MATLAB 8 i korthet

Carina Edlund

31 augusti 2015
Version 1

Synpunkter

Har du synpunkter eller förslag till förbättringar på detta häfte skicka gärna ett ebrev till

golda@kth.se och ange matlabhäfte i ämnesraden.

Tack

Jag vill rikta ett stort och varmt tack till

Åsa Hansson, Erik Dalsryd, Lennart Edsberg, Ninni Carlsund, Katarina Gustavsson och Yngve Sundblad

för att ni bidragit med många värdefulla synpunkter och tips till detta häfte.

Carina Edlund

Innehåll

Kapitel 1

Introduktion

Denna text ger en inblick i hur du kan använda Matlab som ett redskap när du ska utföra matematiska och numeriska beräkningar samt programmera. Texten och tillhörande exempel arbetar uteslutande med kommandon och satser.

Det bör nämnas att Matlab ger möjlighet till fler sätt att arbeta t.ex. via menyer, GUI och appar. Dessa förklaras inte i denna text.

1.1 Introduktion

1.1.1 Komma igång med Matlab

För att starta Matlab på en Mac eller PC kan man dubbelklicka på dess ikon och för Unix kan man skriva *matlab* i ett terminalfönster. Ett Matlabfönster med olika delfönster öppnas. De olika delfönstren som finns är bland annat kommandofönstret (command window), nuvarande katalog (current folder) och arbetsarean (workspace). Man kan lägga till och ta bort delfönster via Matlabfönstrets menyrad.

I kommandofönstret kan man skriva olika kommandon på kommandoraden som skrivs

```
>>
```

och sedan trycka *retur*-knappen för att de ska utföras. Om man vill beräkna $3 + 4$ skriver man det på kommandoraden

```
>> 3+4
7
```

Matlab svarar med att beräkna summan och skriva ut resultatet därefter är Matlab redo att ta emot nya kommandon.

Alla kommandon skrivs med små bokstäver.

Istället för att skriva om tidigare kommandon kan man använda piltangenterna ↑ och ↓ för att bläddra mellan dem eller använda sig av fönstret med kommandohistorian lagrad (command history)

När man skriver en följd av kommandon som man vill hålla ihop eller spara till ett annat tillfälle är det lätt att spara dem i ett script eller en egendefinierad funktionsfil. Via Matlabfönstrets menyrad kan man öppna upp en ny eller befintlig fil och skapa ett script eller en funktionsfil i Matlabs egen editor. Mer detaljer om script och funktionsfiler finns att läsa i avsnitt 1.3.3 respektive 3.3.1.

I arbetsarean (workspace) läggs t.ex. alla variabler som man definierar i ett script eller via kommandoraden. Arbetsarean ger en snabb överblick över vilka variabler som finns och hur de ser ut. För att ta bort innehåll i arbetsarean används kommandot *clear* se avsnitt 1.2.4.

1.1.2 Demonstrationer

För att se vilka olika demonstrationer som finns tillgängliga och därefter köra det man önskar anger man

```
>> demo
```

1.1.3 Hjälp

För att få hjälp i Matlab finns bland annat kommandona *help* och $lookfor$.

help kommando ger en beskrivning av kommandot *kommando*.

$lookfor$ *sökord* listar de kommandon, funktioner som innehåller sökordet *sökord*.

Dessa kommandon gäller även för egendefinierade funktionsfiler. Se avsnitt 3.3.1.

1.2 Variabler

För att kunna återanvända sig av värden kan man namnge dessa. Man kan se det som att man ger ett namn ett visst värde. Detta görs med en tilldelningssats som har formen $namn = värde$.

För att ge variabeln *pris* värdet 5 görs följande tilldelning

```
>> pris = 5
pris =
     5
```

Här görs tilldelningen i kommandofönstret. En tilldelning skrivs på exakt samma sätt oberoende av var man gör den.

Observera att Matlab skiljer på stora och små bokstäver vilket medför att *pris* och $Pris$ är två olika variabler.

```
>> Pris = 6.3
Pris =
     6.3000
```

Variablerna *pris* och *Pris* finns nu definierade i Matlabs arbetsarea (Workspace) och kan därmed användas.

För att komma åt en variabels värde är det bara att ange variabelns namn. För att summera de två priserna, variablerna skriver man

```
>> Pris + pris
ans =
    11.3000
```

För att till exempel skapa nya variabler eller inspektera variabler kan man gå via HOME-menyn och välja Ny variabel respektive Öppna variabel.

1.2.1 Variabelnamn

Följande regler gäller för variabelnamn:

- de måste alltid börja med bokstav.
- de får högst bestå av 63 tecken.
- det är inte tillåtet att använda mellanslag eller interpunktionstecken dvs punkt, kommatecken o.s.v.
- det är tillåtet att använda understrykningstecken, _.

Observera att Matlab är case sensitive dvs skiljer på stora och små bokstäver vilket medför att *pris* och *Pris* är två olika variabler samt att de fördefinierade variablerna kan användaren själv definiera om.

1.2.2 Namnkonvention

Skalärer och vektorer namnas med små bokstäver och matriser med stora.

1.2.3 Fördefinierade variabler

I MatLab finns flera fördefinierade variabler. Dessa är bland annat

eps	maskinnoggrannhet
Inf	förkortning av infinity. *inf* finns även.
	Resultat vid t.ex 1/0
NaN	Not A Number, t.ex. då 0/0
i, j	symboler för den imaginära enheten
	dvs roten ur -1
pi	det fördefinierade värdet på *pi*
	dvs 3.14159265358979...
ans	namn för resultatvärden
realmax	det största ändliga flyttalet $1.7977e+308$

Observera att de fördefinierade variablerna kan användaren själv definiera om.

1.2.4 Ta bort variabler

Variabler kan tas bort från arbetsarean (workspace) med hjälp av kommandot *clear*. Kommandot har många olika tillägg bland annat *clear global* som tar bort globala variabler, *clear* $variabel_1$ $variabel_2$ ... som tar bort de listade variablerna och *clear all* som bland annat tar bort variabler, globala variabler och funktionslänkar.

1.2.5 Variabeltyper

Observera att detta avsnitt kräver kunskaper om *script* och *funktioner* som förklaras i avsnitt 1.3.3 respektive 3.3.

Matlab har tre olika typer av variabler. Variablerna kan vara lokala, globala eller persistenta. Här förklaras endast de lokala och globala variablerna. Detta görs utifrån följande exempel som ritar ut en cirkel.

I funktionsfilen *cirkelKoordinater.m* beräknas x respektive y koordinaterna för cirkeln. I scriptet kallat *cirkelRitare.m* finns uppgifter som specificerar enhetscirkeln. Från scriptet anropas funktionen *cirkelKoordinater* och x respektive y koordinaterna returneras till variablerna $xKoord$ respektive $yKoord$. Slutligen plottas cirkeln ut.

Filen *cirkelKoordinater.m*

```
function [x, y] = cirkelKoordinater(fi)
% Cirkelkoordinater
% Beräknar x och y koordinaterna
% för en cirkel.
% Anropas av funktionen cirkelRitare.m

% Variabeln R görs tillgänglig
global R

% x resp y koordinater beräknas
x = R*cos(fi);
y = R*sin(fi);
% funktionen slut
```

Filen *cirkelRitare.m*

```
% Ritar ut enhetscirkeln.
% Anropar funktionen cirkelKoordinater.m
```

```
% Töm arbetsarean; tar bl.a. bort alla
% tidigare variabler.
clear all

% Stänger alla tidigare grafikfönster
close all

% Variabeln R görs tillgänglig
global R

% Tilldelar värden
R = 1;
fi = 0:0.01:2*pi;

% Koordinaterna beräknas och returneras
[xKoord, yKoord] = cirkelKoordinater(fi);

% Plottar cirkelns koordinater
plot(xKoord,yKoord)

% Sätter axlarna så att grafen
% blir kvadratisk och cirkeln rund.
axis equal
% filen cirkelRitare.m slut
```

Anropet för att rita ut cirkeln är

```
>> cirkelRitare
```

Lokala variabler

De variabler som förekommer i funktioner är lokala om de inte har deklarerats som globala eller persistenta. Med lokal variabel menas att variabeln endast är åtkomlig i den funktion där den förekommer. De lokala variablerna existerar så länge som funktionen exekveras/körs.

I funktionen *cirkelKoordinater.m* är variablerna fi, x samt y lokala. Innehållet i dessa variabler kan alltså INTE nås utanför funktionen.

```
>> x
??? Undefined function or variable 'x'.
```

men gör man däremot

```
>> fi
ans =
    0  0.0100  0.0200  ... 6.2800
```

kommer Matlab att skriva ut alla elementen i fi-vektorn (vilket är ganska många och därför utelämnas flertalet). Det som händer är att variabeln fi i scriptet *cirkelRitare.m* definieras i Matlabs arbetsarea (Workspace) när scriptet anropas. Lägg märke till att de två variablerna fi i scriptet och fi i funktionen är olika men råkar ha samma namn och i detta fall likadant innehåll.

Globala variabler

I vissa situationer kan det vara bra att använda sig av globala variabler men eftersom det lätt kan leda till oönskade bieffekter bör man försöka undvika det. Globala variabler är variabler som görs åtkomliga för de funktioner man önskar och eventuellt också Matlabs arbetsarea (Workspace). På detta vis slipper man skicka variabelns värde som indata till funktionen. En variabel eller som här flera variabler görs globala genom att skriva
$global \quad variabel_1 \quad variabel_2 \quad ... \quad variabel_n$
i varje funktion och script där man vill komma åt dess värde.

I filerna *cirkelKoordinater.m* och *cirkelRitare.m* är radien, R deklarerad som en global variabel. Det innebär att R är en och samma variabel i de båda filerna.

Om man i kommandofönstret skriver

```
>> R
R =
     1
```

så skrivs värdet på den globala variabeln R ut.

1.3 Arbeta med Matlab

1.3.1 Kommentarer

Kommentarer skrivs med tecknet %. Den resterande texten på raden ignoreras av Matlab. Observera att kommentaren gäller högst en rad.

```
>> pris = 5; % tilldelar pris värdet 5
```

Tips! Om man vill kommentera bort ett större stycke, markera texten och använd *Comment* i editorns menyrad.

1.3.2 Undertrycka utskrift

För att förhindra att Matlab skriver ut t.ex en variabels värde eller ett funktionsresultat skriver man ett semikolon efter kommandot, satsen.

```
>> pris = 5
pris =
     5

>> pris;
>> pris
pris =
     5

>> cos(pi/3)
ans =
    0.5000

>> cos(pi/3);
```

1.3.3 Script

Ett script är en fil som innehåller en följd med kommandon, satser. Filnamnet är på formen $filnamn.m$. För att scriptet ska köras dvs alla kommandon exekveras i tur och ordning anropas den med $filnamn$. Ett script kan köras via *Run* i Matlabeditorns menyrad. Det kan också anropas från kommandoraden, ett script eller en funktionsfil.

I scriptet $kvadrat4.m$ nedan tilldelas variabeln x värdet 4 som därefter kvadreras och resultatet skrivs ut.

```
% Script: kvadrat4.m
x = 4;
disp(x^2)
% slut kvadrat4.m
```

Scriptet $kvadrat4.m$ anropas från kommandoraden och resultatet 16 skrivs ut

```
>> kvadrat4
   16
```

Variabler som definieras i ett script lagras i Matlabs arbetsarea (workspace) och blir därmed åtkomliga från alla script och även kommandoraden

```
>> x
x =
     4
```

Det kan vara en god vana att ta bort variabler i början av ett script för att undvika onödigt trassel. Kommandon som t.ex. *clear*, *clear all* och *clear variabelnamn* kan användas för att ta bort variabler, för mer detaljer angående *clear* se avsnitt 1.2.4.

