Ü 3 Dritte Übungseinheit

Inhalt der dritten Übungseinheit:

- Script- und Funktionsdateien, Live Scripts
- Rechenoperationen bei Eingabe von Matrizen und Vektoren
- Bergabstrich löst Gleichungssysteme
- Inverses Problem, Regularisierung
- Newton-Verfahren für Systeme
- Lokale Funktionen
- `fsolve`
- Isolinien- und Oberflächengrafiken

Ü 3.1 Skript- und Funktionsdateien: Zusammenfassung

Wir haben schon mit beiden gearbeitet: Siehe Abschnitte Ü 1.3 und Ü 2.5 . Wichtige Unterschiede bestehen in der Verwendung und dem Gültigkeitsbereich von Variablen. Hier eine kurze Wiederholung und Zusammenfassung.

- Skripts. Sie haben weder Eingabe-Argumente noch Rückgabewerte. Sie greifen auf Daten und Variable des aktuellen Workspace zu. Befehlszeilen in einer Skript-Datei wirken so, als ob sie direkt im *Command Window* eingegeben würden.

  Ein häufiger Fehler ist, dass das Skript Variable aus dem aktuellen Workspace verwendet, aber nicht selber definiert. Sie merken während der aktuellen MATLAB-Sitzung nichts. Erst beim nächsten Neustart meldet MATLAB: `Undefined function or variable`. Geben Sie deswegen ganz zu Beginn des Skripts den Befehl `clear variables`. Damit löschen Sie alle Variablen im Workspace und merken sofort, ob sie Definitionen im Skript vergessen haben. Siehe Abschnitt Ü 1.3.

- Funktions-Dateien. Sie übernehmen Eingabewerte (Argumente)und liefern Resultate. Innerhalb der Funktion deklarierte Variable gelten nur lokal, also innerhalb der Funktion.

- Lokale Funktionen: In einer Funktionsdatei können weitere Funktionen („Unter-Funktionen“) vorkommen. Die erste Funktion (die „Hauptfunktion“) lässt sich von der Befehlszeile im *Command Window* aus oder über Befehlszeilen in einem Skript starten. Alle weiteren Funktionen (Reihenfolge egal) sind nur lokal – also nur für andere Funktionen aus derselben Datei – verfügbar. Es kann ganz praktisch sein, ein Programm mit Unterfunktionen zu gliedern. Weitere Infos: MATLAB-Hilfe, Stichwort *“local functions”*

- Auch Skript-Dateien können (ab MATLAB R2016b) lokale Funktionen enthalten; die dürfen aber erst nach der letzten Zeile des Skript-Codes beginnen. Hilfe dazu suchen Sie unter *“Add Functions to Scripts”*

- Verschachtelte Funktionen (für Fortgeschrittene): das sind Funktionen, die innerhalb einer übergeordneten Funktion deklariert sind. Damit können Sie Zugriff auf Variable noch differenzierter steuern. Details finden Sie in der MATLAB-Hilfe, Stichwort *“nested functions”*.

- Funktions-Einzeiler (*anonymous functions*) sind praktisch, wenn der Funktionsterm aus einem einzigen ausführbaren Befehl besteht. Wurden in Abschnitt Ü 2.5.5 vorgestellt.

- Live-Scripts und -Funktionen (Für Fortgeschrittene): Diese Art von Programm-Dateien kann Quellcode und Output gemeinsam darstellen, Begleittext und Kommentare übersichtlich formatieren, Sie können Schalt- und Steuerflächen für interaktives Arbeiten einbauen und mehr. Solche Dateien haben Endung `.mlx` im Unterschied zu Endung `.m` bei „gewöhnlichen“ Skripts.

  Wenn Sie Ihre Dateien besonders ansprechend gestalten wollen, können Sie ja versuchsweise im Editor auf *Save/Save as/Save as type: Matlab Live Code files (*.mlx)* klicken und so Ihre Datei in ein Live-Script umwandeln. Sie finden mehr Informationen und Beispiele in der MATLAB-Hilfe, Stichwort *Live Scripts and Functions*. Eine nette Ausarbeitung[17] von Aufgabe 8 finden Sie hier (klick!)

Ü 3.2 Rechenausdrücke bei der Eingabe von Vektoren und Matrizen

Eine Erinnerung an die letzte Einheit: Bei der Eingabe von Vektoren und Matrizen wirken Komma, Strichpunkt, Leerzeichen und Zeilenumbruch als Trennzeichen.

Sie können nicht nur Zahlenwerte, sondern auch Rechenausdrücke angeben:

```
>> x=[1+2, 2+3 3+4]
x =
     3     5     7
```

Zeilenvektoren: Leerzeichen oder Komma trennt Komponenten. Hier (nicht zu empfehlen) einmal Komma, einmal Leerzeichen.

```
>> y=[1-2 3 + 4 5 +6 7+ 8]
y =
    -1     7     5     6    15
```

Vorsicht mit Leerzeichen bei + und −: kann als Vorzeichen, als Rechenoperation oder als Trennzeichen interpretiert werden.

Typischer Fehler bei Rechenausdrücken in einem Vektor: Leerzeichen vor Operator wirkt ungewollt als Trennzeichen. Empfehlung: Bei Rechenoperationen entweder keine Leerzeichen oder beidseitig Leerzeichen um Operatoren.

