Matlab Einführung
für Signale und Systeme 2

Karl Rupp

22. Juni 2005

Einleitung

Matlab ist ein dialogorientiertes Programmsystem für numerisches Rechnen und zur Visualisierung der Ergebnisse. Entgegen Formelmanipulationsprogrammen wie Maple oder Mathematica arbeitet Matlab nicht mit (mathematischen) Funktionen im herkömmlichen Sinn, sondern verwendet Vektoren und Matrizen. Dies reicht gerade für numerische Lösungen vollkommen aus und ist vor allem sehr schnell. Andererseits darf man aber nicht vergessen, dass gerade bei z.B. unendlichen Summen auch numerische Fehler entstehen, da man nur auf endlich große Vektoren zurückgreifen kann.

Wünsche, Anregungen, Beschwerden und natürlich auch Fehlermeldungen[1] werden unter

e0325941@student.tuwien.ac.at

erwartet.

1 Befehle in Matlab

Nach dem Start erscheint Matlab für gewöhnlich mit drei Fenstern, nämlich dem Befehlsfenster („Command Window“), einer Übersicht über angelegte Objekte („Workspace“) und einer Liste der zuletzt eingegebenen Befehle („Command History“). Befehle werden für gewöhnlich in Scripts abgelegt, welche dann im Befehlsfenster aufgerufen werden. Alternativ ist es natürlich

[1] Damit sind keine Windows-Fehlermeldungen gemeint, sondern das Melden von Fehlern, die sich in diese Einführung eingeschlichen haben ;-)

auch möglich, die Befehle direkt im Befehlsfenster einzugeben. Hier hat man gerade am Beginn den Vorteil, bei jedem Befehl entsprechende Rückmeldungen vom System zu erhalten.

1.1 Anlegen und Ausführen von Scripts

Scripts werden entweder über den Befehl `edit` oder direkt über die GUI (File->New->M-File) angelegt. Das Script `demoscript.m` wird also beispielsweise über

```
>> edit demoscript.m
```

angelegt. Darin können dann alle Befehle Zeile für Zeile eingegeben werden. Danach wird das Script abgespeichert und dann im Befehlsfenster durch

```
>> demoscript
```

gestartet. Wir wollen uns hier aber auf die direkte Befehlseingabe im Befehlsfenster beschränken.

1.2 Ausgabe unterdrücken

Befehle können, müssen aber nicht mit einem Semikolon abgeschlossen werden. Wird ein Befehl mit einem Semikolon abgeschlossen, gibt Matlab keine Meldungen zurück. Dies wird besonders bei großen Matrizen von Bedeutung. Mit Matlab-Ausgabe:

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

Im Gegensatz dazu ohne Ausgabe:

```
>> v = [1 2 3];
>>
```

2 Arbeiten mit Vektoren und Matrizen

Im folgenden werden zu den jeweiligen Matlab-Befehlen eventuelle Ausgaben meist nicht angeführt, da die Befehle entweder selbsterklärend sind oder danach im Text separat erklärt werden.

2.1 Anlegen der Objekte

Die einfachste Art, einen Vektor anzulegen, ist

```
>> v1 = [1 2 0 5 1 3]
>> v2 = [1, 2, 0, 5, 1, 3]
```

Die beiden Befehle sind gleichwertig und erzeugen jeweils einen **Zeilenvektor**. Möchten wir uns die gerade angelegten Objekte noch einmal ansehen, so geht das ganz einfach über den jeweiligen Objektnamen, also hier beispielsweise über

```
>> v1
 v1 =
      1 2 0 5 1 3
```

Spaltenvektoren werden durch ein Semikolon nach jedem Eintrag erzeugt:

```
>> v3 = [1; 2; 0; 5; 1; 3]
```

Die Verknüpfung von Zeilen- und Spaltenvektoren liefert eine **Matrix**:

```
>> v4 = [1 2; 3 4]
```

erzeugt die Matrix

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Nun ist es aber nicht immer effektiv, immer alle Einträge per Hand einzugeben. Aus diesem Grund stellt Matlab ein paar Möglichkeiten zur Verfügung, die die Eingabe erleichtern. Die wichtigsten möchte ich hier behandeln:

```
>> m1 = ones(3,4)
>> m2 = zeros(3,4)
>> m3 = eye(3)
>> m4 = diag([1 3 5])
```

Diese Befehle erzeugen der Reihe nach die Matrizen

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{pmatrix}$$

Wir können vor allem `ones()` und `zeros()` auch dazu verwenden, um uns entsprechende Vektoren mit Einsern oder Nullen zu erzeugen.