Ett ytterligare exempel på script finns i avsnitt 1.2.5.

1.3.4 Format

Det finns olika utskriftsformat i Matlab. De är bland annat

format	går tillbaka till default
format short	ger 4 siffror (default)
format long	ger 15 siffror
format short e	flyttal med 4 decimaler
format long e	flyttal med 15 decimaler
format compact	tar bort tomma rader

Utskriftsformatet påverkar endast hur talen skrivs ut INTE hur de representeras i Matlab. Alla beräkningar utförs alltid med datorprecision dvs ca 16 siffor.

```
>> format short
>> 10*pi

ans =

   31.4159

>> format compact
>> format long
>> 10*pi
ans =
  31.41592653589793
>> format short e
>> 10*pi
ans =
   3.1416e+01
```

1.3.5 Mätning av tid

För att få ett mått på hur bra en viss beräkning är kan man bland annat mäta hur lång tid det tar att utföra beräkningen. Det kan utföras med *tic/toc*, *cputime* eller via *Run and Time* i editorns menyrad som använder sig av *cputime*.

Tänk på att i vissa fall kan en beräkning ta försumbart liten tid. I dessa fall bör beräkningen upprepas ett antal gånger och därefter kan medelvärdet ge en uppskattning av tidsåtgången. I avsnittet *tic/toc* visas ett exempel.

cputime

Kommandot *cputime* returnerar den tid som gått sedan Matlab startades och kan därför användas för att uppskatta tidsåtgången för en beräkning.

Genom att skriva

```
t = cputime;
% beräkning av sats(er)
t = cputime - t;
```

tilldelas variabeln t i rad 3 den tid det tar att beräkna satsen/satserna.

tic/toc

tic startar ett stoppur och *toc* mäter tiden från det att stoppuret startats.

```
antal = 50;
tid = tic;
for i = 1:antal
   % beräkning av sats(er)
end
tid = toc(tid);
tmedel = tid/antal;
```

Ytterligare förslag att mäta tiden finns i Matlabs help-dokumentation.

Kapitel 2

Matematik och numerik

2.1 Vektorer

För att skapa en följd av värden t.ex. för att avbilda, skapa en tabell används vektorer. Tabeller förklaras längre fram i avsnittet. Värdena i en vektor kallas för en vektors element eller komponenter. Till varje element finns ett index som avspeglar var elementet finns i vektorn. Observera att index alltid är ett positivt heltal och startar med 1.

Ur tabellen nedan kan man skapa två vektorer, vektorn x som innehåller de fem x-värdena och vektorn y som innehåller de fem y-värdena.

index	1	2	3	4	5
x	-2	-1	0	1	2
y	-4	-2	0	2	4

Med den matematiska notationen skrivs vektorerna

$x = (-2 \quad -1 \quad 0 \quad 1 \quad 2)$ och
$y = (-4 \quad -2 \quad 0 \quad 2 \quad 4)$.

En vektors längd anger antalet element i vektorn. För båda vektorerna är längden 5.

2.1.1 Kolonnotation

För att enkelt skapa en följd av värden kan kolonnotation användas. Om man t.ex. vill ha en följd av alla jämna heltal från och med 0 upp till och med 10 kan man skriva

```
>> 0:2:10
ans =
     0     2     4     6     8    10
```

Kolonnotation skrivs på formen
startvärde : *steglängd* : *slutvärde* och returnerar alltid en radvektor.

Om man vill vända på ovanstående följd skriver man istället

```
>> 10:-2:0
ans =
    10     8     6     4     2     0
```

Om steglängden är 1 kan den utelämnas. För att skriva ut alla heltal från och med 0 till och med 5 skriver man

```
>> 0:5
ans =
     0     1     2     3     4     5
```

Varken *startvärde*, *slutvärde* eller *steglängd* behöver vara heltal. Följden 0.02, 0.03, 0.04 kan skrivas

```
>> 0.02:0.01:0.04
ans =
    0.0200    0.0300    0.0400
```

Man kan använda t.ex. aritmetiska uttryck i kolonnotationen

```
>> start = 1;
>> slut = 2;
>> antalPunkter = 5;
>> steg = (slut-start)/(antalPunkter-1);
>> start:steg:slut
ans =
  1.0000  1.2500  1.5000  1.7500  2.0000
```

Obs! Man behöver inte mellanlagra resultat i variabeln *steg*. Här ovan har det gjorts av utrymmesmässiga skäl.

2.1.2 Skapa vektorer

En vektor skapas genom att ange de värden, element som ska ingå. Det finns ett flertal sätt att göra detta, bland annat:

- att ange elementen inom hårdklamrar. Elementen separeras antingen av mellanslag eller kommatecken och ger då en radvektor eller av semikolon eller radbrott, detta ger en kolumnvektor.

- att använda kolonnotation se avsnitt 2.1.1.

- att använda *linspace*.
  linspace(*start*, *slut*, *antal*) skapar en vektor där elementen är ekvidistant fördelade mellan *start*- och *slut*värde. *antal* anger antalet element och om det utelämnas returneras 100 element.
  logspace fungerar på liknande vis men logaritmerar.

- som resultat av beräkning.

För att skapa vektorerna
$x = (-2 \quad -1 \quad 0 \quad 1 \quad 2)$ och
$y = (-4 \quad -2 \quad 0 \quad 2 \quad 4)$ skrivs

```
>> % med hjälp av hårdklamrar
>> x = [-2 -1 0 1 2]
x =
    -2    -1     0     1     2
>> y = [-4 -2 0 2 4]
y =
    -4    -2     0     2     4

>> % med hjälp av kolonnotation
>> x = -2:2
x =
    -2    -1     0     1     2
>> y = -4:2:4
y =
    -4    -2     0     2     4

>> % med hjälp av linspace
>> x = linspace(-2,2,5)
x =
    -2    -1     0     1     2
>> y = linspace(-4,4,5)
y =
    -4    -2     0     2     4

>> % som resultat av beräkning
>> y = 2*x
y =
    -4    -2     0     2     4
```

Nedan följer ytterligare exempel

```
>> z = [3+4 pi*2 7/3]
z =
```

```
    7.0000    6.2832    2.3333

>> linspace(0, pi, 3)
ans =
         0    1.5708    3.1416

>> v = [9:-2:5 -5:-2:-9]
v =
     9     7     5    -5    -7    -9

>> w = linspace(10,20,3)
w =
    10    15    20
```

2.1.3 En vektors längd

För att få fram en vektors längd används *length*. Vektorernas längd i avsnittet 2.1.2 är bland annat

```
>> length(x)
ans =
     5

>> length(w)
ans =
     3
```

2.1.4 En vektors orientering

Det finns två typer av vektorer; radvektor och kolumnvektor. Av namnen framgår hur vektorn är orienterad dvs om alla elementen ligger på en rad eller i en kolumn.

2.1.5 Transponat av en vektor

Man kan konvertera, transponera en radvektor till en kolumnvektor och vice versa med hjälp av transponat operatorn '

```
>> vektor = [ 2 5 7 4]
vektor =
     2     5     7     4

>> vektor'
ans =
     2
     5
     7
     4
```

2.1.6 Åtkomst av vektorers element

Man kan komma åt ett enskilt element och även en del av en vektor med hjälp av indexering. Det görs på formen *vektornamn*(*index*).

Observera att man inte kan ange ett index som inte finns.

Vektorerna som skapats i avsnitt 2.1.2 används nedan.

```
>> % skriver ut hela vektorn w
>> w
w =    10    15    20
```

```
>> % Skriver ut det första elementet
>> w(1)
ans =    10

>> % Skriver ut 1:a och 2:a elementet
>> w(1:2)
ans =    10    15

>> % Skriver ut det 2:a och alla
>> % efterföljande element
>> w(2:end)
ans =    15    20

>> % Skriver ut alla element
>> % Observera: som en kolumnvektor
>> w(:)
ans =
    10
    15
    20

>> % ytterligare exempel
>> % Skriver ut 1:a elementet,
>> % därefter vart annat framifrån
>> v(1:2:end)
ans =     9     5    -7

>> % Skriver ut sista elementet,
>> % därefter vart annat bakifrån
>> v(end:-2:1)
ans =    -9    -5     7

>> % Skriver ut 2:a och 4:e elementet,
>> v([2 4])
ans =    7    -5

>> % Försöker skriva ut det 10:e elementet
>> w(10)
???  Index exceeds matrix dimensions.
```

Felet som görs här är att man anger ett index för ett element som inte finns. w innehåller endast 3 element inte 10.

2.1.7 Utöka och omdefiniera vektorer

Befintliga vektorer kan omdefinieras och användas för att skapa nya vektorer.

Ett sätt att utöka en vektor är att utöka indexmängden och tilldela den nya delen av vektorn värden.

Ett annat sätt är att rekursivt bygga upp en vektor t.ex. med hjälp av en *for*- eller *while*-slinga. Detta är mycket användbart när man vill spara undan information i beräkningssteg.

Vektorerna som skapats i avsnitt 2.1.2 används nedan för att visa de olika sätten.

```
>> % x-vektorn definieras om
>> x = [x 4:5]
x =
  -2  -1   0   1   2   4   5

>> % förändrar innehållet vid index 4
>> x(4) = 45
x =
  -2  -1   0   45   2   4   5

>> % slår samman 2 vektorer till en ny
>> xNew = [w y]
xNew =
  10  15  20  -4  -2  0  2  4

>> % förändrar innehållet vid index 4
>> y(4) = 12
y =
   -4    -2     0    12     4
```

Observera att vektorn $xNew$ är oförändrad.

```
>> % Utökar vektorn values i varje
>> % steg i for-slingan
>> values = []
>> for i = 1:3
      values = [values i];
   end
>> values
values =
     1     2     3
```

För ytterligare exempel på utökad vektor se avsnitt 3.2.

2.1.8 Söka i vektorer

För att söka i en vektor kan funktionen *find* användas. I exemplet nedan returneras de index vars element är större än 1.1

```
>> x = 0:0.5:2;
>> find(x>1.1)
ans =
     4     5
```

2.1.9 Tabeller

Tabeller kan man bygga upp med hjälp av vektorer men det kan också göras med *table*. En av fördelarna med att bygga en tabell med *table* är att man kan lagra data av olika typer.

```
>> x=(0:0.5:2)';
>> y=20*x;
>> land = {'Argentina','Brasilien','Chile','Dominikanska republiken','Equador'}'
>> T = table(land, x, y)

T =

              land                 x     y
    ------------------------    ---    --

    'Argentina'                   0     0
    'Brasilien'                 0.5    10
    'Chile'                       1    20
    'Dominikanska republiken'   1.5    30
    'Equador'                     2    40

>>
```

I exemplet nedan visas hur man kan bygga upp och skriva ut en tabell med hjälp av vektorer och i det här fallet x- och y-koordinater.

```
>> x = 0:0.5:2;
>> y = 20*x;
>> koordinater = [x' y'];
>> disp(koordinater)
         0         0
    0.5000   10.0000
    1.0000   20.0000
    1.5000   30.0000
    2.0000   40.0000
```

I avsnittet 3.2.2 finns ytterligare exempel.