Regeln bei Plus oder Minus in Termen:

- beidseitig kein Abstand: Rechenoperation wird ausgeführt (`+` und `-` wirken als *binäre Operatoren*)
- beidseitig Leerzeichen: ebenfalls als binäre Operation interpretiert.
- links Leerzeichen, rechts nicht: Leerzeichen links wirkt als Trennzeichen zwischen Matrix-Elementen, Operatoren `+` und `-` wirken als *unäre Operatoren* (als *Vorzeichen* des folgenden Terms).

  Passiert oft unabsichtlich, erzeugt Fehlermeldung oder falsches Ergebnis.
- rechts Leerzeichen, links nicht: binäre Operation (aber wenn Sie das absichtlich machen, ist das ziemlich verhaltensoriginell).

[17] von Chr. Tuschl, danke schön!

Ü 3.3 Gleichungssysteme: MATLABs schräge Schreibung

MATLAB verwendet – völlig normal – den Schrägstrich als Divisionsoperator: `x = 3/4` liefert wenig überraschend `x = 0.75000`.

Eher unüblich ist der andersrum gekippte „Bergabstrich“ `\` oder Backslash:

```
>> x=3\4
x =
    1.3333
```

Auch `\` dividiert, aber in der Form `x = Nenner\Zähler`. Es liegt ja auch wirklich, wenn Sie sich `\` als abwärts geneigten Bruchstrich vorstellen (oder den Bildschirm kurz mal nach links kippen), der linke Term *unter* und der rechte *über* dem Bruchstrich.

Grenzwertig originell ist jedoch MATLABs Interpretation des Bergabstrichs bei Gleichungssystemen. Das Wichtigste in Kürze:

`x = A\b` löst (oder „löst“) das Gleichungssystem $A\mathbf{x} = \mathbf{b}$

Nicht jedes von MATLAB so berechnete Ergebnis ist die Lösung eines Gleichungssystems im eigentlichen Sinn – deswegen die Anführungszeichen bei „löst“.

Niemand will sich im Detail merken, was genau MATLAB bei den verschiedenen Sonderfällen tut. Nur damit Sie sehen und gewarnt sind, was alles passieren kann, hier eine Aufzählung.

Der Befehl `x = A\b` liefert für ein Gleichungssystem $A\mathbf{x} = \mathbf{b}$

- bei nicht singulärer $n \times n$- Matrix die eindeutige Lösung;
- bei singulärer $n \times n$- Matrix eine Warnmeldung und, falls es Lösungen gibt, eine Lösung mit möglichst vielen Null-Komponenten. Falls es keine Lösung gibt, liefert MATLAB meistens unsinnige Zahlenwerte.
- bei einer $n \times m$- Matrix mit $n < m$ (*unterbestimmtes* Gleichungssystem), falls es Lösungen gibt, eine Lösung mit möglichst vielen Null-Komponenten.
- bei einer $n \times m$- Matrix mit $n > m$ (*überbestimmtes* Gleichungssystem), die „am wenigsten falsche Lösung“. Das ist jener Vektor $\mathbf{x}$, für den $A\mathbf{x} - \mathbf{b}$ die kleinste 2-Norm hat. Man spricht von der *Kleinste-Quadrate-Lösung*.
- Sonderfälle sind $n \times m$- Matrizen mit $n \neq m$ und Rang $< \min(m, n)$. Matlab warnt: „Warning: Rank deficient“ und liefert eine Kleinste-Quadrate-Lösung.

Gleichungssysteme mit mehreren rechten Seiten

Bei gleicher Matrix A und mehreren rechten Seiten $\mathbf{b}, \mathbf{c}, \mathbf{d}, \ldots$ lassen sich die Gleichungssysteme $A\mathbf{x} = \mathbf{b}, A\mathbf{y} = \mathbf{c}, A\mathbf{z} = \mathbf{d}, \ldots$ gemeinsam lösen. Stellen Sie alle rechten Seiten als Spaltenvektoren in einer Matrix B zusammen: `B = [b,c,d]`. MATLABs `\` liefert eine Matrix X, deren Spalten die Lösungsvektoren enthält: $X = [\mathbf{x}, \mathbf{y}, \mathbf{z}]$.

`X = A\B` löst (oder „löst“) die Gleichungssysteme $A \cdot X = B$

Schrägstriche zwischen Matrizen, allgemeiner Fall

Die Kurzfassung:

> Verwenden Sie den „Bergaufstrich“ `/` zwischen Skalaren als Divisionsoperator und den „Bergabstrich“ `\` zwischen Matrizen und Vektoren zum Lösen linearer Gleichungssysteme.

Wenn Sie mehr über MATLABs Umgang mit `/` und `\` wissen wollen, lesen Sie weiter…

Sie könnten $x = 3/4$ auch in der Form $x = 3 \cdot 4^{-1}$ schreiben, oder $x = 4^{-1} \cdot 3$. Niemand tut das bei skalaren Termen. Bei Matrizen ist die Schreibweise mit Inversen jedoch Standard, zum Beispiel $A \cdot B^{-1}$ oder $A^{-1} \cdot B$. MATLAB erlaubt dafür die Bruchstrich-Schreibung:

$$A \cdot B^{-1} = A/B \quad \text{ und } A^{-1} \cdot B = A\backslash B$$

Stellen Sie sich `/B` als B^{-1} vor, weil B „unter“ dem Bruchstrich liegt, und entsprechend `A\` als A^{-1}. Die Schrägstriche stehen absichtlich *links* von B und *rechts* von A – Matrixmultiplikation ist nicht kommutativ! Je nachdem, von welcher Seite Sie mit einer Inversen multiplizieren wollen, verwenden Sie `/` oder `\`.