Sehr häufig werden Vektoren mit den natürlichen Zahlen in geordneter Reihenfolge benötigt, um zum Beispiel daraus Eingangsfunktionen zu bilden: Das Befehlsformat lautet [*untere Schranke:Schrittweite*:*obere Schranke*], wobei die Schrittweite ein optionales Argument ist.

```
>> s1 = [0:20]
>> s2 = [0:0.5:10]
```

Der erste Befehl erzeugt den Vektor (0 1 2 ... 19 20), der zweite den Vektor (0 0.5 1 ... 9 9.5 10).

Abschließend sei auch noch die Transposition erwähnt, die über einen Apostroph implementiert ist:

```
>> m1 = m'
```

m1 ist dann die zu m transponierte Matrix. So lassen sich auch Zeilenvektoren in Spaltenvektoren überführen und umgekehrt.

2.2 Rechnen mit Matrizen und Vektoren

Es ist ganz wichtig, dass wir uns jederzeit im Klaren sind, ob wir jetzt Matrizen multiplizieren oder ob wir nur auf den Einträgen der Matrix operieren möchten. Im Normalfall ist es nämlich ein großer Unterschied, ob wir eine ganze Matrix (im mathematischen Sinn) quadrieren oder ob wir nur deren Einträge quadrieren. Da Matlab beide Möglichkeiten zur Verfügung stellt, müssen wir auch die entsprechenden Operatoren auseinander halten. Glücklicherweise gibt es bei den Operatoren + und − keinen Unterschied zwischen einer Operation auf der ganzen Matrix und einer elementweisen Operation.

Lediglich beim Produkt, bei der Division und beim Exponenten ist Vorsicht geboten. Möchten wir *elementweise* operieren, so verwenden wir die Operatoren `.*` , `./` bzw. `.^` Mittels `help ops` kann eine kurze Übersicht über die verfügbaren Operatoren angezeigt werden.

Ein Beispiel zur Verdeutlichung:

```
>> [1 2; 3 4] * [5 6; 7 8]
>> [1 2; 3 4] .* [5 6; 7 8]
```

Diese beiden Befehle erzeugen der Reihe nach die Ausgaben

$$\begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

$$\begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix}$$

Im ersten Fall wird eine Matrizenmultiplikation durchgeführt (herkömmliche Multiplikation), im zweiten Fall wird lediglich elementweise multipliziert (d.h. Einträge an der gleichen Position werden multipliziert). Entsprechend verhält es sich mit der Division und dem Exponenten.

2.3 Zusammensetzen von Objekten

Haben wir zwei Vektoren und möchten diese zu einer Matrix oder einem (größeren) Vektor zusammensetzen, so stellt das unter Matlab kein Problem dar:

```
>> v1 = [1 2 3]
```

$$v1 = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

```
>> v2 = [4 5 6]
```

$$v2 = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$$

```
>> m1 = [v1; v2]
```

$$m1 = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
>> v3 = [v1 v2]
```

$$v3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

```
>> v4 = [v1'; v2']
```

$$v4 = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix}$$

2.4 Zugriff auf Teilbereiche von Objekten

Möchten wir auf einzelne Elemente einer Matrix zugreifen, so geschieht dies über Angabe der Zeilen- und Spaltenindizes in nachfolgender Klammer. Dabei wird bei Eins zu zählen begonnen und nicht (wie bei vielen Programmiersprachen üblich) bei Null.