2.2 Matriser

Matriser används t.ex. för att lösa ekvationssystem. En matris består av rader och kolumner. Värdena i en matris kallas för element eller komponenter. Till varje element finns ett index som avspeglar var elementet finns i matrisen. Index skrivs på formen: $radIndex, kolumnIndex$. $radIndex$ anger i vilken rad elementet finns och $kolumnIndex$ anger i vilken kolumn.

Observera att index alltid består av positiva heltal och startar med 1.

Följande ekvationssystem ska skrivas i vektorform

$$\begin{cases} 3x + 2y = 5 \\ 2x + 3y = 10 \end{cases}$$

Det beskrivs med hjälp av $Ak = b$ där

$$A = \begin{pmatrix} 3 & 2 \\ 2 & 3 \end{pmatrix}, \quad k = \begin{pmatrix} x \\ y \end{pmatrix} \quad \text{och} \quad b = \begin{pmatrix} 5 \\ 10 \end{pmatrix}$$

Med en matris storlek menar man hur många rader och kolumner en matris har. Detta kallas även en matris dimension. Dimensionen för matrisen A ovan är 2×2.

För att lösa ekvationssystemet läs mer i kapitel 2.5.

2.2.1 Skapa matriser

En matris skapas genom att ange de värden, element som ska ingå. Nedan visas exempel på de vanligaste metoderna där även ett flertal av Matlabs matris skapande funktioner används. Tänk på att raderna separeras med semikolon eller radbrott och kolumnerna med mellanslag eller komma.

```
>> % Elementen anges inom hårdklamrar.
>> A = [1 2; 3 4]
A =
     1     2
     3     4

>> A = [1, 2
3, 4]

A =
     1     2
     3     4
```

Matrisen A är av dimensionen 2×2. Den första raden innehåller elementen 1 och 2, den andra raden 3 och 4. Den första kolumnen innehåller elementen 1 och 3, den andra 2 och 4.

```
>> % En matris kan byggas upp av
>> % en vektor med vektorer.
>> B = [ 1:3; 4:6]
B =
     1     2     3
     4     5     6


>> b1 = 1:3
b1 =
     1     2     3
>> b2 = 4:6
b2 =
     4     5     6

>> B = [ b1; b2]
B =
     1     2     3
     4     5     6
```

B är en matris av dimension 2×3.

```
>> % ones skapar en matris med ettor.
>> C = ones(2,3)
C =
     1     1     1
     1     1     1
```

C har dimensionen 2×3.

```
>> % zeros skapar en matris med nollor.
>> D = zeros(2,4)
D =
     0     0     0     0
     0     0     0     0
```

D har dimensionen 2×4.

```
>> % eye skapar en enhetsmatris.
>> E = eye(2)
E =
     1     0
     0     1
```

Resultatmatrisen är alltid kvadratisk. Här är E av dimension 2×2.

```
>> % diag skapar en matris med en vektors
>> % element på eller vid diagonalen.
>> f = 1:3
f =
     1     2     3
>> F1 = diag(f)
F1 =
     1     0     0
     0     2     0
     0     0     3

>> F2 = diag(b1,-1)
F2 =
     0     0     0     0
     1     0     0     0
     0     2     0     0
     0     0     3     0
```

Observera att vektorn $b1$:s element ligger ett steg under diagonalen, detta sker tack vare -1 i anropet. Resultatmatrisen har en storlek sådan att vektorns alla element återfinns i matrisen.

```
>> % repmat skapar en matris med en
>> % annan matris som block.
>> G = repmat(F1,3,2)
G =
     1     0     0     1     0     0
     0     2     0     0     2     0
     0     0     3     0     0     3
     1     0     0     1     0     0
     0     2     0     0     2     0
     0     0     3     0     0     3
     1     0     0     1     0     0
     0     2     0     0     2     0
     0     0     3     0     0     3
```

2.2.2 En matris storlek

För att få fram storleken på en matris används *size*. Storleken för några av matriserna i avsnitt 2.2.1 är

```
>> size(F1)
ans =
```

```
     3     3

>> size(G)
ans =
     9     6
```

2.2.3 Transponat av en matris

För att transponera en matris används operatorn ’

```
>> B = [ 1:3; 4:6]
B =
     1     2     3
     4     5     6

>> B'
ans =
     1     4
     2     5
     3     6
```

2.2.4 Åtkomst av matrisers element

Man kan komma åt ett enskilt element och även en del av en matris med hjälp av indexering. Det görs på formen $matrisnamn(radIndex, kolumnIndex)$ Observera att man inte kan ange ett index som inte finns.

Matriserna som skapats i avsnittet 2.2.1 används nedan.

```
>> % Skriver ut matrisen B
>> B
B =
     1     2     3
     4     5     6

>> % Skriver ut elementet i rad 2,
>> % kolumn 3
>> B(2,3)
ans =
     6
>> % Skriver ut elementen i rad 1,
>> % kolumnerna 1 och 2
>> B(1,1:2)
ans =
     1     2

>> % Skriver ut elementen i rad 1,
>> % kolumnerna 1 och därefter
>> B(1,1:end)
ans =
     1     2     3

>> % Skriver ut elementen i rad 1,
>> % alla kolumner. Dvs samma som ovan
>> B(1,:)
ans =
     1     2     3


>> % Skriver ut elementen i alla rader,
>> % kolumnerna 2 och 3
>> B(:,2:3)
ans =
     2     3
     5     6

>> % Plockar ut var 3:e element i
```

```
>> % 1:a kolumnen
>> G(1:3:end,1)
ans =
     1
     1
     1


>> B(3,3)
???  Index exceeds matrix dimensions.
```

Felet som görs här är att man anger ett radindex för en rad som inte finns. B innehåller endast 2 rader inte 3.

2.2.5 Utöka och omdefiniera matriser

Befintliga matriser kan omdefinieras och användas för att skapa nya matriser.

Ett sätt att utöka en matris är att utöka indexmängden och tilldela den nya delen av matrisen värden.

Ett annat sätt är att rekursivt bygga upp en matris t.ex. med hjälp av en *for*- eller *while*-slinga. Detta är mycket användbart när man vill spara undan information i beräkningssteg.

Matriserna som skapats i avsnittet 2.2.1 används nedan.

```
>> % Skriver ut B
>> B
B =
     1     2     3
     4     5     6

>> % En ny rad läggs till B. Observera
>> % att antalet kolumner är lika
>> B = [B; 7:9]
B =
     1     2     3
     4     5     6
     7     8     9

>> % Förändrar innehållet i
>> % rad 1, kolumn 3
>> B(1,3) = 33
B =
     1     2    33
     4     5     6
```

```
     7     8     9

>> % Kolumnvektor definieras
>> b = [34; 45; 56]


b =
    34
    45
    56

>> % B och b slås samman till en
>> % ny matris. Observera att
>> % antalet rader är lika
>> BNew = [B b]
BNew =
     1     2    33    34
     4     5     6    45
     7     8     9    56


>> % Förändrar innehållet i
>> % rad 1, kolumn 3
>> B(1,3) = 3
B =
     1     2     3
     4     5     6
     7     8     9
```

Observera att matrisen $BNew$ är oförändrad.

```
>> % Den fjärde kolumnen tas bort
>> % i BNew
>> BNew(:,4) = []
BNew =

     1     2    33
     4     5     6
     7     8     9

>> % Den första kolumnen i B
>> % kopieras tre gånger till BNew.
>> BNew = B(:,[1 1 1])
BNew =
```

```
     1     1     1
     4     4     4
     7     7     7


>> % Ändrar dimensioner ty kopierar
>> % en delmatris till en vektor.
>> bRad(1:6) = B(1:2,:)
bRad =
     1     4     2     5     3     6


>> % Skapar en vektor som anger index
>> index = [1 3]
index =
      1     3

>> % Elementen vid index i matris B
>> % kopieras till C
>> C = B(index, index)
C =
     1     3
     7     9
```

För att utöka en matris i en *for*- eller *while*-slinga se avsnittet 3.2.2.

Varnande exempel

Nedan följer ett antal mycket vanliga felaktiga kommandon och deras respektive felmeddelande.

```
>> % Kan inte tilldela en position
>> % i B den tomma matrisen
>> B(3,3) = []
???  Indexed empty matrix assignment
is not allowed.


>> % Kan inte skriva ut rad 4 ty finns ej
>> BNew = B(4,:)
???  Index exceeds matrix dimensions.


>> % B av dimension 3*3 kan inte läggas
```

```
>> % in i ett mindre utrymme, ty
>> % dimensionen av BNew(1:2,:) är 2*3
>> BNew(1:2,:) = B
??? Subscripted assignment
    dimension mismatch.


>> % Kan inte lägga till en ny rad
>> % med olika antal kolumner
>> B = [B; 7:11]
???  Error using ==> vertcat
All rows in the bracketed expression
must have the same number of columns.
```

2.2.6 Matrismanipulerande funktioner

Funktioner som t.ex. *reshape* och : kan omforma matriser och vektorer. *reshape* skrivs på formen $reshape(A, rader, kolumner)$ där elementen i A omformas till en matris av storleken $rader \times kolumner$.

I exemplen nedan utgår man från matrisen B

```
>> B = [ 1:3; 4:6]
B =
     1     2     3
     4     5     6
```

För att få alla element i B som en kolumnvektor kan man skriva på följande två sätt

```
>> b = B(:)
b =
     1
     4
     2
     5
     3
     6

>> reshape(B,6,1)
ans =
     1
     4
     2
     5
     3
     6
```

Med hjälp av *reshape* kan man omforma en vektor eller matris.

```
>> reshape(b,3,2)
ans =

     1     5
     4     3
     2     6


>> reshape(B,3,2)
ans =
     1     5
     4     3
     2     6
```

Resultatet av de båda kommandona är lika.

Varnande exempel

När man använder *reshape* måste antalet element överenstämma mellan indatavektorn och resultatvektorn. I exemplet nedan har indatavektorn 6 element och utdatavektorn 9 vilket genererar felmeddelandet.

```
>> reshape(b,3,3)
??? Error using ==> reshape
To RESHAPE the number of elements
 must not change.
```

2.2.7 Matrisfunktioner

I Matlab finns ett flertal funktioner som opererar på matriser. Här visas endast ett urval.

```
>> % A definieras
>> A = [1 2; 3 4]
A =

     1     2
     3     4
```

Rangen för en matris

```
>> rank(A)
ans =
     2
```

Determinanten för en matris

```
>> det(A)
ans =
     -2
```

Inversen för en matris

```
>> inv(A)
ans =
   -2.0000    1.0000
    1.5000   -0.5000
```

Konditionstalet för en matris då den euklidiska normen används

```
>> cond(A)
ans =
     14.9330
```

Konditionstalet för en matris då maxnormen används

```
>> cond(A,inf)
ans =
     21.0000
```

Egenvärden för en matris

```
>> eig(A)
ans =
   -0.3723
    5.3723
```

Den euklidiska normen för en matris

```
>> norm(A)
ans =
    5.4650
```

Maxnormen för en matris

```
>> norm(A,inf)
ans =
    7
```

2.2.8 Söka i matriser

För att söka i en matris kan funktionen *find* användas. I exemplet nedan returneras de (linjära) index vars element är lika 1

```
>> A = eye(3);
>> find(A==1)'
ans =
     1     5     9
```

2.2.9 Glesa matriser

I vissa tillämpningar får man en stor matris med få element som är skilda från noll. Dessa matriser kallas för glesa matriser. Istället för att lagra hela matrisen lagrar man den viktiga informationen dvs de element som är skilda från noll.