MATLAB berechnet nicht wirklich die inversen Matrizen und multipliziert damit. Das explizite Ausrechnen einer Inversen ist rechenaufwändig und mit Rundungsfehlern behaftet. In Wirklichkeit löst MATLAB mit der schrägen Bruchstrichschreibweise lineare Gleichungssysteme. Das ist nur bei nichtsingulären quadratischen Matrizen algebraisch äquivalent zur Multiplikation mit der Inversen.

Gleichungssysteme statt inverser Matrix

Wenn in mathematischen Ausdrücken eine Multiplikation mit der inversen Matrix auftritt, brauchen Sie für die rechnerische Auswertung die Inverse in expliziter Form nicht wirklich[18].

Für das numerische Rechnen besteht ein gewaltiger Unterschied im Hinblick auf Rechenaufwand und -genauigkeit, ob Sie eine Inverse berechnen und damit multiplizieren, oder so umformen, dass Sie Gleichungssysteme lösen.

> Vermeiden Sie explizite Multiplikation mit einer inversen Matrix! Es verursacht unnötigen Rechenaufwand und unerwünschte Rundungsfehler.
> `X = A\B` berechnet $A^{-1} \cdot B$ als Lösung des Gleichungssystems $A \cdot X = B$
> `X = A/B` berechnet $A \cdot B^{-1}$ als Lösung des Gleichungssystems $X \cdot B = A$

Zur Erklärung, warum sich Multiplikation mit Inverser auf Lösung eines Gleichungssystems zurückführen lässt, hier die entsprechenden Umformungen. Achtung, man muss „auf der richtigen Seite“ multiplizieren, damit die Inversen verschwinden: im ersten Fall multipliziert man A von links, das andere mal mit B von rechts. (Matrixmultiplikation ist nicht kommutativ!)

$$\begin{aligned} X &= A^{-1} \cdot B && | \; A\cdot \\ A \cdot X &= A \cdot A^{-1} \cdot B \\ A \cdot X &= B \end{aligned} \qquad\qquad \begin{aligned} X &= A \cdot B^{-1} && | \; \cdot B \\ X \cdot B &= A \cdot B^{-1} \cdot B \\ X \cdot B &= A \end{aligned}$$

[18] Damit Sie erst gar nicht in Versuchung kommen, eine Inverse zu berechnen, verschweigen diese Übungsunterlagen (vorerst einmal) absichtlich den MATLAB-Befehl zur Berechnung der Inversen.

Ü 3.4 Inverse Probleme, Regularisierung

Was eine Matrix tut,
macht die Inverse wieder gut.

Abschnitt Ü 2.4 hat erklärt: Die Matrix-Vektor-Multiplikation $\mathbf{y} = A \cdot \mathbf{x}$ definiert eine Transformation *Eingabedaten* $\rightarrow$ *Bilddaten*. Oft liegt das Problem in umgekehrter Form vor: Gegeben ist der Ergebnisvektor $\mathbf{y}$, welcher Vektor $\mathbf{x}$ führt zu diesem Ergebnis?

Wenn die inverse Matrix existiert, ist die theoretische Antwort einfach:

$$\mathbf{y} = A \cdot \mathbf{x} \qquad \Leftrightarrow \qquad \mathbf{x} = A^{-1} \cdot \mathbf{y} \quad ,$$

Es kann aber sein, dass mehrere $\mathbf{x}$-Vektoren denselben Ergebnisvektor $\mathbf{y}$ haben, oder es überhaupt keinen $\mathbf{x}$-Vektor gibt, der exakt zum Ergebnisvektor $\mathbf{y}$ führt. Dann wird es schwierig. Aber selbst wenn eine Inverse theoretisch existiert, kann es sein, dass die Rücktransformation damit praktisch unmöglich ist.

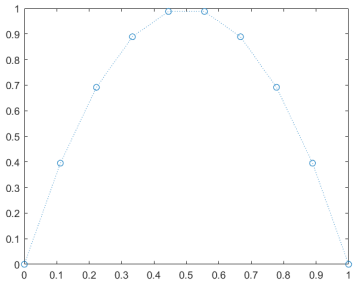
Inverses Problem: Ergebnis bekannt, Ursache gesucht. Oft schwierig.

Hier stellen wir Ihnen eine solche Situation vor, und zeigen, wie man damit umgehen kann.

