ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Kommunikationstechnik

Prof. Dr. H. Bölcskei
Sternwartstrasse 7
CH-8092 Zürich

PPS "Bits on Air"
1. Teil, Matlab-Tutorial
Vorbereitungsaufgaben

Markus Gärtner, Samuel Brändle & Patrick Kuppinger

Revidierte Version vom 16. August 2016

1 Einleitung

Im ersten Teil des Praktikums wirst Du Dich mit der Programmierumgebung von `Matlab` vertraut machen. Da alle folgenden Geräte und Prozesse in `Matlab` simuliert werden, ist es wichtig, dass Du in diesem Teil die Funktionen und vor allem die Arbeitsweise von `Matlab` verstehst. Du wirst den Umgang mit Variablen und Matrizen üben, lernen, wie man in `Matlab` mit ihnen rechnen kann, wie man die erhaltenen Lösungen graphisch darstellt und wie man eigene Funktionen schreibt.

Dieses `Matlab`-Tutorial ist speziell auf dieses Praktikum zugeschnitten. `Matlab` bietet eine Fülle von Funktionen an, von denen hier nur wenige behandelt werden. In den Vorlesungen "Analysis III" und "Numerische Methoden" und eventuell auch in anderen PPS-Veranstaltungen wirst Du weitere Funktionen von `Matlab` kennenlernen.

`Matlab` eignet sich bestens zum Rumspielen. Wenn Du Dich fragst, wie etwas funktioniert, dann probier's einfach aus (diese Aufforderung wirst Du in diesem Tutorial noch etliche Male lesen)!

Dieses Tutorial besteht einerseits aus Erklärungen und andererseits aus Aufgaben, welche mit dicken Punkten gekennzeichnet sind. Die verwendeten Matrizen werden mit einem einzelnen Grossbuchstaben bezeichnet. Mit diesen Variablen solltest Du nur machen, was im Text beschrieben ist und sie sonst unverändert lassen, da viele von ihnen später wieder benötigt werden. Möchtest Du in eigenen Versuchen Variablen verwenden, so gib ihnen Namen mit mehreren Buchstaben (zulässige Variablennamen in `Matlab` bestehen aus einem Buchstaben gefolgt von beliebig vielen weiteren Buchstaben, Ziffern oder Underscores, andere Sonderzeichen sind nicht erlaubt).

2 Operationen auf Matrizen

2.1 Eingeben von Matrizen

In `Matlab` sind alle Variablen Matrizen. Ein Zeilenvektor ist eine $1 \times n$-Matrix, ein Spaltenvektor eine $n \times 1$-Matrix und ein Skalar ist schlicht eine 1×1-Matrix. Gib einmal den Befehl `A = [1,2,3; 4,5,6; 7,8,9]` ein und drücke Enter. `Matlab` wird Dir die 3×3-Matrix anzeigen, welche in der Variable A gespeichert wurde. Wie Du siehst, werden Matrizen in eckigen Klammern eingegeben, die einzelnen Zeilen werden durch ein Semikolon getrennt und die einzelnen Elemente innerhalb einer Zeile durch Kommata. Statt Letzteren können wir auch Leerschläge verwenden, der Befehl `A = [1 2 3; 4 5 6; 7 8 9]` hätte genau denselben Effekt. Unsere 3×3-Matrix ist nun gespeichert und wir können sie über die Variable `A` wieder aufrufen. Gib einmal `A` ein und drücke Enter, die Matrix wird erneut angezeigt. Beachte, dass `Matlab` case-sensitive ist, `A` ist also nicht gleich `a`. Probier's aus, indem Du versuchst, Dir den Inhalt der Variable `a` anzeigen zu lassen.

`Matlab` bietet eine Reihe von Funktionen an, welche das Erstellen von häufig benötigten Matrizen erleichtern. Der Befehl `B = eye(3)` erstellt beispielsweise die 3×3-Identitätsmatrix und speichert sie in der Variablen `B`, probier's aus. Über `help` *Funktionsname* kannst Du Dir die Hilfe zu einer Funktion anzeigen lassen (Funktionsnamen werden in den Hilfstexten jeweils in Grossbuchstaben geschrieben, dies ist jedoch nur ein Mittel zur graphischen Hervorhebung, die Funktionen müssen in der Kommandozeile in Kleinbuchstaben eingegeben werden). Benutze die Hilfe, um herauszufinden, was die Funktionen `eye`, `ones` und `zeros` machen, und probiere sie aus. Du brauchst die erstellten Matrizen beim Ausprobieren nicht jedes Mal in einer speziellen Variable zu speichern. Gibst Du einen Befehl ohne Zuweisung ein (z.B. `ones(2,5)`), so speichert `Matlab` das Ergebnis in der Variable `ans`.