Om A är en gles $n \times n$-matris så lagras den på gles form med kommandot *sparse*

```
>> sparseA = sparse(A);
```

För att fylla matrisen används *full*

```
>> B = full(sparseA);
```

Matriserna A och B innehåller likadana element.

Strukturen hos en gles matris på gles form kan man se med hjälp av kommandot *spy*

```
>> spy(sparseA)
```

Ett ekvationssystem med en gles matris som t.ex. $Ax = b$ där A och $sparseA$ är definierad som ovan och b är en $n \times 1$-vektor kan man lösa med \. Mer om ekvationslösning finns i kapitel 2.5.

Snabb lösning

```
>> x = sparseA\b;
```

Långsam lösning

```
>> x = A\b;
```

2.3 Matematiska operationer

I detta avsnitt presenteras de aritmetiska operationerna och elementära matematiska funktioner. Matematikens prioriteringsregler används och parenteser kan användas för att förändra prioritetsordningen.

2.3.1 Aritmetiska operationer

Addition, subtraktion, multiplikation och division skrivs i Matlab som $+$, $-$, $*$ respektive $/$. I Matlab finns ytterligare tre operationer definierade. Det är upphöjt till, transponat och vänsterdivision som skrivs med tecknen $\hat{}$, $'$ respektive $\backslash$.

Dessa operationer kan alla användas vid beräkning av skalärer, vektorer och matriser. I Matlab används dessutom punktnotation för att tillåta elementvis operation vilket beskrivs i avsnitt 2.3.2.

I avsnittet 2.3.3 visas hur dessa operationer används och ger även exempel på vanliga fel.

2.3.2 Punktnotation

Med punktnotation menar man att man sätter en punkt framför aktuell operator. Dessa skrivs alltså som $.*$ $./$ $.\backslash$ $.\hat{}$ $.'$. När man använder punktnotation innebär det att operationen utförs elementvis. För operationerna $+$ och $-$ finns ingen motsvarande punktnotation eftersom de redan arbetar elementvis.

För att visa hur en elementvis operation går till ges här två exempel. I båda exemplen används multiplikation.

Elementvis multiplikation med vektorer

$$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix} .* \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \Rightarrow$$

$$\Rightarrow \begin{pmatrix} 1*4 & 2*5 & 3*6 \end{pmatrix} \Rightarrow$$

$$\Rightarrow \begin{pmatrix} 4 & 10 & 18 \end{pmatrix}$$

Elementvis multiplikation med matriser

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} .* \begin{pmatrix} pi & 0.5 \\ 5 & 10 \end{pmatrix} \Rightarrow$$

$$\Rightarrow \begin{pmatrix} 1*pi & 2*0.5 \\ 3*5 & 4*10 \end{pmatrix} \Rightarrow$$

$$\Rightarrow \begin{pmatrix} pi & 1 \\ 15 & 40 \end{pmatrix}$$

I avsnitt 2.3.3 finns fler exempel.

2.3.3 Algebra

Nedan följer diverse exempel för att visa olika operationer med skalärer, vektorer och matriser.

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

>> 2*A
ans =
     2     4
     6     8

>> A + A
ans =
     2     4
     6     8
Matriserna måste vara av samma dimension

>> 2+A
ans =
     3     4
     5     6

>> A^2
ans =
     7    10
    15    22
Matrisen måste vara kvadratisk

>> A*A
ans =
     7    10
    15    22
Antalet kolumner i vänster matris
måste vara lika med antalet rader
i höger matris

>> A.^2
ans =
     1     4
     9    16
Multiplikationen sker elementvis

Skapar en ny matris
>> B = [10 20;20 10]
B =
```

```
    10    20
    20    10

>> A*B
ans =
    50    40
   110   100
Antalet kolumner i vänster matris,
A måste vara lika med antalet rader
i höger matris, B.

Skapar en vektor
>> x = [2 10]
x =
     2    10

>> x*A
ans =
    32    44


>> A*x'
ans =
    22
    46

>>  x*x'
ans =
    104

>>  x'*x
ans =
     4    20
    20   100

>> x.*x
ans =
     4   100
Multiplikationen sker elementvis.
```

2.3.4 Elementära matematiska funktioner

Nedan följer en tabell med några vanliga funktioner.

abs()	absolutbelopp
sqrt()	kvadratrot
round()	avrundar till närmaste heltal
fix()	avrundar till närmaste heltal mot noll
floor()	avrundar till närmaste heltal nedåt
ceil()	avrundar till närmaste heltal uppåt
sign()	ger tecknet
rem()	ger resten vid heltalsdivsion
sin()	sinus, indata är i *radianer*
sind()	sinus, indata är i *grader*
cos()	cosinus, indata är i *radianer*
cosd()	cosinus, indata är i *grader*
tan()	tangens, indata är i *radianer*
tand()	tangens, indata är i *grader*
asin()	arcsinus
acos()	arccosinus
atan()	arctangens
exp()	exponentialfunktionen e^x
log()	naturliga logaritmen
*log*10()	10-logaritmen

Indata till dessa funktioner kan var en skalär, vektor eller matris. Funktionerna opererar alltid elementvis på vektorer och matriser. Här följer några exempel

```
>> x = 9;
>> sqrt(x)
ans = 3

>> y = [4 -16 9];
>> sqrt(y)
ans = 2.0000   0+4.0000i   3.0000

>> cos(acos(pi))
ans = 3.14159265358979

>> A = [pi pi/2; 3*pi/5 pi/4];
>> sin(A)
ans =
    0.0000    1.0000
    0.9511    0.7071
```

Tänk på att svaret från dessa funktioner kan vara ett komplext tal.

2.3.5 Flera matematiska funktioner

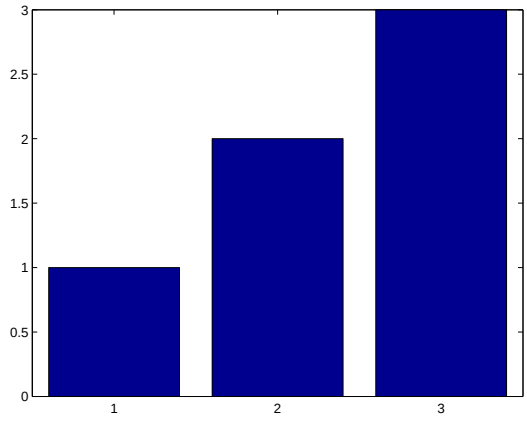
Nedan följer en tabell med några vanliga funktioner.

$bar()$	stapeldiagram
$diff()$	differens mellan element
$hist()$	histogram
$max()$	max element
$mean()$	medelvärde
$median()$	medianvärde
$min()$	min element
$prod()$	produkten av elementen
$rand()$	genererar slumptal
$sort()$	sorterar element i stigande ordning
$sum()$	summerar elementen

Indata till funktionerna kan vara skalärer, vektorer eller matriser. Då indata är matriser utförs funktionen kolumnvis. Nedan följer ett par exempel med vektor och matris.

```
>> v = 1:3
v =
     1     2     3

>> sum(v)
ans =
     6

>> min(v)
ans =
     1

>> diff(v)
ans =
     1     1

>> bar(v)
>> % se figur nedan

>> B = [1 5 3; 4 2 6]
B =
     1     5     3
     4     2     6
```

```
>> diff(B)
ans =
     3    -3     3

>> mean(B)
ans =
    2.5000    3.5000    4.5000

>> max(B)
ans =
     4     5     6

>> rand()
ans =
    0.2093

% slumpar fram 2 heltal mellan 1 t.o.m. 5
>> ceil(5.*rand(2,1))
ans =
     3
     1
```



Figur 2.1: bar(v)

2.4 Polynom

I Matlab finns ett flertal funktioner för att arbeta med polynom. För att kunna använda dessa måste man veta att ett polynom kan avbildas med hjälp av vektorer. Ta t.ex. 3:e-gradspolynomet $p1(x) = 10x^3 + 5x - 13$ som kan skrivas som
$p1(x) = 10x^3 + 0x^2 + 5x^1 - 13x^0$. Det beskrivs med hjälp av vektorn: $[10 \quad 0 \quad 5 \quad -13]$. Elementen i vektorn svarar mot koefficienterna i polynomet. Koefficienten för termen med den högsta potensen står först i vektorn, övriga följer därefter i ordning och sist är den koefficient vars term innehåller x^0.

Observera att koefficienter som är lika noll måste anges som noll i vektorn samt att vektorns längd alltid är ett större än polynomets gradtal. Vektorn $[4 \quad 0 \quad 0]$ beskriver alltså 2:a-gradspolynomet $p2(x) = 4 * x^2$.

2.4.1 Finn rötterna till ett polynom

Funktionen *roots* returnerar ett polynoms rötter.

```
>> % p1 definierar polynomet p1(x)
>> p1 = [10 0 5 -13];
>> r1 = roots(p1)
r1 =

  -0.4699 + 1.0782i
  -0.4699 - 1.0782i
   0.9398
```

De tre rötterna returneras som en kolumnvektor. Två av rötterna är komplexa.

```
>> % p2 definierar polynomet p2(x)
>> p2 = [4 0 0]
>> r2 = roots(p2)
r2 =

     0
     0
```

2.4.2 Finn ett polynom till rötterna

Från en uppsättning rötter kan ett motsvarande polynom definieras med hjälp av *poly*

```
>> % Rötterna definieras av variabeln r1
>> p11 = poly(r1)
```

```
p11 =
    1.0000   -0.0000    0.5000   -1.3000
```

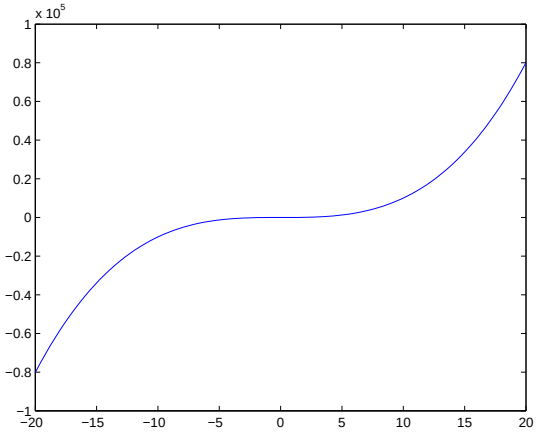
Här returneras en radvektor vars element är koefficienterna till polynomet $p11(x) = x^3 + x/2 - 1.3$

Observera att koefficient för den term med den högsta potensen alltid är ett.

2.4.3 Beräkna ett polynom

För att enkelt kunna beräkna ett polynom kan *polyval* användas. Här beräknas och plottas 3:e-gradspolynomet $p1(x) = 10x^3 + 5x - 13$ då $x \in [-20, 20]$