Aufgabe 21: Schlecht konditioniertes inverses Problem

Erzeugen Sie eine Zeitreihe $\mathbf{x}(t)$ mit n Datenpunkten; hier mit parabelförmigen Verlauf; zeichnen Sie die Daten.

```
n=10;
t = linspace(0,1,n)';  % t Spaltenvektor!
x = 4*t.*(1-t);
plot(t,x,'o:')
```



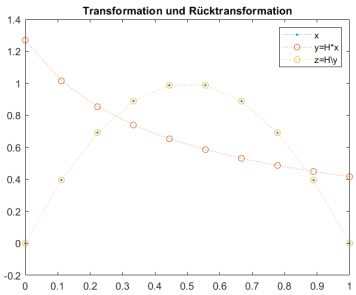
Wenden Sie auf $\mathbf{x}$ eine Transformation, die sogenannte Hilbert-Matrix, an: $\mathbf{y} = H \cdot \mathbf{x}$. Die Matrix ist einfach aufgebaut, ihre Inverse (sie enthält nur ganzzahlige Elemente) lässt sich explizit angeben. In MATLAB erzeuht sie der Befehl `hilb(n)` . Zum Beispiel für $n = 4$

$$H = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{bmatrix} \qquad H^{-1} = \begin{bmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{bmatrix}$$

Auf den Ergebnisvektor $\mathbf{y}$ wenden Sie gleich wieder die Rücktransformation an. Das heißt, Sie lösen das Gleichungssystem $H \cdot \mathbf{z} = \mathbf{y}$. Theoretisch[19] muss dann matürlich $\mathbf{z}$ ident mit dem Ausgangsvektor $\mathbf{x}$ sein. Zeichnen Sie daher $\mathbf{x}, \mathbf{y}$ und $\mathbf{z}$. In MATLAB führen Sie das so aus:

[19] Alte Lebensweisheit: *Theoretisch ist kein Unterschied zwischen Theorie und Praxis. Praktisch schon.*

```
H = hilb(n);
y = H*x;
z = H\y;
plot(t,x,'.:',t,y,'o:',t,z,'o:')
legend('x','y=H*x','z=H\y')
title('Transformation und Rücktransformation')
```



In der Abbildung hier, für $n = 10$ Datenpunkte, liegen die **x**- und **z**-Datenpunkte perfekt übereinander.

Wiederholen Sie die Aufgabe für größere n. Ab welchem n liefert die Rücktransformation deutlich unterschiedliche Werte, ab wann katastrophal unbrauchbare?

Die Hilbert-Matrix ist nur ein Beispiel für Transformationen, die auf extrem schlecht konditionierte inverse Probleme führen.

Sie sollen nun trotzdem für $n = 50$ die Rücktransformation durchführen. In solchen Fällen wenden Sie Regularisierungsmethoden an. Hier das Kochrezept für Tichonov-Regularisierung.

Statt

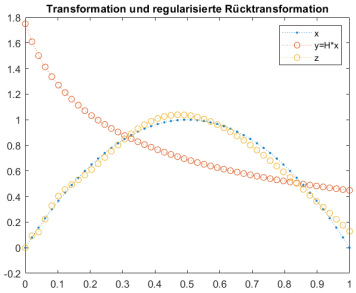
```
z = H\y;
```

verwenden Sie für die Rücktransformation

```
alpha= …
z = (H'*H + alpha*eye(n))\H'*y;
```

für ein *sehr* kleines $\alpha > 0$.

Für welchen Wert α können Sie $n = 50$ Datenpunkte transformieren und rücktransformieren, so dass in der Zeichnung **x** und **z** möglichst gut übereinstimmen? Hier rechts ist die Übereinstimmung noch nicht besonders gut. Bei günstig gewähltem α sollte sie deutlich besser sein.



Regularisierung für inverse Probleme tritt in vielen Anwendungsgebieten auf. Das ist ein hochaktuelles Thema, obwohl es dazu noch nicht einmal deutsche Wikipedia-Einträge gibt[20].

Die Übungsunterlagen gehen auf die Theorie nicht ein[21], sie wollen hier nur zeigen: selbst theoretisch anspruchsvollen Methoden lassen sich mit einfachen MATLAB-Befehlen ausführen.

[20] siehe die Stichworte *Regularization (mathematics)* und *Tikhonov regularization* in der englischen Wikipedia

[21] Erklärung der Bedeutung von α (nur für Interessierte): Der Ergebnisvektor **y** ist in der Praxis nicht völlig exakt bestimmbar. Wenn man relative Fehler der Größenordnung α im Ergebnis **y** zulässt, dann gibt es unendlich viele Ausgangsvektoren **x**, die im Rahmen der Genauigkeit den Ergebnisvektor **y** hätten erzeugen können. Der obige Kochrezept-Befehl findet unter all diesen Vektoren jenen mit kleinster 2-Norm.

Das ist zwar nicht notwendig der „richtige“ Ausgangsvektor, aber unter bestimmten Zusatzannahmen der „plausibelste“.

Ü 3.5 Newton-Verfahren für Systeme: Anleitung

Lösen Sie das folgende nichtlineare Gleichungssystem mit dem Newton-Verfahren.

$$\begin{aligned} 2x_1 - x_2 &= e^{-x_1} \\ x_1 + 2\sin x_2 &= \cos x_2 \end{aligned}$$

Es folgt eine Schritt-für-Schritt-Anleitung. Am Ende dieses Abschnittes gibt es auch ein Link zu einer fertigen Lösung. Dringender Rat: arbeiten Sie zuerst diese Anleitung durch (insbesondere, wenn Sie der Meinung sind, dass fertige Musterlösungen Zeit und Mühe sparen…)

Gleichung auf Nullstellen-Problem umformen

Bringen Sie zuerst die Gleichungen in die Form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

$$\begin{aligned} 2x_1 - x_2 - e^{-x_1} &= 0 \\ x_1 + 2\sin x_2 - \cos x_2 &= 0 \end{aligned}$$

Jacobi-Matrix berechnen

Berechnen Sie die Jacobi-Matrix: $D_{\mathbf{f}}(\mathbf{x}) = \begin{bmatrix} 2 + e^{-x_1} & -1 \\ 1 & 2\cos x_2 + \sin x_2 \end{bmatrix}$

Startvektor wählen

Beginnen Sie mit dem Startvektor $\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$. Werten Sie $\mathbf{f}$ und $D_{\mathbf{f}}$ für $\mathbf{x}^{(0)}$ aus – besser noch: lassen Sie MATLAB das tun (wird gleich erklärt).