`Matlab` kann auch bereits gespeicherte Matrizen zu neuen zusammensetzen. `C = [A B]` (mit `A` und `B` von vorhin) erstellt beispielsweise eine 3×6-Matrix und `D = [A zeros(3,2); ones(1,5)]` eine 4×5-Matrix, probier's aus. Wichtig ist, dass `Matlab` zwar die Werte für die neuen Matrizen aus `A` holt, dann aber sofort "vergisst", dass sie von dort stammen. Änderst Du also nachträglich A, so bleiben C und D davon unbeeinflusst! Beachte bitte auch, dass die Dimensionen der Submatrizen, die Du zusammensetzen möchtest, kompatibel sein müssen, d.h. Submatrizen in derselben Zeile müssen beide dieselbe Zeilenzahl aufweisen, solche in derselben Spalte dieselbe Spaltenzahl. Eine Zusammensetzung wie `[A eye(2)]` funktioniert nicht.

Eine weitere Möglichkeit, Vektoren, deren Elemente einer bestimmten Regelmässigkeit folgen, einzugeben, besteht in der Doppelpunkt-Notation. Dabei gibt man den Wert des ersten Elementes an, eine Schrittweite und den Wert des letzten Elementes - jeweils durch einen Doppelpunkt getrennt. `[10:2:18]` erstellt z.B. den Vektor `[10 12 14 16 18]`, `10` ist das Startelement, `2` die Schrittweite und `18` das Endelement. Gibt man keine Schrittweite an, so wählt `Matlab` eins als Schrittweite; `[1:n]` liefert also beispielsweise einen Zeilenvektor mit allen ganzen Zahlen von `1` bis und mit `n`, probier's aus. Mit einem Apostroph nach der Matrix kannst Du übrigens die Transponierte bilden, also z.B. `D'` oder `[1:5]'`. Verwende bei den folgenden Aufgaben die Doppelpunkt-Notation:

- Erstelle einen Zeilenvektor, der die "Siebnerreihe" (von 7 bis 70) enthält.

- Erstelle einen Spaltenvektor, der aus allen ungeraden Zahlen zwischen 17 und 33 in aufsteigender Reihenfolge besteht.
- Erstelle einen Zeilenvektor, der alle Zahlen zwischen 87 und 141 enthält, die durch 5 teilbar sind - in absteigender Reihenfolge.

2.2 Zugriff auf einzelne Matrixelemente und Submatrizen

`Matlab` erlaubt uns, auf einzelne Matrixelemente und Submatrizen zuzugreifen. Dies geschieht durch Angabe der gewünschten Zeile(n) und Spalte(n) in runden Klammern. Mehrere Zeilen/Spalten können entweder einzeln genannt werden oder wir verwenden die Doppelpunkt-Notation. Der Doppelpunkt alleine bedeutet "alle Zeilen" (also eine ganze Spalte) resp. "alle Spalten" (also eine ganze Zeile). Ein paar Beispiele zur Veranschaulichung (wir verwenden die vorher definierten Matrizen `A`, `B` und `C`, lass Dir die doch zuerst alle nochmals anzeigen):

`C(2,3)` ist das dritte Element in der zweiten Zeile von `C` (Zeile 2, Spalte 3).

`C(1,1:4)` ist ein Zeilenvektor mit den ersten vier Elementen aus der ersten Zeile von `C` (Zeile 1, Spalten 1 bis 4).

`C(2:3,1:2)` ist die 2×2-Submatrix in der "unteren linken Ecke" von `C` (Zeilen 2 und 3, Spalten 1 und 2).

`C(:,3)` ist die gesamte dritte Spalte von `C` (alle Zeilen, Spalte 3).

`C(1:2,:)` sind die ersten zwei Zeilen von `C` (Zeilen 1 und 2, alle Spalten).

`C(:,[2 4])` ist eine 3×2-Matrix, welche die Spalten zwei und vier von `C` enthält (alle Zeilen, Spalten 2 und 4).

Probier's aus! Diese Notation kann auf beiden Seiten einer Zuweisung benutzt werden, also sowohl für Lese- als auch für Schreibzugriffe:

`B(:,[1 3]) = A(:,1:2)` ersetzt Spalten eins und drei von `B` durch die ersten beiden Spalten von `A`. Beachte, dass als Ergebnis die gesamte neue Matrix `B` angezeigt wird und nicht nur die veränderten Spalten.