```
>> m = [1 2 3; 4 5 6]
>> m(2,3)
```

erzeugt die Ausgabe '6'. Bei Vektoren reicht auch die Angabe eines Indizes. Gibt man bei einer Matrix lediglich ein Argument an, so wird zuerst die Elemente spaltenweise nummeriert, der Index 2 entspricht also dem zweiten Element der ersten Spalte und so weiter.

Möchte man bei Matrizen einen Zeilenvektor oder Spaltenvektor extrahieren, so ersetzt man den jeweiligen Index durch einen Doppelpunkt. Bei der eben eingeführten Matrix m würde das so aussehen:

```
>> v1 = m(:,2)
```

$$v1 = \begin{pmatrix} 2 \\ 5 \end{pmatrix}$$

```
>> v2 = m(2,:)
```

$$v2 = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$$

Eine interessante Möglichkeit ist es auch, als Index einen Vektor zu übergeben. So erhält man durch

```
>> v1 = m([1 2])
```

den ersten und zweiten Eintrag der Matrix m. Entsprechend liefert

```
>> m(1:4)
```

die ersten vier Einträge (spaltenweise gezählt, siehe oben) der Matrix m.

2.5 Ausgabe durch plot() und stem()

Haben wir Vektoren und Matrizen angelegt und möchten diese grafisch Visualisieren, so gibt es einerseits den Befehl `plot()` und andererseits den Befehl `stem()`. Ersterer ist für zeitkontinuierliche Prozesse gedacht, zweiterer für zeitdiskrete.

Es gibt viele Formatierungsmöglichkeiten für diese beiden Befehle, diese sind in der Matlab-Hilfe nachzuschlagen. Wir werden hauptsächlich `stem()` verwenden, da wir zeitdiskrete Signale betrachten.

`stem()` nimmt die Werte aus dem übergebenen Vektor. Wird jedoch eine Matrix übergeben, so wird für jede Spalte der Matrix ein eigener Plot erstellt. Daher wollen wir auch in den folgenden Beispielen mit Spaltenvektoren rechnen, da dann eine (logisch) einfache Zusammensetzung von mehreren Spaltenvektoren zu einer Matrix, die wir anschließend plotten können, gegeben ist. Ein Beispiel:

```
>> stem([1 5; 2 6; 3 7; 4 8; 5 9])
```

zeichnet zwei diskrete Kurven mit Steigung 1. (Kurve 1 von 1 bis 5, Kurve 2 von 5 bis 9)

Weiters ist es auch möglich, die Werte der x-Achse als Vektor mitzuübergeben, wenn keine äquidistanten Zerlegungen vorliegen, der Nullpunkt verschoben ist, etc. Die Syntax lautet dann `stem(X,Y)`, wobei X ein Vektor mit den x-Werten und Y ein Vektor mit den zugehörigen y-Werten ist. Für genauere Informationen möchte ich auf die Matlab-Hilfe verweisen.

3 Learning by Doing

Damit wir nach der bisherigen Spielerei mit Vektoren und Matrizen unser Wissen auch praktisch umsetzen können, möchte ich hier nun drei Beispiele bringen, um zu demonstrieren, wie Lösungen entstehen (sollten/könnten). Bei den jeweiligen Überschriften stehen die zugehörigen Beispielnummern aus der SuS2 Beispielsammlung in Klammer.

3.1 Beispiel 1: Erzeugung von periodischen Signalen (8.4)

Zuerst wollen wir uns einen (Spalten-)Vektor erzeugen, der unserer Zeitachse entspricht:

```
>> t = [0:50]'
```

Da wir aber periodische Signale benötigen, werden wir auch eine Art periodische Zeitachse benötigen. Durch

```
>> t = mod(t, 7)
```

erzeugen wir uns einen periodischen Vektor mit Einträgen von Null bis Sechs.

Nun erzeugen wir ein paar Signale, indem wir auf unseren periodischen Zeit-Vektor arithmetische Operationen ausführen:

```
>> x1 = sin(pi /3 .* t)
>> x2 = 0.8 .^ t
>> x3 = t .^ 0.8
>> x4 = 1.5 .* t - 4 .* t .^ 0.5 + exp(-t)
```

Damit haben wir nun vier periodische Signale x1 bis x4 erzeugt, *wobei die Punkte vor den Operanden wichtig sind.* Um diese gemeinsam plotten zu können, verpacken wir diese noch in eine Matrix. Da x1 bis x4 bereits Spaltenvektoren sind, brauchen wir diese lediglich nebeneinander zu reihen:

```
>> x = [x1 x2 x3 x4]
```

Schließlich lassen wir das ganze noch plotten:

```
>> stem(x)
```

und freuen uns über unser erstes gelöstes Beispiel ;-)

3.2 Beispiel 2: Berechnen des Ausgangs durch Faltung des Eingangs mit der Stoßantwort (8.16)

Wiederum führen wir zuerst unseren Zeitvektor ein:

```
>> t = [-5:50]'
```

Diesmal starten wir von -5, um auch das Verhalten vor dem Nullpunkt zu sehen.

Die Impulsantwort werden wir gleich direkt anschreiben:

```
>> h = [0.25 .^ [0:9]]'
```

Als Eingang verwenden wir die Funktionen aus den Unterpunkten a) und e), wobei wir auch den verschobenen Zeitnullpunkt beachten:

```
>> x1 = [zeros(1, 5) ones(1, 51)]'
>> x2 = [0.25 .^ t]
>> x2 = x2 .* x1
```

Der letzte Befehl ist notwendig, um Nullen vor dem Zeitnullpunkt zu erzeugen. Um nun den Ausgang zu erzeugen, verwenden wir wie empfohlen die Funktion `conv()`. Dabei ist aber zu beachten, dass der resultierende Vektor größer als die beiden Argument-Vektoren ist. In den letzten Einträgen des Ergebnisvektors stehen 'Überreste'[2] des Stoßantwort-Vektors. Diese werden wir aber ganz einfach abschneiden:

```
>> y1 = conv(x1, h)
>> y1 = y1(1:length(t))
>> y2 = conv(x2, h)
>> y2 = y2(1:length(t))
```

Das Abschneiden ist natürlich nicht notwendig, jedoch würden dann bei den Plots unschöne Funktionssprünge am rechten Rand entstehen und zusätzlich müssten wir nachträglich die Zeitachse verlängern.

Schlussendlich wollen wir wieder die Systemantworten plotten lassen:

```
>> stem(t, [y1 y2])
```

Damit wäre auch dieses Beispiel (zumindest zu zwei Fünftel) gelöst.

3.3 Beispiel 3: Auflösen einer Differenzengleichung durch die Funktion filter() (4.3)

Mit Hilfe der $\mathcal{Z}$-Transformation transformiert sich die Übertragungsfunktion eine Differenzengleichung zu einem Quotienten zweier Polynome. Die Funktion `filter()` erwartet als Argumente die Koeffizienten des Zählers, des Nenners (jeweils als Vektoren) und das Eingangssignal.

Die Übertragungsfunktion ist in diesem Beispiel durch

$$H(z) = \frac{1}{1 - \frac{1}{2}z^{-1} + \frac{1}{4}z^{-2}}$$

[2] Am einfachsten ist sich das durch das Übereinanderschieben der Vektoren zu veranschaulichen (grafisches Falten). Am Anfang (meist erwünscht) und am Ende (meist unerwünscht) überdeckt der eine Vektor den anderen nicht vollständig

gegeben. Die Koeffizientenvektoren sind also [1] für den Zähler und
$[1 \quad -0.5 \quad 0.25]$ für den Nenner.

Wir generieren uns wiederum einen Zeitvektor und die Eingangsfunktion:

```
>> t = [-5: 30]'
>> x1 = [zeros(5,1); 0.5 .^ t(6:length(t))]
```

Die Zuweisung zu x1 ist etwas trickreich: Zuerst generieren wir fünf Nullen am Anfang (Verschiebung des Zeitnullpunkts durch t). Die restlichen Einträge entstehen, indem wir aus t alle Einträge ab 0 extrahieren (`t(6:length(t))`) und dann 0.5 hoch diesen Einträgen nehmen. Alternativ hätten wir auch wie im vorigen Beispiel mit dem Einheitssprung-Vektor multiplizieren können.

Um den Ausgang zu erzeugen, verwenden wir nun die Funktion `filter()`:

```
>> y = filter([1], [1 -0.5 0.25], x1)
```

Damit ist die Arbeit auch schon getan und wir lassen uns das Ergebnis präsentieren:

```
>> stem(t,y)
```

4 Ausblick

Natürlich stellen die vorgestellten Funktionen nur einen Bruchteil der Möglichkeiten von Matlab dar. Mit dem nun vorhandenen Grundwerkzeug sollte es aber keine großen Schwierigkeiten machen, auch komplexere Aufgaben zu bewältigen.

Viele hilfreiche Funktionen sind in der Signal Processing Toolbox zu finden. In der Hilfe sind dazu noch reichlich Informationen verfügbar.

Schlussendlich bleibt mir noch, viel Erfolg beim Bearbeiten und Lösen der Beispiele zu wünschen.