Korrektur-Vektor berechnen

Sie berechnen einen Korrektur-Vektor $\Delta\mathbf{x}^{(0)}$ als Lösung eines linearen Gleichungssystems. Dessen Matrix ist die Jacobi-Matrix, auf der rechten Seite steht $-\mathbf{f}(\mathbf{x}^{(0)})$.

Zur Theorie lesen Sie bitte im Skriptum nach. Dort finden Sie:

> Iterationsvorschrift
>
> $$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \Delta\mathbf{x}^{(k)}$$
>
> mit $\Delta\mathbf{x}^{(k)}$ als Lösung von $D_f(\mathbf{x}^{(k)})\Delta\mathbf{x}^{(k)} = -\mathbf{f}(\mathbf{x}^{(k)})$

MATLAB löst lineare Gleichungssysteme mit dem „Bergab-Strich“-Operator `\`. Ein System $A\mathbf{x} = \mathbf{b}$ würden Sie in der Form `x = A\b` lösen. Für das Newton-Verfahren können Sie Gleichungslösen `Deltax = Df\(-f)` und Korrekturschritt `xneu = x + Deltax` als Einzeiler zusammenfassen: `xneu = x - Df\f`

Programmierung

Programmieren Sie $\mathbf{f}(\mathbf{x})$ und $D_{\mathbf{f}}(\mathbf{x})$ als MATLAB-Funktionen und speichern Sie die beiden als Dateien `f.m` und `Df.m`.

Abschnitte Ü 2.1 und Ü 2.2 haben die Eingabe von Vektoren und Matrizen schon kurz erklärt. Die Matrixelemente in einer Zeile können Sie durch Beistriche oder Leerzeichen trennen. Strichpunkte oder der Beginn einer neuen Zeile im Quelltext trennen die Zeilen in Matrizen und Spaltenvektoren.

Vervollständigen Sie die beiden folgenden Funktionsdateien!

```
function y = f(x)
y= [ ... %erste Komponente
     ... %zweite Komponente
   ];
end
```

Vektorwertige Funktion: Vektor als Eingabe, Vektor als Ergebnis!

```
function dy = Df(x)
dy= [ ... %erste Zeile
      ... %zweite Zeile
   ];
end
```

Matrixwertige Funktion: Vektor als Eingabe, Matrix als Ergebnis!

Prüfen Sie im *Command Window*, ob sich die beiden Funktionsdateien korrekt aufrufen lassen. Das sollte herauskommen:

```
>> x=[1; 1];
>> f(x)
ans =
    0.6321
    2.1426
>> Df(x)
ans =
    2.3679   -1.0000
    1.0000    1.9221
```

Achtung, setzen Sie Leerzeichen korrekt, wenn Sie die Jacobi-Matrix programmieren! Darauf weist schon Abschnitt Ü 3.2 hin; hier nochmal:

Leerzeichen um + und − bei Termen in Matrizen:
RICHTIG

- keine Lerzeichen. Beispiel `[ a+b, x+y ]`
- beiderseits Leerzeichen. Beispiel `[ a + b, x + y ]`

FALSCH:

- einseitig Lerzeichen. Beispiel `[ a +b, x +y ]`

Handbetriebene Iteration im Command Window

Wenn Funktion und Jacobi-Matrix korrekt programmiert sind, testen Sie im *Command Window* die Newton-Iteration. Beginnen Sie die Suche mit $\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

```
>> x=[1; 1];
>> x = x - Df(x)\f(x)
x =
          0.395158347619808
          0.199928444993336
>> x = x - Df(x)\f(x)
x =
          0.449399668569469
          0.261761474532574
>> x = x - Df(x)\f(x)
x =
          0.449562385013556
          0.261217530152151
>> x = x - Df(x)\f(x)
x =
          0.449562377948136
          0.261217503068493
```

Der Startwert ist mäßig genau. Die zweite Iteration liefert aber immerhin schon drei signifikante Stellen; ab dann lässt sich quadratisches Konvergenzverhalten beobachten.

Ü 3.6 Lokale Funktionen

Die beiden Funktionsdateien zusammen mit der Newton-Iteration sind in eine Datei zum Herunterladen gepackt. Dieses Programm zeigt Ihnen die Verwendung lokaler Funktionen.

> Ob Sie Aufgaben in einer Datei mit lokalen Funktionen oder mit anonymous functions programmieren, oder ob Sie Ihre Lösung lieber in Skripts und Funktionen in getrennten Dateien aufteilen, bleibt Ihnen überlassen! Alle Varianten haben Vor- und Nachteile.