Bei solchen Zuweisungen müssen die Submatrizen auf beiden Seiten natürlich dieselben Dimensionen aufweisen. Arbeiten wir mit einem Zeilen- oder Spaltenvektor, so genügt die Angabe einer Zahl zur Bestimmung eines Elementes. Nehmen wir z.B. `E = [4:2:18]`:

`E(5)` ist das fünfte Element von `E`.

`E(3:6)` ist ein Zeilenvektor mit den Elementen drei bis sechs von `E`.

`E([1 3 8])` sind die Elemente eins, drei und acht von `E` in einem Zeilenvektor.

Beachte, dass in `Matlab` die Numerierung mit eins beginnt (im Gegensatz zu Programmiersprachen wie C, wo das erste Element eines Arrays den Index null besitzt). Versuche mal, Dir das Element null von `E` anzeigen zu lassen.

In `Matlab` kannst Du übrigens wie in der UNIX-Shell mit den auf- und ab-Tasten (↑, ↓) durch die letzten eingegebenen Befehle scrollen, sie verändern und erneut ausführen. Ebenso kannst Du `Matlab` mit Tab einen Befehl vervollständigen lassen. Kann der Befehl noch nicht eindeutig identifiziert werden, so bietet Dir `Matlab` nach einem weiteren Drücken der Tab-Taste eine Auswahl an.

- Auch bei $m \times n$-Matrizen kannst Du lediglich eine einzige Zahl zur Bestimmung eines Elementes angeben. Versuche herauszufinden, wie die Matrixelemente in diesem Fall ausgewählt werden (was ist `C(3),C(4),C(2:5),C(:)`?).
- Ersetze die 8 in `C` durch eine 12.
- Ersetze die dritte Spalte von B durch lauter Dreien.
- Ersetze die Elemente zwei bis vier in der letzten Zeile von `C` durch die zweite Zeile von `A`.
- Ersetze die letzte Spalte von `C` durch die Transponierte der ersten Zeile von `A`.
- Was bewirkt der Befehl `F = E(8:-1:1)`?

2.3 Arithmetische Operationen

Matrizen gleicher Grösse können addiert (+) und subtrahiert (−) werden, was bekanntlich elementweise geschieht. Weiter kann eine Matrix mit einem Skalar multipliziert (`*`) oder durch einen Skalar dividiert (`/`) werden, diese Operationen werden ebenfalls elementweise ausgeführt.

Werden zwei Matrizen multipliziert (z.B. `A*B`), so führt `Matlab` eine Matrixmultiplikation aus. Diese ist gemäss der Definition nur möglich, wenn die Spaltenzahl der ersten Matrix mit der Zeilenzahl der zweiten Matrix übereinstimmt, ansonsten erscheint eine Fehlermeldung. Neben dieser Matrixmultiplikation können Matrizen gleicher Dimension auch elementweise multipliziert werden; dies geschieht durch Voranstellen eines Punktes vor den Multiplikationsoperator (`A.*B`). Probier's aus. Analog werden Matrizen gleicher Dimension elementweise dividiert (`A./B`).

- Was geschieht, wenn Du einen Skalar zu einer Matrix addierst (z.B. `A+3`)?
- Erstelle eine 9×6-Matrix, die lauter Fünfen enthält.
- Was machen die Operationen `A^2` und `A.^2`?
- Erstelle einen Spaltenvektor, der alle Zweierpotenzen von 2 bis 1024 in aufsteigender Reihenfolge enthält.

3 Funktionen

3.1 Skalarfunktionen

Skalarfunktionen operieren auf einzelnen Zahlen. Werden sie auf eine Matrix angewandt, so werden sie elementweise ausgewertet. Beispiele für diese Klasse von Funktionen sind die trigonometrischen Funktionen `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, die Exponentialfunktion `exp`, der natürliche Logarithmus `log`, die Modulo-Funktion `mod`, die Betragsfunktion `abs`, die Wurzelfunktion `sqrt`, die Signum-Funktion `sign` und die Rundungsfunktionen `round`, `floor`, `ceil` und `fix`. Die Argumente werden den Funktionen in Klammern übergeben, also z.B. `sqrt(A)`. Funktionen können auch ineinander verschachtelt werden, z.B. `round(exp(A))`, probier's aus.

