in zwei Unbekannten:

$$\begin{aligned} f(x, y) = 4x - y + xy - 1 &= 0 \\ g(x, y) = -x + 6y + \log xy - 2 &= 0 \end{aligned}$$

Formulieren Sie dafür ein Fixpunktverfahren und testen Sie!