Holen Sie sich die Datei und führen Sie sie aus. Für eine sauber formatierte Version können Sie die Datei auch in ein Live-Script umwandeln oder die *publish*-Möglichkeit von MATLAB nützen. Siehe MATLAB-Hilfe, Stichwort *Publish and Share MATLAB Code*. So sollte die Ausgabe aussehen:

```
>> NewtonMusterAufgabe
Konvergenz nach 4 Iterationen
Nullstelle:
          0.449562377948136
          0.261217503068493
```

Damit sollte es leicht fallen, die folgende Aufgabe zu lösen:

Aufgabe 22:

Aufgabe 20 fragt nach einer Lösung des Gleichungssystems

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

durch Fixpunkt-Iteration. Nun sollen Sie das Newton-Verfahren anwenden. Orientieren Sie sich am Beispiel des vorigen Abschnittes und am Musterprogramm `NewtonMusterAufgabe.m`. Im Skriptum (Seite 27) ist dieses Beispiel auch durchgerechnet.

Aufgabe 23:

Lösen Sie das folgende nichtlineare Gleichungssystem mit dem Newton-Verfahren. Startvektor $[1; -1; 0]$. Iterieren Sie, bis aufeinanderfolgende Lösungen in der ∞-Norm, das ist in MATLAB `norm(…,inf)`, auf 10^{-12} übereinstimmen.

$$\begin{aligned} x^2 + \sin^2 y + z &= 1 \\ e^x + e^{-x} - yz &= 2 \\ x + y + z^2 &= 0 \end{aligned}$$

Aufgabe 24:

Gegeben sind die Funktionsgleichungen

$$\begin{aligned} y &= 2x^2 - 1 \\ y &= \sin(3x) \quad \text{für } -1 \le x \le 1 \quad . \end{aligned}$$

Zeichnen Sie beide Kurven und lesen Sie aus der Graphik die Koordinaten der Schnittpunkte auf eine Nachkommastelle genau ab.

Formen Sie beide Gleichungen auf $\cdots = 0$ um und berechnen Sie die Lösung des nichtlinearen 2×2-Systems mit dem Newton-Verfahren.

Verwenden Sie die Näherungen aus der graphischen Darstellung als Startwerte. Konvergenzkriterium: sollen sich in keiner Komponente um mehr als 10^{-9} unterscheiden.

Aufgabe 25:

In einer wässrigen Natriumacetat-Lösung bestimmt ein nichtlineares System von vier Gleichungen die Konzentration der Ionen H^+, OH^-, Ac^- und der undissoziierten Säure HAc. Gegeben sind $K_w = 10^{-14}, K_s = 10^{-4,75}$ und eine variable Konzentration C.

$H^+ \cdot OH^-$	=	K_w	Ionenprodukt des Wassers
$H^+ \cdot Ac^-$	=	$K_s \cdot HAc$	Säure-Dissoziationsgleichgewicht
$HAc + Ac^-$	=	C	Acetat-Gesamtkonzentration
$Ac^- + OH^-$	=	$C + H^+$	Elektroneutralität

Wenn Sie die unbekannten Konzentrationen H^+, OH^-, Ac^-, HAc mit Variablen x_1, x_2, x_3, x_4 bezeichnen, lässt sich die Aufgabe umformulieren zu

$$\mathbf{f}(\mathbf{x}) = 0 \quad \text{mit} \quad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} -K_w + x_1 x_2 \\ x_1 x_3 - K_s x_4 \\ -C + x_3 + x_4 \\ C + x_1 - x_2 - x_3 \end{bmatrix}$$

Schreiben Sie eine MATLAB-Funktion `f(x,C)`, die für einen Eingabevektor $\mathbf{x} = (x_1, x_2, x_3, x_4)$ und gegebene Konzentration C den Vektor $\mathbf{f}(\mathbf{x})$ berechnet. ($K_w = 10^{-14}, K_s = 10^{-4,75}$ sind fix gegeben.) Schreiben Sie auch die entsprechende Funktion für die Jacobi-Matrix und ein Newton-Verfahren zur Nullstellenberechnung. Haben Sie eine Lösung gefunden, dann ist $-\log_{10} x_1$ der pH-Wert.

Bei diesem Beispiel sind gute Startwerte wichtig. Empfehlung:

$$\mathbf{x}^{(0)} = [10^{-7}; 10^{-7}; C; 0]$$

Zum Vergleich Testwerte für $C = 0.01$

```
>> x0=[1.e-7;1.e-7;0.01;0];
>> f(x0,0.01)
ans =
  1.0e-008 *

  -0.00000000000000
   0.10000000000000
                  0
                  0

>> df(x0)
ans =

   0.00000010000000   0.00000010000000                   0                   0
   0.01000000000000                   0   0.00000010000000  -0.00001778279410
                  0                   0   1.00000000000000   1.00000000000000
   1.00000000000000  -1.00000000000000  -1.00000000000000                   0

>> x0=x0-df(x0)\f(x0,0.01)
x0 =

   0.00000000035638
   0.00000019964362
   0.00999980071276
   0.00000019928724
```

Ü 3.7 Lösen von Systemen mit *fsolve*

Gleichungssysteme in der Form $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ kann MATLABs *Optimization Toolbox* mit dem Befehl `fsolve` lösen. Der Aufruf funktioniert ähnlich wie `fzero`.