Wenn Du einen Befehl mit einem Semikolon abschliesst (z.B. `G = abs(A-10);`), so wird die Berechnung zwar ausgeführt, das Ergebnis aber nicht angezeigt (probier's aus und rufe nachher `G` auf). Später werden wir mit sehr grossen Matrizen arbeiten, bei denen das Anzeigen von Zwischenresultaten oft länger dauert als deren Berechnung. Wir werden also praktisch alle Befehle mit einem Semikolon abschliessen und uns nur die Endresultate (meist graphisch) darstellen lassen.

- Benutze die Hilfe, um die Unterschiede zwischen den Rundungsfunktionen `round`, `floor`, `ceil` und `fix` zu ergründen und probiere sie aus.
- Finde eine 2×2-Matrix `H`, so dass `round(H)` gleich `ceil(H)` ist, aber `round(H)` ungleich `floor(H)`.
- Welche Bedingung müsste `H` erfüllen, damit `fix(H)` gleich `ceil(H)` ist?

3.2 Vektorfunktionen

Ein weiterer Typ von Funktionen operiert auf Vektoren (Spalten- oder Zeilenvektoren). Wird eine solche Funktion auf eine Matrix angewandt, so wird sie Spalte für Spalte ausgewertet, das Ergebnis ist ein Zeilenvektor, der die Ergebnisse für die einzelnen Spalten enthält. Beispiele für Vektorfunktionen sind `max` (bestimmt das grösste Element eines Vektors), `min` (bestimmt das kleinste Element eines Vektors), `sum` (berechnet die Summe der Vektorelemente) und `prod` (berechnet das Produkt der Vektorelemente). Probier's aus, z.B. mit `sum(A)`, `prod(C)`, `max(D)`.

Was machen wir aber, wenn wir die Summe *aller* Matrixelemente bestimmen möchten? Wir wenden die entsprechende Vektorfunktion einfach zweimal an, also `sum(sum(A))`! Zuerst werden die Spaltensummen bestimmt und im zweiten Schritt die Summe dieser Spaltensummen. Probier's aus.

Wie in C können Funktionen auch in `Matlab` "überladen" sein, d.h. sie verhalten sich je nach Anzahl und Art der Parameter unterschiedlich. In `Matlab` gibt es sogar Funktionen, die mehrere Resultate zurückgeben, wenn man die Zuweisung entsprechend gestaltet. `[m,p] = max(E)` beispielsweise speichert in `m` den Wert des grössten Elementes von `E` und in `p` dessen Position, probier's aus.

- Was ist das Ergebnis von `max([7 2 1 9], [4 5 -6 -2])`? Probier's aus und konsultiere die Hilfe.
- `sum(A)` liefert uns einen Zeilenvektor mit den Spaltensummen von `A`. Wie erhalten wir einen Spaltenvektor mit den Zeilensummen von `A`?
- Erweitere `D` mit einer fünften Zeile, welche die Spaltensummen der ursprünglichen Matrix `D` enthält.

3.3 Matrixfunktionen

Dieser dritte Typ von Funktionen operiert auf Matrizen. Da wir vor allem mit Vektoren arbeiten werden, benötigen wir eigentlich nur zwei Matrixfunktionen: `size` und `length`. `size(C)` gibt einen Zeilenvektor mit zwei Elementen zurück, der Zeilenzahl und der Spaltenzahl von `C`. `length(C)` ist gleich `max(size(C))`, kann also verwendet werden, um die Länge eines Vektors zu bestimmen.

Weitere Matrixfunktionen sind beispielsweise `inv` (Inverse), `det` (Determinante), `eig` (Eigenwerte und -vektoren), `norm` (Norm), `cond` (Kondition), `lu` (LR-Zerlegung), `qr` (QR-Zerlegung) und `svd` (Singulärwertzerlegung).

- `diag` ist ebenfalls eine Matrixfunktion. Was machen `diag(A)`, `diag([1 2 3])` und `diag(diag(A))`?
- Was steht nach `[U,V] = eig(A)` in `U` und `V`? Was ist `U*V*inv(U)`?

3.4 Zufallszahlen

Die Funktion `rand(m,n)` erzeugt eine $m \times n$-Matrix, deren Elemente gleichverteilte Zufallszahlen zwischen null und eins sind, probier's aus.

- Erzeuge einen Spaltenvektor mit 10 Zufallszahlen zwischen 0 und 700.
- Erzeuge eine 4×7-Matrix mit zufälligen ganzen Zahlen zwischen -10 und +10.
- Erzeuge einen Zeilenvektor mit 15 ungeraden Zufallszahlen zwischen 0 und 100.