```
>> % p1 definierar polynomet p1(x)
>> p1 = [10 0 5 -13]
>> % x1, vektor med värden på x
>> x1 = linspace(-20, 20);
>> % values1, vektor med värdena p1(x1)
>> values1 = polyval(p1,x1);
>> % polynomet plottas
>> plot(x1,values1)
```



Figur 2.2: $p1(x) = 10x^3 + 5x - 13$

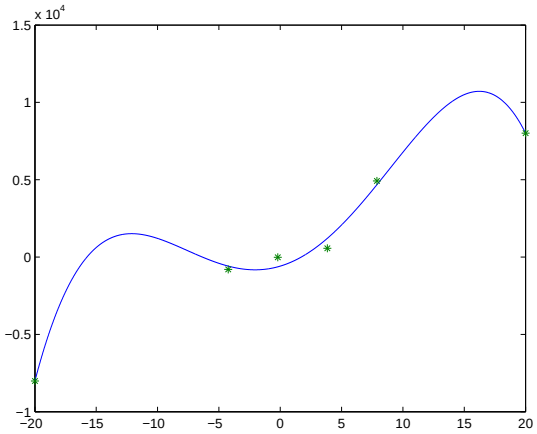
2.4.4 Anpassa ett polynom till mätdata

För att anpassa ett polynom till en given mängd mätdata används $polyfit$. Kommandot $polyfit$ beräknar det polynom som bäst passar mätdata. Med bäst menas här att mätdata anpassas till polynomet i minstakvadratenmening. När man anropar $polyfit$ skickar man med mätpunkter, mätvärden

och det gradtal man vill att polynomet ska ha. En radvektor som beskriver det polynom som anpassats till mätdata returneras. Tänk på att polynomets gradtal inte kan vara större än antal mätdata - 1 för att vara entydigt. Observera också att då ett polynom har ett högt gradtal kan man råka ut för Runges fenomen.

Nedan visas hur man kan använda sig av $polyfit$ för att till mätpunkterna i vektorn $xData$ och mätvärdena i $yData$ anpassa ett 4:e-gradspolynom, $p(x)$. Vektorn p beskriver $p(x)$. $p(x)$ plottas slutligen i ett finare x-intervall tillsammans med mätdata, se figur 2.3.

```
>> xData = [-20 -4.24 -0.20 3.84 7.88 20];
>> yData = [-8010 -800 -14 570 4920 8010];
>> gradtal = 4;

>> % p beskriver p(x)
>> p = polyfit(xData,yData,gradtal);
>> % Ett finare x-intervall skapas
>> x = -20:0.01:20;
>> % Polynomet beräknas för x-intervallet
>> values = polyval(p,x);
>> % Polynomet plottas. Mätdata prickas ut
>> plot(x,values,xData,yData,'*')
```



Figur 2.3: Ett 4:e-gradspolynom anpassat till data.

2.5 Linjär algebra

För att lösa ett ekvationssystem $Ak = b$ används operatorn \. Då systemet är välbestämt dvs A har dimensionen $n \times n$ används Gausselimination. Då

systemet är under- eller överbestämt dvs A har dimensionen $n \times m$ där $n \neq m$ erhålls en minstakvadratenlösning.

För att lösa ekvationssystemet som beskrivs i inledningen av kapitel 2.2 skrivs följande

```
>> A = [3 2; 2 3]
A =
     3     2
     2     3

>> b = [5 10]'
b =
     5
    10
```

Systemet löses:

```
>> k = A\b
k =
    -1.0000
     4.0000
```

2.5.1 Varnande exempel

Se upp för följande fel

```
>> A\b'
Error using  \
Matrix dimensions must agree.

>> k = A/b
??? Error using ==> /
Matrix dimensions must agree.

>> k = A/b'
k =
    0.2800
    0.3200
```

Lägg märke till att i de två sista fallen används felaktig operator om man vill lösa ekvationssystemet.

2.6 Numeriska funktioner

Nedan följer en tabell med några vanliga numeriska funktioner.

- dblquad($fun, xmin, xmax, ymin, ymax$)
  Beräknar dubbelintegralen av fun över området $x \in [xmin, xmax]$, $y \in [ymin, ymax]$.
- fplot(fun,$limit$)
  Funktionen fun plottas för x-värdena i intervallet $limit = [xmin, xmax]$.
- fsolve(fun,$x0$)
  Beräknar nollstället till funktionen fun med startvärdet $x0$. Löser ekvationer på formen: F(X)=0 där F och X är vektorer eller matriser.
- fzero(fun,$x0$)
  Beräknar nollstället till funktionen fun med startvärdet $x0$. Löser ekvationer på formen: f(x)=0 där f och x är skalära värden.
- ode23
  Löser icke-styva differentialekvationer. Metod av låg ordning. Se avsnitt 2.7.
- ode45
  Löser icke-styva differentialekvationer. Metod av medelordning. Se avsnitt 2.7.
- quad(fun,A,B)
  Beräknar integralen av fun över intervallet $[A, B]$.
- quadl(fun,A,B)
  Beräknar integralen av fun över intervallet $[A, B]$. Kan vara mer effektiv vid hög noggrannhet och integrander som är glatta.

Till alla dessa funktioner är indata bland annat en funktion, ovan kallad fun. Funktionen kan bland annat definieras i funktionsfil eller som anonym funktion. För att läsa mer om olika funktioner se avsnitt 3.3.

Nedan följer ett par exempel som visar hur anropen kan se ut. I de första exemplen används funktionsfil därefter görs samma anrop med en anonym funktion.

`Med funktionsfil:`

```
% M-filen sq2.m skrivs som:
function res = sq2(x)
```

```
res =  x.^2-2;
% slut på filen sq2.m
```

```
>> % Nollstället till sq2 beräknas
>> fzero('sq2', 2)
ans =
    1.4142
```

```
>> % Integralen av sq2 över
>> % intervallet [0, 2] beräknas
>> quad('sq2',0,2)
ans =
   -1.3333
```

Med anonym funktion:

```
>> % Den anonyma funktionen sq2 definieras
>> sq2 = @(x) x.^2-2
sq2 =
    @(x) x.^2-2
```

```
>> % Nollstället till sq2 beräknas
>> fzero(sq2,2)
ans =
    1.4142

>> % Integralen av sq2 över
>> % intervallet [0, 2] beräknas
>> quad(sq2,0,2)
ans =
   -1.3333
```

Lägg märke till att funktionerna använder sig av punktnotation detta eftersom indata till *quad* måste kunna vara en vektor. För ytterligare exempel se avsnitt 3.3.

2.7 Differentialekvationer

För att lösa ordinära differentialekvationer som är formulerade som ett begynnelsevärdesproblem finns ett antal olika lösare, bland annat *ode*23 och *ode*45.

För att använda dessa lösare måste differentialekvationen skrivas som ett system på formen $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ där $\mathbf{y}(t_0) = \mathbf{y}_0$.

Ett anrop till *ode*23 har formen
$[t_{out}, y_{out}] = ode23('fun', timespan, y0, options)$.
Anropet till *ode*45 är analogt.

Indata är

- funktionen fun som beskriver $\mathbf{f}(t, \mathbf{y})$. Eftersom $\mathbf{f}$ och $\mathbf{y}$ är vektorer måste funktionen fun returnera en kolumnvektor som motsvarar $\mathbf{f}(t, \mathbf{y})$.

- tidsintervallet *timespan* som är en radvektor som anger start- och sluttid.
  $timespan = [t_0, t_{slut}]$.

- begynnelsevärden $y0$ som är en kolumnvektor som beskriver $\mathbf{y}(t_0)$.

- valfria *options* som sätts med hjälp av kommandot *odeset*. Detta används för att ändra t.ex. toleransen.

Utdata är

- vektorn t_{out} där alla tidpunkterna finns

- matrisen y_{out} som innehåller lösningen till systemet vid de olika tidpunkterna i t_{out}.

2.7.1 Exempel

Säg att man vill lösa differentialekvationen
$y'' + y = e^{-t}$ där $y(0) = 1$ och $y'(0) = 0$ för tiden $t \in [0, 10]$.

För att lösa ekvationen skrivs det om till systemet

$$\mathbf{y}' = \begin{pmatrix} y1' \\ y2' \end{pmatrix} = \begin{pmatrix} y2 \\ -y1 + e^{-t} \end{pmatrix}$$

där

$$\mathbf{y}(0) = \begin{pmatrix} y1(0) \\ y2(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

Problemet är nu på den önskade formen $\mathbf{y}' = \mathbf{f}(t, \mathbf{y})$ där $\mathbf{y}(t_0) = \mathbf{y}_0$. Funktionsfilen $f.m$ definieras för systemet

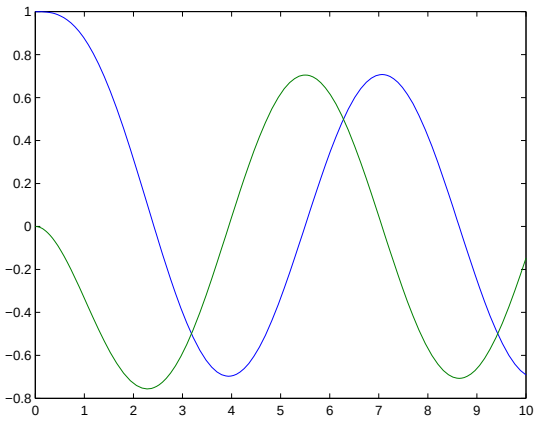
```
function res = f(t,y)


res = [y(2); -y(1) + exp(-t)];

% slut på f.m
```

Nu kan ekvationen lösas och anropet skrivs

```
>> tidV = [0, 10];
>> y0 = [1 0];
>> [tout,yout]=ode23('f',tidV,y0)
```

I vektorn *tout* ligger de olika tidpunkterna. Lägg märke till att tidsstegen inte är konstanta utan att de varierar. Observera att i detta exempel är *yout* en matris med två kolumner, i den första finns alla y-värden och i den andra alla y'-värden. För att plotta lösningen kan man skriva

```
>> plot(tout,yout)
```



Figur 2.4: Lösning till $y'' + y = e^{-t}$

Det finns möjlighet att ange t.ex. relativ- och absolutfelet för beräkningarna. Genom anropet

```
>>options=odeset('reltol',1e-4,'abstol',1e-8)
```

sätts tolerenserna för felen. Anropet till *ode*23 blir nu istället

```
>> tidV = [0, 10];
>> y0 = [1 0];
>> [tout,yout]=ode23(@f,tidV,y0,options)
```

2.8 Villkor

För att kunna jämföra olika värden med varandra finns relationsoperatorer. Uttryck som använder relationsoperatorer kallas för villkor. För att sätta samman olika villkor, relationer, används de logiska operatorerna. Resultatet av både relationsoperatorer och logiska operatorer är alltid ett sanningsvärde dvs antingen falskt eller sant. I Matlab är 0 falskt och 1 sant. Relationsoperatorerna har högre prioritet än de logiska operatorerna men lägre än de aritmetiska operationerna.

2.8.1 Relationsoperatorer

Relationsoperatorerna är:

```
mindre än                       <
större än                       >
mindre än eller lika med        <=
större än eller lika med        >=
likhet                          ==
olikhet                         ~=
```

Ett litet exempel är att testa om talet 3 är lika med 3. Eftersom det är sant så returneras 1.

```
>> 3==3
ans =
     1
```

Man kan även jämföra vektorer med relationsoperatorerna. Svaret blir då en vektor med sanningsvärden.

```
>> [1 2 3] == [1 5 4]
ans =
     1     0     0
```

Jämförelsen har alltså utförts elementvis.

Det fungerar också att jämföra en vektor med ett skalärt värde. Svaret blir även här en vektor med sanningsvärden.

```
>> [1 2 3] == 2
ans =
     0     1     0
```

Varnande exempel

Tänk på att = används då man vill tilldela en variabel ett värde! Det är mycket vanligt att man råkar skriva = istället för ==.