Angenommen, Sie haben die Funktion aus Abschnitt Ü 3.5 als Funktionsdatei `f.m` programmiert. Dann lauten Aufruf und Ergebnis

```
>> fsolve(@f,[1,1])

Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the default value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>

ans =
 0.449562378440135 0.261217525788617
```

Sie ahnen vielleicht aufgrund der umfangreichen Ausgabe: so einfach kann das nicht gewesen sein. Ist es auch nicht – die zugrunde liegenden Methoden können wir im Rahmen dieser Übung nicht behandeln.

Ab der achten Stelle unterscheiden sich die Resultate von denen des Newton-Verfahrens. Wenn Sie auf hohe Genauigkeit Wert legen, müssten Sie die Standard-Einstellungen von `fsolve` anpassen[22].

Aufgabe 25 lässt sich mit Standardeinstellungen gar nicht lösen. (`fsolve` behauptet zwar auch „Equation solved“, aber das Resultat liegt total daneben.)

[22] Challenge: wer findet Einstellungen, so dass `fsolve` auf alle 16 double-Stellen genau dieselben Werte liefert wie das Newton-Verfahren? Ich hab probiert und es nicht geschafft. C.B.

Aufgabe 26:

Lösen Sie Aufgaben 22, 23 und 24 mit `fsolve`. Stellen Sie das Anzeigeformat im *command window* auf `format long g`, und vergleichen Sie die Genauigkeit mit den Resultaten des Newton-Verfahrens.

Ü 3.8 Isolinien- und Flächen-Diagramme

Aufgabe 27: Darstellung einer Funktion $z = f(x, y)$ durch Isolinien

Stellen Sie die Funktion

$$f : \mathbb{R}^2 \to \mathbb{R}, \quad z = xe^{-x^2-y^2}$$

als Isolinien-Diagramm dar. Am einfachsten geht das mit dem Befehl `ezcontour`: (amerikanisch-englisch ausgesprochen klingt das wie *ea-sy-contour*)

```
>> ezcontour('x*exp(-x^2␣-␣y^2)')
```

Achtung, Hochkommas ' nicht vergessen!

Wenn Sie aber genauer festlegen wollen, wie und was sie darstellen wollen, verwenden Sie den „richtigen“ Befehl `contour`. Die MATLAB-Hilfe zum Stichwort `contour` erklärt, wie das geht. Sie können nach einem dort angegebenen Musterbeispiel (je nach Matlab-Version scrollen Sie drei, vier Abbildungen nach unten) arbeiten:

Für ein Isolinien-Diagramm (engl: *contour plot*) der Funktion $z = xe^{-x^2-y^2}$ im Bereich $-2 \leq x \leq 2, -2 \leq y \leq 3$ erzeugen Sie zuerst x- und y-Werte auf einem Gitter in der xy-Ebene.

```
>> x = -2:0.2:2;
>> y = -2:0.2:3;
>> [X,Y] = meshgrid(x,y);
```

Beachte: in englischen Texten steht oft `.2` (ohne 0 vor dem Dezimalpunkt), wo bei uns 0,2 stehen würde. Auch der MATLAB-Ausdruck `-2:.2:2` ist korrekt.

Für die (x, y)-Wertepaare berechnen Sie eine Matrix Z mit dem Befehl

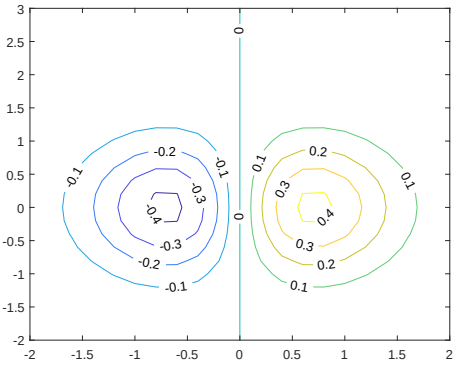
```
>> Z = X.*exp(-X.^2-Y.^2);
```

Anschließend erzeugen Sie den Isolinien-Plot:

```
>> contour(X,Y,Z,'ShowText','on')
>> colormap cool
```

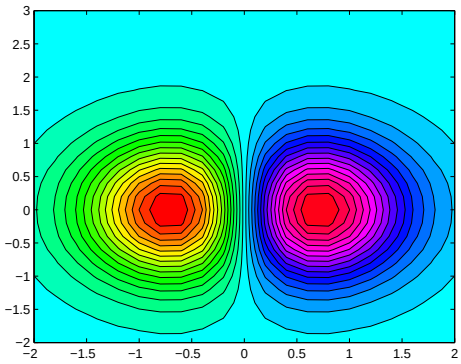
Es gibt verschiedene Varianten des contour-Befehls. In der MATLAB-Hilfe finden Sie weitere Beispiele (Befehle contourf, contour3)

So sollte das Isolinien-Diagramm aussehen, das Sie mit den obigen Befehlen erzeugt haben.



Weitere Varianten des `contour`-Befehls: `contour(X,Y,Z,30)` erzeugt 30 Isolinien; `contour(X,Y,Z,-1:0.1:1)` erzeugt Isolinien für Werte von -1 bis 1 in 0,1-Schritten.