4 Vergleiche, Bedingungen und Schleifen

4.1 Vergleiche und Bedingungen

Mit den Operatoren < (kleiner als), > (grösser als), <= (kleiner gleich), >= (grösser gleich), == (gleich) und ~= (ungleich) können Matrizen gleicher Dimension verglichen werden. Der Vergleich wird dabei elementweise durchgeführt und liefert 1 (für true) und 0 (für false) zurück. Probier's aus, z.B. mit `A == tril(A)` (`tril(A)` ist die untere Dreiecksmatrix von `A`).

Die Ergebnismatrix enthält an den Stellen Einsen, an denen die Elemente der beiden verglichenen Matrizen die Bedingung erfüllen, und an allen anderen Nullen. Vergleiche können mit den logischen Operatoren `&` (und), `|` (oder) und `~` (nicht) verknüpft werden.

Beachte, dass wie in C eine Zuweisung durch ein einzelnes Gleichheitszeichen erfolgt, während für einen Vergleich deren zwei verwendet werden.

Mit den Vektorfunktionen `any` und `all` können wir Matrixvergleiche auf Vektoren oder Skalare reduzieren. `any(E)` ist dann eins, wenn der Vektor `E` mindestens ein Element enthält, welches nicht gleich null ist. Und `all(E)` ist dann eins, wenn alle Elemente von `E` ungleich null sind. Bei Matrizen müssen wir `any` und `all` zweimal anwenden, da es sich um Vektorfunktionen handelt, `any(any(A))` gibt beispielsweise an, ob `A` mindestens ein Element enthält, welches ungleich null ist.

Eine `if`-Bedingung hat in `Matlab` folgende allgemeine Form:

`if` *Bedingung*
  Befehle
`elseif` *Bedingung*
  Befehle
`else`
  Befehle
`end`

Bei der Bedingung `if A==B` werden die Befehle innerhalb des `if`-Blocks nur ausgeführt, wenn alle Elemente von `A` und `B` identisch sind, also `all(all(A==B))` gleich eins ist.

- Wenn wir einen Befehl ausführen möchten, falls `A` und `B` nicht identisch sind, müssen wir `if any(any(A~=B))` schreiben, `if A~=B` funktioniert nicht. Weshalb?
- Finde einen Ausdruck, der genau dann eins ist, wenn alle Elemente von `A` kleiner als das kleinste Element von `B` sind.
- Finde einen Ausdruck, der genau dann eins ist, wenn mindestens ein Element in `A` mindestens 20% grösser ist als das Element an derselben Stelle in `B`.

4.2 While- und for-Schleifen

Schleifen haben in `Matlab` eine ähnliche Struktur wie in anderen bekannten Programmiersprachen. Wir geben je ein Beispiel für eine `while`- und eine `for`-Schleife:

```
f = [1];
g = 1;
while g < 1000
  f = [f; g];
  g = f(length(f)-1) + g;
end
f
```

```
s = 0;
n = 100;
for k = [1:n]
  s = s+k;
end
s
```

Matlab ist für das Rechnen mit Matrizen optimiert. Matrixoperationen werden daher deutlich schneller ausgeführt als Schleifen-Konstruktionen. Bei grösseren Berechnungen - wie sie gegen Ende dieses Praktikums nötig werden - kann eine "gute" Implementation (ohne Schleifen) weniger als eine Sekunde in Anspruch nehmen, während eine "schlechte" (mit vielen Schleifen) nach Minuten noch am Rechnen ist. Achte also darauf, dass Du Schleifen möglichst vermeidest.

Die Zeit, welche Matlab für eine Berechnung benötigt, kann gemessen werden. Der Befehl `tic` startet eine Stoppuhr und der Befehl `toc` gibt die seit dem Start vergangene Zeit in Sekunden zurück. Damit die Zeit für das Eintippen der Befehle (deren Ausführungszeit man messen möchte) nicht mitgemessen wird, schreibt man `tic;` *Befehle*`; toc` auf eine Zeile und drückt erst dann Enter.

- Was berechnet die `while`-Schleife in der Variable `f`?
- Was ist das Ergebnis `s` der `for`-Schleife? Wie hätten wir dieselbe Berechnung ohne Schleife mit einem einzigen Befehl machen können?
- Erhöhe `n` auf 1000000 und messe mit `tic` und `toc`, welche Lösung schneller ist, die mit der Schleife oder die ohne (die Schleifen-Lösung kannst Du so auf einer einzigen Zeile eintippen: `tic; s=0; for k=[1:1000000] s=s+k; end; toc`).