```
>> x = 3;
>> y = 4;
>> x = y
x =
     4
```

Variabeln x har fått värdet 4 istället för att man jämfört x med y!

2.8.2 Logiska operatorer

De logiska operatorerna är:

```
och            &
eller          |
icke           ~
```

Ett exempel som kontrollerar om både variabeln x och y är positiva tal visas nedan.

```
>> x=2;
>> y=1;
>> (x>0) & (y>0)
ans =
     1
```

Eftersom båda variablerna är positiva returneras 1 (sant).

Det går också bra att använda vektorer med de logiska operatorerna.

```
>> ([1 2 3] == [1 5 4]) & ([1 2 3] == 2)
ans =
     0     0     0

>> [1 2 3] == [1 5 4] | [1 2 3] == 2
ans =
     1     1     0
```

Kapitel 3

Programmering

3.1 Villkorlig sats

För att kunna välja mellan olika alternativ finns konstruktioner som bland annat *if*, *switch* och *case*. Här beskrivs endast *if* konstruktionen.

3.1.1 IF-sats

En *if*-sats skrivs på formen:

```
if villkor
   sats(er), utförs om villkor är sant
else
   sats(er), utförs om villkor är falskt
end
```

eller om man vill testa flera olika villkor

```
if villkor1
   sats(er), utförs om villkor1 är sant
elseif villkor2
   sats(er), utförs om villkor2 är sant
   och villkor1 är falskt
   .
   .
   .
end
```

Observera att:

- det till varje *if* måste finnas ett *end* som avslutar *if*-satsen.
- man INTE avslutar *elseif* med *end*.
- *else*-delen kan utelämnas.

- satserna som utförs kan i princip vara vilken Matlab-kod som helst t.ex. if- eller for-sats.

Inledande exempel

Säg att man vill skriva ut absolutbeloppet av ett reellt tal x. Man kan då börja med att undersöka om x är mindre än noll. Om det är sant skrivs resultatet av det negerade talet ut dvs $-x$ annars skrivs x ut.

En beskrivning som löser problemet är

```
om x<0 skriv ut -x annars skriv ut x
```

Beskrivningen, algoritmen, skrivs i Matlab som

```
if x<0
   disp(-x)
else
   disp(x)
end
```

Nedan visas hur exemplet körs i kommandofönstret. if-satsen har fått några extra rader som skriver ut lite förtydligande text.

```
>> x = -3;

>> if x<0
      disp('Absolutbeloppet av')
      disp('det negativa talet är:')
      disp(-x)
   else
      disp('Absolutbeloppet är:')
      disp(x)
   end   % if-satsen slutar här

Absolutbeloppet av
det negativa talet är:
     3
```

Först tilldelas x värdet -3, varefter if-satsen definieras och körs och slutligen skrivs texten samt resultatet 3 ut.

Ytterligare exempel

I följande exempel utökas det inledande exemplet med att särskilja fallen $x > 0$ och $x = 0$. Om $x = 0$ returneras noll annars roten ur absolutbeloppet av x.

```
>> x = -4;
>> if x<0
      disp('x är negativt')
      disp(sqrt(-x))
   elseif x>0
      disp('x är positivt')
      disp(sqrt(x))
   else
      disp('x är lika noll')
      disp(0)
   end % if slut

x är negativt
ans =
     2
```

3.2 Repetition - upprepning

För att upprepa t.ex. en viss typ av beräkningar finns i Matlab *for* och *while* konstruktioner. Man kan även skriva egna rekursiva funktioner se avsnitt 3.3.

3.2.1 For-sats

En *for*-sats används med fördel när man vet hur många gånger något ska utföras dvs hur många gånger en sats eller flera ska upprepas.

En *for*-sats skrivs på formen

```
for variabel = uttryck
   sats(er)
end
```

där satserna som utförs kan i princip vara vilken Matlab-kod som helst t.ex. *if*- eller *for*-sats.

Inledande exempel

Säg att man vill skriva ut en rad med en snöflinga tre gånger. Eftersom man vet hur många gånger raden ska skrivs ut passar en *for*-sats bra. Uppräkningen kan då ordnas med att en variabel t.ex. i tilldelas värdet 1 för första varvet, värdet 2 för andra varvet och slutligen värdet 3 för det tredje och avslutande varvet. Variabeln i håller alltså i detta fall reda på hur många gånger upprepningen sker. I Matlab motsvaras det helt av raden $for \quad i = 1:3$, för mer om kolonnotation se avsnitt 2.1.1. Den enda satsen som ska utföras är att snöflingan ska skrivas ut på en rad, vilket görs med kommadot $disp('*')$. För att markera slutet på *for*-satsen stängs den med *end*.

Nedan visas hur for-satsen definieras och körs i kommandofönstret samt att tre rader med en snöflinga skrivs ut.

```
>> for i = 1:3
      disp('*')
   end % for-satsen slut
*
*
*
```

Uppräkningen kan även göras med hjälp av en vektor. När man vill summera elementen i en vektor kan man skriva

```
>> sum = 0;
>> for v = [1 10 15]
     sum = sum + v;
  end  % for-satsen slut

>> sum
sum =
    26
```

I det första varvet är $v = 1$ samt $sum = 0 + 1$, andra varvet är $v = 10$ samt $sum = 1 + 10$ och slutligen är $v = 15$ samt $sum = 11 + 15$. Precis som i exemplet innan så tilldelas variabeln v element från vektorn i tur och ordning.

Ytterligare exempel

Nedan följer ytterligare några exempel.

När man beräknar fakulteten av det godtyckliga talet n kan man använda

```
>> n = 3;
>> fakultet_n = 1;
>> for i = 1:n
     fakultet_n = fakultet_n * i;
   end % for-sats slut

>> fakultet_n
fakultet_n =
     6
```

När man vill spara undan information i de olika stegen i for-slingan går det att göra med en vektor på följande sätt

```
>> v = [];
>> for i = 1:4
     v = [v i+1];
  end   % for-sats slut

>> v
v =
     2     3     4     5
```

Jämför gärna med det liknande exemplet i avsnittet 3.2.2.

3.2.2 While-sats

En *while*-sats används med fördel när man inte vet hur många gånger något ska utföras.

En *while*-sats skrivs på formen

```
while villkor
   sats(er)
end
```

där satserna som utförs kan i princip vara vilken Matlab-kod som helst t.ex. if- eller for-sats.

Inledande exempel

Säg att man vill summera den harmoniska serien
$1/1 + 1/2 + \ldots + 1/n$ för att ta reda på det minsta värde på n som gör att summan är större än 1.8. Summan $1/1 + 1/2 + 1/3$ är lika med $1.8333\ldots$. Eftersom det är större än 1.8 så är det minsta värdet på $n = 3$.

En beskrivning, algoritm, som löser problemet är

```
starta med att nollställa, dvs
summan = 0 och n = 0

så länge som summan är mindre än eller
lika med 1.8 så gör:

   * räkna upp n med ett
   * till den befintliga summan läggs 1/n
```

Algortimen översätts till följande Matlab-kod som här skrivs och körs direkt i Matlabs kommandofönster

```
>> summan = 0;
>> n = 0;

>> while summan <= 1.8
      n = n + 1;
      summan = summan + 1/n;
   end   % while-slinga slut

>> summan
summan =
    1.8333

>> n
n =
     3
```

När koden körs utförs följande steg i while-slingan:
först kontrolleras om villkoret dvs *summan* är mindre än eller lika med 1.8. Eftersom detta är sant utförs satserna inuti while-slingan. n får då värdet 1 och *summan* får då värdet 1 ty $0 + 1/1$. Eftersom alla satserna inuti while-slingan nu har gjorts kontrolleras åter villkoret för slingan. *summan* är fortfarande mindre än eller lika med 1.8 alltså utförs satserna inuti while-slingan ytterligare en gång. Detta upprepas alltså tills villkoret blir falskt, vilket sker efter tre varv i detta problem.

Ytterligare exempel

Nedan följer ytterligare några exempel.

När man beräknar fakulteten av det godtyckliga talet n kan man använda

```
>> n = 4;
>> fakultet_n = 1;

>> while n > 0
     fakultet_n = fakultet_n * n;
     n = n-1;
   end  % while slut

>> fakultet_n
fakultet_n =
     24
```

För att beräkna $sqrt(2)$ med hjälp av Newton-Raphsons metod samt när man vill spara undan information i de olika stegen i en *while*-slinga går det att göra med en matris på följande sätt

```
>> % initierar
>> x = 1;
>> h = 1;
>> data = [];
>> tolerans = 1e-8;
>> iter = 1;

>> % Newton-Raphson
>> while abs(h) > tolerans & iter < 5
     f = x^2-2;
     fprim = 2*x;
     h = f/fprim;
     % sparar information
     data = [data; iter x f fprim h];
     x = x - h;
     iter = iter + 1;
   end  % while slut

>> x
x =
    1.4142

>> % skriver ut tabellen med data
>> % kolumnerna är: iter x f fprim h
>> data
data =
    1.0000    1.0000   -1.0000    2.0000   -0.5000
    2.0000    1.5000    0.2500    3.0000    0.0833
    3.0000    1.4167    0.0069    2.8333    0.0025
    4.0000    1.4142    0.0000    2.8284    0.0000
```

Observera att denna slinga avbryts då $iter = 5$. Jämför gärna med det liknande exemplet i avsnittet 3.2.1.

Varnande exempel

Varning! Det är mycket lätt att skriva ett villkor och en *while*-slinga felaktigt. Ett klassiskt fel är då man får en oändlig upprepning, loop.

```
>> n = 3;
>> fakultet_n = 1;
>> while n > 0
    fakultet_n = fakultet_n * n
  end % while-slinga slut
```

Eftersom n alltid är 3 i *while*-slingan avbryts den aldrig. För att avbryta körningen tryck ned *ctrl*-tangenten samtidigt som tangenten c, dvs $ctrl - c$.

3.3 Skapa egna funktioner

För att underlätta arbetet i Matlab kan man skriva egna funktioner på i huvudsak tre olika sätt.

Då man vill återanvända sina funktioner används funktionsfiler. På detta sätt kan man skapa en uppsättning egna funktioner som är lätta att använda i t.ex. script och funktionsfiler. När det däremot inte finns ett krav på återanvändbarhet är anonyma och inline funktioner att föredra.

De tre olika sätten att skapa egna funktioner beskrivs med hjälp av en exempelfunktion. Funktionen som definieras är den matematiska funktionen för att beräkna kvadraten för ett godtyckligt reellt tal. Matematiskt kan man skriva definitionen som

$$square(x) = x^2$$

För att beräkna kvadraten för $x = 2$ skriver man

$$square(2)$$

I datasammanhang kallas detta för att anropa en funktion.

Funktionens namn är *square* och funktionskroppen dvs x^2 anger vad som ska göras. I detta fall upphöjs värdet på x till 2.

3.3.1 Funktionsfil

En funktionsfil är en Matlabfil, d.v.s en M-fil, som man kan skicka indata till och som kan returnera utdata. Genom att på detta sätt spara funktionen i en separat M-fil kan den återanvändas gång efter annan. Funktionsfilen skrivs i M-filen $funktionsnamn.m$ och den generella formen är

```
function utvar=funktionsnamn(invar)
% Ange bra sökord på denna s.k. H1-rad.
% Dessa rader är kommentarer och här
% skrivs funktionens hjälptext.

funktionskropp
%funktionen slut
```

Lägg märke till att filen har samma namn som funktionen. Välj ett beskrivande funktionsnamn som inte redan används varken av användaren eller Matlab.

Indata som skickas med i funktionsanropet tas emot av *invar*, dvs indata kopieras till *invar*. Variabeln *utvar* ser till att funktionen kan returnera ett

svar från funktionen. Både *invar* och *utvar* kan utelämnas. I funktionskroppen anges vad som ska göras.

För att funktionen ska returnera ett värde måste funktionskroppen innehålla en tilldelning till *utvar*.

Funktionen kan anropas från kommandofönstret, ett script eller från en annan funktion. Anropet har formen

```
funktionsnamn (indata)
```

När en funktion anropas utförs, exekveras, funktionfilens rader i tur och ordning.

Exempelfunktionen *square*

Exempelfunktionen *square* skrivs i filen *square.m*

```
function result = square(x)
% KVADRAT square
% Beräknar kvadraten för reella tal.
% Indata kan vara skalär eller vektor.