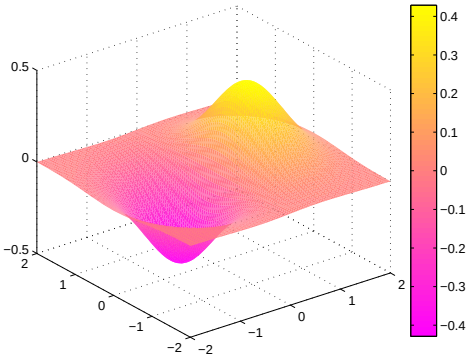
Erforschen Sie die Optionen des `contour`-Befehles. Ihre Aufgabe ist, herauszufinden, mit welchen Befehlen sie den Plot möglichst genau so aussehen lassen, wie hier gezeigt:



Aufgabe 28: Darstellung einer Funktion $z = f(x, y)$ als Fläche im Raum

Ganz ähnlich wie `contour` funktioniert der Befehl `surf`. Informieren Sie sich in der MATLAB-Hilfe und zeichnen Sie die Funktion aus Aufgabe 27 als Fläche im Raum.

Ihre Aufgabe ist, herauszufinden, mit welchen Befehlen sie den Plot möglichst genau so aussehen lassen, wie hier gezeigt (Datenbereich, Auflösung, Farbgebung, Farbbalken…).



Ü 3.9 Graphische Suche nach Nullstellen für nichtlineare Gleichungssysteme in zwei Variablen

Für eine Funktion $f : \mathbb{R} \to \mathbb{R}$ in der Form $y = f(x)$ lassen sich Nullstellen aus dem Funktionsgraph ablesen. Das kennen Sie schon aus der Mittelschule und der ersten Vorlesung.

Eine Funktion $f : \mathbb{R}^2 \to \mathbb{R}$ können Sie in der Form $z = f(x, y)$ als Fläche im Raum oder als Isoliniengrafik darstellen und so ebenfalls graphisch die Lage von Nullstellen finden. Eine Nullstelle ist in diesem Fall ein Wertepaar (x, y), das die Gleichung $f(x, y) = 0$ erfüllt.

Für ein System zweier nichtlinearer Gleichungen in der Form

$$f_1(x, y) = 0$$
$$f_2(x, y) = 0$$

kann man die Funktion

$$f(x, y) = [f_1(x, y)]^2 + [f_2(x, y)]^2$$

definieren. Die ist genau dann gleich 0, wenn f_1 und f_2 gleich Null sind. Kleiner als 0 kann f (aufgrund ihrer Definition als Summe zweier Quadrate) auch nicht werden. Die Nullstellen von f sind also zugleich auch globale Minima von f und Lösungen des nichtlinearen Gleichungssystems $f_1 = 0$, $f_2 = 0$.

Die nächsten beiden Aufgaben illustrieren diesen Zusammenhang.

Einleitung zu den Aufgaben 29 und 30

Die Funktion $f : \mathbb{R}^2 \to \mathbb{R}$, gegeben durch

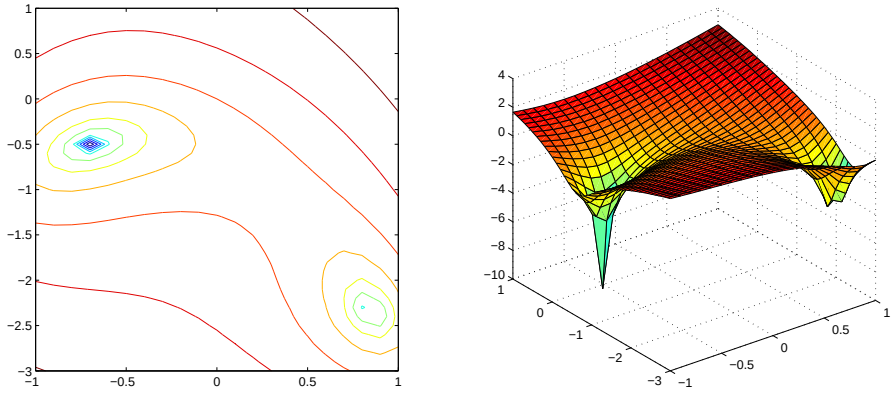
$$f(x, y) = \left[x^2 + \sin y\right]^2 + \left[e^x + y\right]^2$$

hat Minimalwert 0 für genau jene Wertepaare $(x; y)$, die Lösung des folgenden nichtlinearen Gleichungssystems sind:

$$\begin{aligned} x^2 + \sin y &= 0 \\ e^x + y &= 0 \end{aligned}$$

Aufgabe 29: Nullstellen einer Funktion in zwei Variablen (Teil 1)

Zeichnen Sie für $\log f$, den Logarithmus der oben gegebenen Funktion, im Bereich $-1 < x < 1$ und $-3 < y < 1$ eine `contour`- und eine `surf`-Grafik in der Art der Abbildung. (In der logarithmischen Darstellung erkennt man die Lage des Minimums deutlicher als in direkter Darstellung von f.) Verwenden Sie ≈ 20 Gitterpunkte in x- und ≈ 40 Gitterpunkte in y-Richtung.



Aufgabe 30: Nullstellen einer Funktion in zwei Variablen (Teil 2)

Lösen Sie das oben gegebene nichtlineare Gleichungssystem mit dem Newton-Verfahren. Finden Sie beide Lösungen im Bereich $-1 < x < 1$ und $-3 < y < 1$. Passende Startwerte können Sie aus der grafischen Darstellung (Aufgabe 29) entnehmen. Abbruchkriterium: Aufeinanderfolgende Iterationen unterscheiden sich, gemessen als Summe der Beträge aller Komponenten-Differenzen, weniger als 10^{-8}.