5 Graphische Darstellung von Matrizen

5.1 Plots

Matlab kann Ergebnisse auch graphisch in sogenannten Plots darstellen. Als Beispiel möchten wir uns $\sin(x)$ auf dem Intervall $[-\pi, 2\pi]$ anzeigen lassen. In anderen Mathematikprogrammen und auch auf einigen graphischen Taschenrechnern brauchst Du bloss `sin(x)` einzutippen und schon wird gezeichnet. In Matlab geht das so nicht; Matlab kann sich unter einem Sinus nichts vorstellen, es weiss nur, wie es den Sinus einer gegebenen Zahl berechnen kann. Wir müssen daher die Sinuswerte einiger x-Werte ausrechnen und uns diese dann anzeigen lassen. Die x-Werte definieren wir als Vektor, z.B. `x = [-pi:0.5:2*pi];` (π ist in Matlab in der Konstanten `pi` gespeichert). Im nächsten Schritt berechnen wir für jeden dieser x-Werte seinen Sinuswert: `y = sin(x);` (wir erinnern uns: `sin` ist eine Skalarfunktion und wird daher elementweise ausgewertet).

Mit dem Befehl `plot(x,y)` lassen wir Matlab die y-Werte gegenüber den x-Werten in ein Koordinatensystem eintragen und die einzelnen Punkte verbinden, probier's aus. Du wirst

feststellen, dass unser Sinus ziemlich eckig aussieht. Das liegt daran, dass wir nur wenige Werte (wie viele sind's?) ausgerechnet haben. Wir definieren nun einen neuen x-Vektor `x = [-pi:0.01:2*pi];` (wie viele Werte sind's jetzt?), berechnen erneut die Sinus-Werte (`y = sin(x);`) und lassen `Matlab` wieder zeichnen. Der ursprüngliche Plot wird gelöscht und der neue gezeichnet. Möchtest Du mehrere Kurven in dasselbe Koordinatensystem zeichnen lassen, so kannst Du nach dem ersten `plot`-Befehl die Anweisung `hold on` eingeben, alle folgenden Plots werden dann ins gleiche Bild gezeichnet. Mit `hold off` wird diese Funktion wieder deaktiviert, der nächste Plot überschreibt dann alle vorherigen.

Du kannst Plots auch in mehrere verschiedene Fenster zeichnen lassen. `figure(n)` öffnet ein neues Graphik-Fenster mit der Nummer `n`. Alle folgenden `plot`-Befehle werden dann in diesem Fenster ausgeführt, bis Du mit `figure(m)` auf ein anderes Fenster wechselst.
Da dies bei allzuvielen Plots sehr unübersichtlich werden kann empfiehlt es sich den Befehl `subplot()` zu verwenden. Subplot unterteilt das Fenster in mehrere Graphen, die somit schön miteinander verglichen werden können.

- Der Funktion `plot` kann man mit einem dritten Argument u.a. vorschreiben, in welcher Farbe ein Graph gezeichnet werden soll. Benutze die Hilfe, um herauszufinden, wie die Syntax dafür aussieht und was man sonst noch bestimmen kann. Was macht `plot`, wenn man nur einen einzelnen Vektor übergibt?
- Stelle die Funktionen $\sin(x)$, $\sin(2x)$ und $\sin(3x)$ auf dem Intervall $[0, 2\pi]$ in derselben Graphik in drei verschiedenen Farben dar.
- Versuche nun die vorher verwendeten Funktionen $\sin(x)$, $\sin(2x)$ und $\sin(3x)$ auf dem Intervall $[0, 2\pi]$ in derselben Graphik untereinander mithilfe des Befehls `subplot` darzustellen.
- Die Funktionen `semilogx`, `semilogy`, `loglog` und `stem` zeichnen ebenfalls Graphen. Wodurch unterscheiden sie sich von `plot`?
- Aus der Analysis kennst Du die Beziehung $\sin^2(\alpha) + \cos^2(\alpha) = 1$ ("trigonometrischer Pythagoras"). Wir rechnen die Summe nun mit $\alpha = 4$ in `Matlab` aus und testen, ob das Ergebnis gleich eins ist: `sin(4)^2+cos(4)^2 == 1`. Dann wiederholen wir dasselbe mit $\alpha = 5$. Wie erklärst Du Dir die Ergebnisse? (Fakultativ: Lass Dir einen Plot über ein geeignetes Intervall zeichnen, der anzeigt, für welche α $\sin^2(\alpha) + \cos^2(\alpha)$ gleich eins ist.)