% funktionskroppen börjar
result = x.^2;
% funktionskroppen slutar
```

När *square* anropas från kommandofönstret med

```
>> svar = square(2);
```

får variabeln x i funktionen *square* värdet 2. I funktionskroppen beräknas sedan kvadraten av 2 till 4 som därefter tilldelas variabeln *result*. Eftersom *result* har angetts som ett utdata kommer dess värde 4 att returneras som ett svar från funktionsanropet. Variabeln *svar* tilldelas alltså 4.

```
>> svar
svar =
     4
```

Till skillnad mot den matematiska definitionen av *square* kan den Matlabdefinierade funktionen *square* beräkna kvadraten av varje element i en vektor. Detta är möjligt eftersom punktnotationen .^ har använts istället för ^ , för punktnotation se avsnitt 2.3.2. När *square* anropas från kommandofönstret med en vektor returneras en vektor. Varje elementen i vektorn har kvadrerats som syns i exemplet nedan.

```
>> vektor = [1 2 3];
>> svarsVektor = square(vektor)
svarsVektor =
     1     4     9
```

Flera indata

I de fall man behöver skicka in fler värden än ett till en funktion sker det genom att man istället för den generella funktionsfilens *invar* specificerar variablerna $in_1, in_2, ...in_n$. Då funktionen anropas tilldelas variablerna i tur och ordning de värden som anges vid anropet. Anropet $funktionsnamn(indata_1, indata_2, ...indata_n)$ leder alltså till att $in_1 = indata_1$, $in_2 = indata_2$ osv.

Ett exempel på en sådan funktion ses i avsnittet 3.3.1 nedan.

Flera utdata

För att en funktion ska returnera fler än ett utdata ersätter man den generella funktionsfilens *utvar* med en vektor $[ut_1, ut_2, ...ut_n]$ som innehåller lika många variabler som värden man vill returnera. Variablerna tilldelas värden i funktionskroppen.

Nedan följer ett exempel som beräknar koordinaterna för enhetscirkeln. Lägg märke till att funktionen inte har något indata men att den returnerar två vektorer som utdata.

```
function [x,y] = cirkelKoordinater()
% cirkel koordinater
% Beräknar x och y koordinaterna för
% cirkeln med radie = 1 och origo
% i punkten = (0,0)

% funktionskroppen börjar
fi = 0:0.01:2*pi;
x = cos(fi);
y = sin(fi);
% funktionskroppen slutar
```

Anropet

```
>> [xKoord, yKoord]=cirkelKoordinater;
```

leder till att variablerna $xKoord$ och $yKoord$ i tur och ordning tilldelas de värden som variablerna x och y har inuti funktionen $cirkelKoordinater$.

Varning! Om anropet istället skrivs

```
>> [Koord] = cirkelKoordinater;
```

i tron att man får ut både x- och y-koordinaterna misstar man sig. $Koord$ kommer endast att innehålla x-koordinaterna.

Funktioner som anropar funktioner

Man kan lätt skapa funktioner som anropar funktioner. I funktionen *hyp* nedan beräknas hypotenusan av en rätvinklig triangel med hjälp av *square*. I filen *hyp.m* är funktionen *hyp* definierad

```
function res = hyp(sida1, sida2)
% Hypotenusa
% Beräknar hypotenusan av en
% rätvinklig triangel.

% funktionskroppen börjar
res=sqrt(square(sida1)+square(sida2));
% funktionskroppen slutar
```

hyp anropas med 3 och 4 som är triangelns respektive sidor

```
>> svar = hyp(3,4);
```

När anropet $hyp(3,4)$ görs leder det till att variablerna i funktionen *hyp* tilldelas värden, dvs
$sida1 = 3$ och $sida2 = 4$. Därefter görs två nya anrop; $square(3)$ och $square(4)$. Dessa anrop beräknas till 9 respektive 16 och returneras för att sedan summeras till 25. Roten dras ur summan och slutligen returneras talet 5 till variabeln *svar*.

Funktioner som indata

Man kan även skicka med funktioner som indata till vissa andra funktioner. För att beräkna integralen av en funktion över ett visst intervall kan exemplevis funktionen *quad* användas.

Nedan beräknas integralen av funktionen $f(x) = x^2$ över intervallet $x \in [0, 1]$.

```
>> quad('square',0,1)
ans =
    0.3333
```

Observera att för att *quad* ska fungera måste funktionen *square*:s indata kunna vara en vektor, se även avsnitt 2.6.

Sökord och hjälptext

Kommandona *lookfor*, *help* och *type* kan även fungera för egendefinierade funktioner. Läs mer om dem i avsnitt 1.1.3.

Då man i kommandofönstret anger

```
>> lookfor kvadrat
square.m: % KVADRAT square
```

svarar Matlab med att skriva ut namnen på de funktioner vars H1-rad innehåller sökordet *kvadrat*. Funktionernas respektive H1-rader skrivs också ut.

Använder man däremot kommandot *help* i kommadofönstret skrivs funktionens H1-rad och de efterföljande och sammanhängande kommenterade raderna ut. Denna text är en funktions hjälptext.

```
>> help square
  KVADRAT square
  Beräknar kvadraten för reella tal.
  Indata kan vara skalär eller vektor.

    Reference page in Help browser
       doc square
```

Ytterligare exempel

En funktion behöver varken något indata eller utdata och därför kan *invar* och *utvar* utelämnas. Följande är ett exempel på en funktion som skriver ut snöflingor. Innehållet i filen *skrivAsterisker.m*:

```
function skrivAsterisker
%

disp('***************')
% funktionskroppen slutar
```

Funktionen anropas och snöflingor skrivs ut i kommadofönstret.

```
>> skrivAsterisker
***************
```

3.3.2 Anonym funktion

En anonym funktion kan skrivas i princip var som helst och skrivs på formen
$funktionsnamn = @(invar)funktionskropp$
Tecknet @ talar om att det är en anonym funktion som definieras.

Exempelfunktionen *square*

Här definieras funktionen i kommandofönstret och därefter anropas den.

```
>> square = @(x) x.^2
>> square(2)
ans =
     4
```

Till skillnad mot den matematiska definitionen av *square* kan den Matlab definierade funktionen *square* beräkna kvadraten av varje element i en vektor. Detta är möjligt eftersom punktnotationen .^ har använts istället för ^, för punktnotation se avsnitt 2.3.2. När *square* anropas från kommandofönstret med en vektor returneras en vektor istället. Varje element i vektorn har kvadrerats som syns i exemplet nedan.

```
>> vektor = [1 2 3];
>> svarsVektor = square(vektor)
svarsVektor =
     1     4     9
```

Funktioner som anropar funktioner

Även för anonyma funktioner är det möjligt att skapa funktioner som anropar funktioner. Nedan beräknas återigen hypotenusan och förutom hur funktionen definieras fungerar det som det beskrivs i avsnitt 3.3.1.

```
hyp=@(x,y) sqrt(square(x) + square(y))
```

```
>> svar = hyp(3,4)
svar =
     5
```

Funktioner som indata

Man kan även skicka med funktioner som indata till vissa andra funktioner.

För att beräkna integralen av en funktion över ett visst intervall används t.ex. funktionen *quad*. Funktionens indata måste kunna vara en vektor, se även avsnitt 2.6.

```
>> quad(@(x)square(x),0,1)
ans =
    0.3333
```

Man kan utelämna tilldelningen till variabeln *square*. Då använder man sig direkt av funktionskroppen.

```
>> quad(@(x) x.^2,0,1)
ans =
    0.3333
```

3.3.3 inline

Formen för kunstruktionen är
$funktionsnamn =$
$inline('funktionskropp','invar')$

Exempelfunktionen *square*

```
>> square = inline('x.^2','x');
>> square(3)
ans =
     9
```

För att beräkna hypotenusan definieras

```
>> hyp=inline('sqrt(x^2+y^2)','x','y');
>> hyp(3,4)
ans =
     5
```

Kapitel 4

Grafik

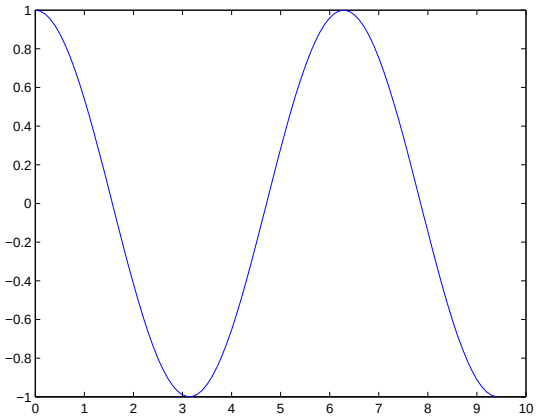
I Matlab finns stora möjligheter att göra grafiska presentationer i 2D såväl som 3D. Här presenteras endast en liten del av det grafiska utbudet. En del av det grafiska utbudet finns via meny-val som till stor del är självförklarande och förklaras därmed inte här.

4.1 Enkel plottning i 2D

För att rita en kurva i 2D kan man använda kommandot *plot* som skrivs på formen $plot(indata)$.

Säg att man vill plotta $y = cos(x)$ där
$x \in [0, 3pi]$. Eftersom en graf är uppbyggd av linjer som går mellan koordinatparen $(x_i, y(x_i))$ och $(x_{i+1}, y(x_{i+1}))$ bör man se till att x är finfördelat för att få en fin graf. I detta exempel sätts $x = 0 : 0.1 : 3 * pi$ och med nedanstående kommandon får man en fin kurva som visas i figur 4.1.