5.2 Histogramme

Histogramme sind eine weitere Möglichkeit, Vektoren graphisch darzustellen. Der Befehl `hist(V)` unterteilt das Intervall zwischen dem kleinsten und dem grössten Element des Vektors `V` in zehn gleich grosse Bereiche und zeigt mit Balken an, wie viele Elemente in welchem Bereich liegen. Probier's aus. Durch einen zweiten Parameter kann die Anzahl der Bereiche verändert werden.

- Erzeuge einen Zeilenvektor `W` mit 1000 Zufallszahlen zwischen 0 und 100. Wie wird das Histogramm von `sqrt(W)` aussehen? Probier's aus.

- Lass Dir `hist(rand(1,10000), 100)` und `hist(randn(1,10000), 100)` je in einem Fenster anzeigen (auch hier kannst Du `figure` verwenden). Was fällt auf? Benutze die Hilfe, um herauszufinden, wodurch sich die Funktionen `rand` und `randn` unterscheiden.

6 M-Files

6.1 Script-Files

Wir können ein Stück `Matlab`-Code in einer Textdatei speichern. Die Datei muss die Endung ".m" besitzen und wird daher auch "M-File" genannt. Tippen wir anschliessend den Namen der Datei (ohne die Endung ".m") in die Kommandozeile, so führt `Matlab` den darin gespeicherten Code aus - genau so, wie wenn wir ihn Zeile für Zeile direkt in die Kommandozeile geschrieben hätten. Diese sogenannten Script-Files erlauben uns also, eine Reihe von Befehlen unter einem Namen zusammenzufassen, über den wir sie dann ausführen können.

Erstelle auf dem Desktop einen neuen Ordner, in welchem Du Deine M-Files abspeichern kannst. Wird in `Matlab` ein Befehl eingegeben, so durchsucht es eine Liste von Verzeichnissen, um das entsprechende M-File zu finden. Zu dieser Liste müssen wir Deinen Ordner nun hinzufügen, damit `Matlab` Deine Programme auch findet. Wähle dazu "Set Path..." im Menü "File".

- Schreibe ein Script-File mit dem Namen `plot_3sinus.m`, welches die zweite Aufgabe aus dem Abschnitt 5.1 löst. Definiere darin einen Vektor `x` sowie die Vektoren `y1`, `y2` und `y3` mit den Werten der drei verschiedenen Sinus-Funktionen. Beim Aufruf von `plot_3sinus` sollte direkt der entsprechende Plot erscheinen.

In Script-Files verwendete Variablen sind global, d.h. sie sind auch nach dem Ausführen des Scripts noch gespeichert und falls vorher schon Variablen gleichen Namens existierten, werden diese überschrieben. Mit dem Befehl `who` kannst Du Dir alle momentan gespeicherten Variablen auflisten lassen. Du wirst feststellen, dass `x`, `y1`, `y2` und `y3` auch darunter sind. Mit `whos` kannst Du Dir übrigens weitere Informationen zu den Variablen anzeigen lassen und mit `clear` *Variablenname* kannst Du eine Variable löschen (Achtung: `clear` alleine löscht alle Variablen!).

M-Files können auch andere M-Files aufrufen (auch sich selbst rekursiv).

6.2 Function-Files

Function-Files sind der zweite Typ von M-Files. Sie unterscheiden sich dadurch von Script-Files, dass ihnen Parameter übergeben werden können und dass Variablen in Function-Files lokal sind, also nach dem Ausführen der Funktion nicht mehr gespeichert sind. Sehen wir uns ein Beispiel einer solchen Funktion an:

```
function r = randint(m,n,a,b)
%RANDINT Randomly generated integral matrix.
%    randint(m,n) returns an m-by-n matrix with
%    entries between 0 and 9.
%    randint(m,n,a,b) returns an m-by-n matrix
%    with entries between integers a and b.
if nargin < 3, a = 0; b = 9; end
r = floor((b-a+1)*rand(m,n)) + a;
```

Function-Files beginnen mit dem Wort `function`, gefolgt von der Rückgabevariable (oder einer Matrix von Rückgabewerten), einem Gleichheitszeichen, dem Namen der Funktion und den Übergabeparametern in Klammern. Beachte, dass der Name des M-Files stimmen muss, damit Du in `Matlab` mit der Funktion arbeiten kannst; der Funktionsname, der nach dem Gleichheitszeichen steht, ist eigentlich irrelevant. Nach der ersten Zeile folgen einige Kommentarzeilen (diese beginnen in `Matlab` mit `%` und veranlassen es, den Rest der Zeile zu ignorieren) und dann der eigentliche Code der Funktion. Der erste zusammenhängende Block von Kommentarzeilen wird angezeigt, wenn Du `help randint` eingibst, Du kannst also eigene Funktionen mit Hilfstexten versehen, wie sie die bereits vorhandenen `Matlab`-Funktionen auch haben. Beachte, dass in `Matlab`-Funktionen kein `return`-Befehl (wie beispielsweise in C) nötig ist. Zurückgegeben wird einfach das, was am Schluss in der Rückgabevariable (die in der ersten Zeile definiert wurde) steht. `Matlab` kennt zwar auch einen `return`-Befehl, doch dieser kommt nur zum Einsatz, wenn man eine Funktion vorzeitig verlassen und zum aufrufenden Programm zurückkehren möchte. Speichere obigen Code in einer Datei `randint.m` in Deinem Ordner und probiere die Funktion aus.

Selbst geschriebene Funktionen sind gleichwertig mit den in `Matlab` bereits integrierten Funktionen. Den Code von bereits vorhandenen Funktionen kannst Du Dir mit dem Befehl `type` anzeigen lassen, ausser es handelt sich um sogenannt eingebaute Funktionen, die in C geschrieben sind. Probiere `type sin`, `type ceil`, `type vander`, `type hist`.

- Wozu dient die zweitletzte Zeile in der Funktion `randint`?
- Schreibe eine Funktion `ntiefk(n,k)`, welche Binomialkoeffizienten[1] berechnet.

[1] Falls Du Dich nicht an die Formel erinnerst:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Wie kannst Du in `Matlab` Fakultäten berechnen? Überleg es Dir oder spicke in der bereits vorhandenen Funktion `factorial` mithilfe des Befehls `type` (in dieser Funktion findest Du erst einige Vergleiche. Was testen sie?). In `Matlab` gibt es übrigens bereits eine Funktion, welche die Binomialkoeffizienten berechnet, wie heisst sie?

- Schreibe eine Funktion `binomdist(n,p)`, welche die Dichtefunktion der Binomialverteilung[2] berechnet. Wird sie ohne Zuweisung aufgerufen, so soll sie die Funktionswerte mit `stem` graphisch darstellen. Wird das Resultat einer Variable zugewiesen, so soll diese Variable nachher ein Zeilenvektor sein, der die Werte von $f(x)$ für $0 \leq x \leq n$ enthält. `hist` zeigt ebenfalls einen Plot, falls keine Zuweisung gemacht wird, spicke nötigenfalls dort, um rauszufinden, wie das gemacht wird. Was ist (unabhänging von `n` und `p`) `sum(binomdist(n,p))`?

7 mat-Files

Die Matrizen und Vektoren werden normalerweise nur in den Workspace abgespeichert, wo sie selbstverständlich nur für das aufrufende Programm, zu dem der Workspace gehört zugänglich sind. Will man aber 'globale' Variablen (z.b. die Abtastfrequenz oder die Wortlänge), also eine Variable, auf die mehrere verschiedene Programme zugreiffen können muss man diese als Datei Abspeichern. Dies macht man mit den mat-Files.
Führe folgende Zeilen Schritt für Schritt durch und vollziehe nach was wo im speicher steht und für wen zugreiffbar ist.

```
A=rand(1);
B=rand(1)+1;
save('data.mat','A','B');
clear all;
A
A=42;
load('data.mat');
A
```

[2] Auch hier die Formel zur Erinnerung:

$$f(x) = P(X = x) = \binom{n}{x} \cdot p^x \cdot (1-p)^{n-x}$$

n und x sind nichtnegative ganze Zahlen und für p gilt $0 \leq p \leq 1$. Die Binomialverteilung gibt an, wie gross die Wahrscheinlichkeit ist, dass von n Zufallsexperimenten, von denen jedes einzelne unabhängig von den andern mit Wahrscheinlichkeit p erfolgreich ist, X gelingen. Für die Berechnung kannst Du Deine eigene Funktion `ntiefk` verwenden oder die bereits in `Matlab` integrierte. Oder aber Du machst es ganz anders...