```
>> x = 0:0.1:3*pi;
>> y = cos(x);
>> plot(x,y)
```



Figur 4.1: cos(x)

Plotten kommer att göras i ett nytt fönster om inget tidigare finns. Finns ett fönster kommer plotten att göras i det senast använda och då tas tidigare plottar bort (om inte *hold on* har gjorts som beskrivs i avsnitt 4.2).

Kurvan i figur 4.1 kan även göras med hjälp av *fplot*. Följande kommandon kan då användas

```
>> limit = [0 3*pi];
>> fplot(@(x)cos(x),limit)
```

4.2 Fler grafer i samma fönster

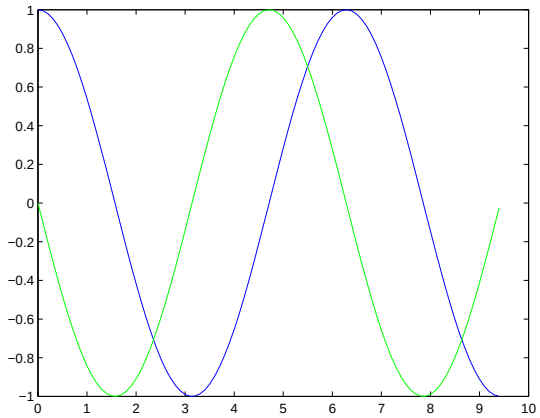
När man t.ex. vill jämföra två funktioner kan det underlätta om man plottar dem i samma fönster. Här följer några exempel på hur man kan plotta funktionerna $y1(x) = cos(x)$ och $y2 = -sin(x)$ i samma fönster.

I det första exemplet används *hold on* för att förhindra att *plot*-kommandot tar bort tidigare plottar. Med kommandot *hold on* kan nya plottar läggas till utan att den gamla tas bort. Plotten visas i figur 4.2.

```
>> x = 0:0.1:3*pi;
>> y = cos(x);
>> plot(x,y)
>> hold on
>> y = -sin(x);
>> plot(x,y,'g')
```

Lägg märke till att variabeln y kan återanvändas. I *plot*-kommandot anges linjefärgen grön, mer om detta i avsnitt 4.8.

OBS! Vill man efter detta plotta i ett rengjort grafik-fönster ger man kommandot *hold off*.



Figur 4.2: $cos(x)$ och $-sin(x)$

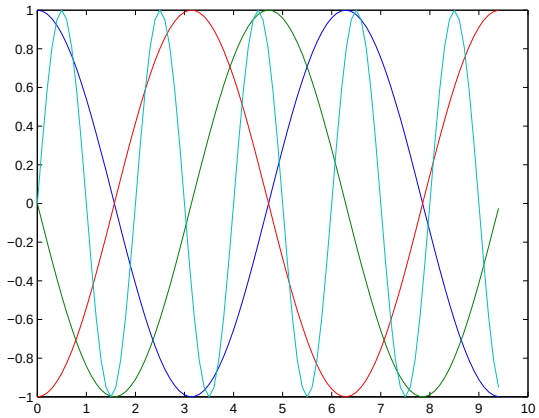
För att plotta samma som ovan kan man utöka *plot*-kommandot med fler indata. Fördelen med detta är att kurvorna automatiskt får olika färger och därför kan skiljas ifrån varandra. Nackdelen är att man måste använda fler variabelnamn.

```
>> x = 0:0.1:3*pi;
>> y1 = cos(x);
>> y2 = -sin(x);
>> plot(x,y1,x,y2)
```

I det tredje exemplet elimineras nackdelen ovan genom att lägga funktionerna i en vektor. Antalet funktioner som plottas utökas till fyra stycken. Plotten visas i figur 4.3.

```
>> x = [0:0.1:3*pi]';
>> Y = [cos(x) -sin(x) -cos(x) sin(pi*x)];
>> % Utöka x-vektorn
>> X = x(:,[1 1 1 1]);
>> plot(X,Y)
```

Här utökas x-vektorn men det kan också göras med hjälp av en *for*-slinga, se avsnitt 4.8.



Figur 4.3: Fyra olika funktioner.

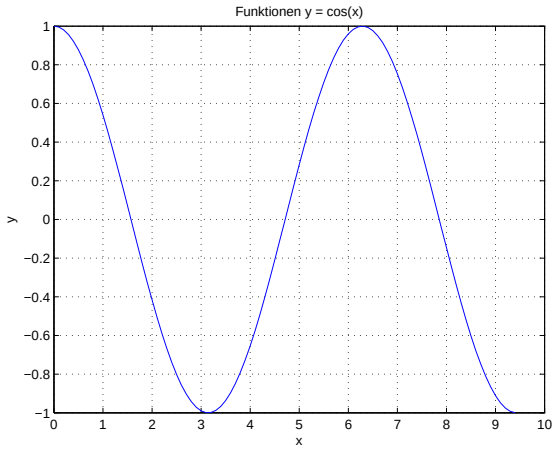
4.3 grid

För att få ett rutnät i grafen ges kommandot *grid*, se exempel i avsnittet 4.4.

4.4 Titel och axeletiketter

När en plott har gjorts kan man ge den en titel med $title('text')$ och märka koordinaternas axlar med lämplig text. x-axeln märks med kommandot $xlabel('xtext')$ och på liknande vis märks y och z-axlarna. Resultatet av följande exempel visas i figur 4.4.

```
>> x = 0:0.1:3*pi;
>> y = cos(x);
>> plot(x,y)
>> grid
>> title('Funktionen y = cos(x)')
>> xlabel('x')
>> ylabel('y')
```



Figur 4.4: Titel, axeletiketter och grid

4.5 Axlar

Om inget anges vid plottningen skalar Matlab axlarna automatiskt. För att själv skala om axlarna finns bland annat dessa fyra kommandon

- *axis*([*xmin* *xmax* *ymin* *ymax*])
  x-axeln begränsas till [*xmin*, *xmax*]
  y-axeln begränsas till [*ymin*, *ymax*]

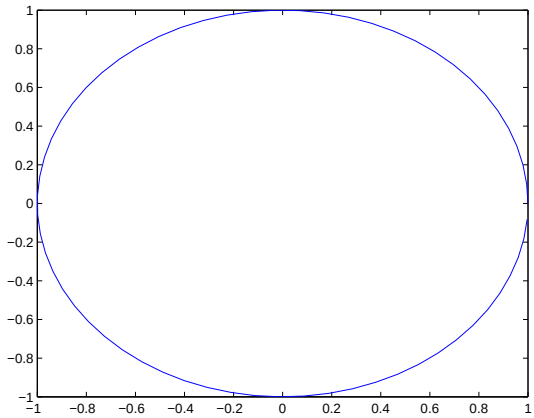
- *axis auto*
  Sätter tillbaka axlarna till default

- *axis square*
  Sätter den aktuella plotten till att vara en kvadrat istället för default rektangeln.

- *axis equal*
  Skalningsfaktorerna för båda axlarna sätts lika.

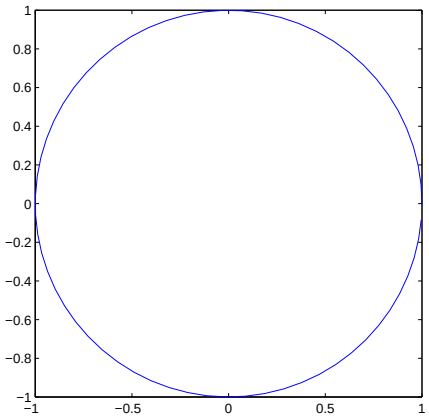
Det kan ibland vara bättre att skala om axlarna. Ett exempel är då man vill plotta enhetscirkeln. Följande görs där Matlabs egen skalning används

```
>> fi = 0:0.1:2*pi;
>> x = cos(fi);
>> y = sin(fi);
>> plot(x,y)
```



Figur 4.5: Enhetscirkeln!

Plotten i figur 4.5 ser ut som en oval figur trots att det är enhetscirkeln som är beräknad! Det förklaras med att plotten görs i ett rektangulärt område (för figuren). För att ändra och åtgärda detta anges *axis square* eller *axis equal* och plotten visar en rund cirkel i figur 4.6.

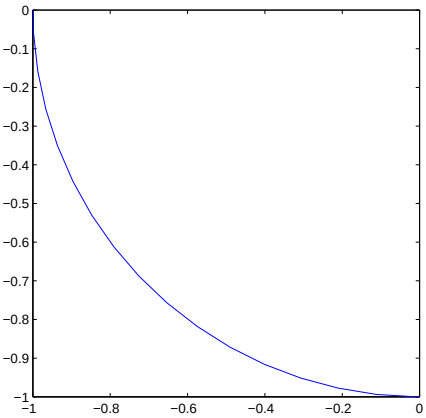


Figur 4.6: En rund enhetscirkel.

För att endast visa den nedre vänstra fjärdedelen av cirkeln som visas i figur 4.7 ges kommandot

```
>> axis([-1 0 -1 0])
```

För att återgå till defaultinställningarna anges *axis auto*



Figur 4.7: Nedre vänstra fjärdedelen av enhetscirkeln.

4.6 Läsa in koordinater ur grafen

Ur en graf kan det ibland vara bekvämt att med hjälp av musen ange intressanta punkter. *ginput* läser in de punkter som man klickar på med musen.

Följande exempel läser in tre punkter via musklick.

```
>> x = 0:0.1:3*pi;
>> y = cos(x);
>> plot(x,y)
>> grid on
>> antalPunkter = 3;
>> [X, Y] = ginput(antalPunkter);
```

De tre punkternas koordinater tilldelas vektorerna X och Y.

Om man utelämnar argumentet *antalPunkter* till *ginput* avslutas inläsningen när man trycker på *enter*- knappen. Anropet ser då ut

```
>> [X, Y] = ginput;
```

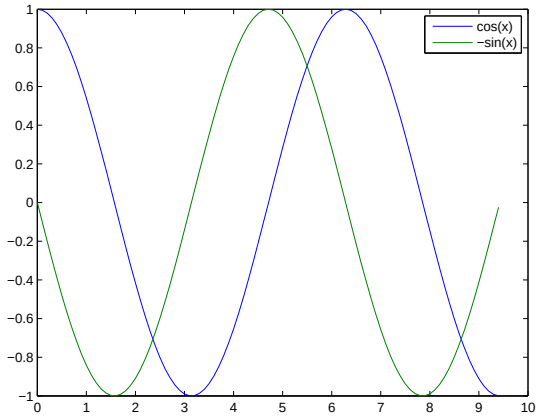
4.7 Hjälptext i grafen

För att lägga in förklarande text till en kurva i fönstret finns flera möjligheter. Här visas *legend*, *text* och *gtext*.

För att få en faktaruta i grafen ges kommandot $legend('text'_1, \ldots, 'text'_n)$. Faktarutan innehåller $text_1$ till första kurvan, $text_2$ till andra kurvan osv. Resultatet av följande exempel visas i figur 4.8

```
>> x = 0:0.1:3*pi;
```

```
>> y1 = cos(x);
>> y2 = -sin(x);
>> plot(x,y1,x,y2)
>> legend('cos(x)','-sin(x)')
```

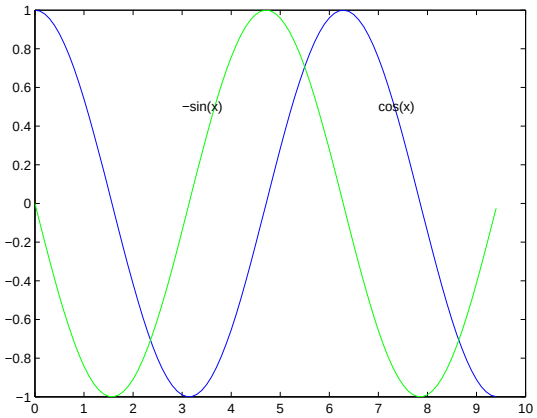


Figur 4.8: Faktaruta

Faktarutan kan nu flyttas med musen.

Man kan även placera ut text i grafen genom att ange koordinaterna eller klicka med musen. Nedan följer exempel på respektive förfarande och ett plottresultat visas i figur 4.9

```
>> % ange koordinaterna
>> x = 0:0.1:3*pi;
>> y = cos(x);
>> box on
>> plot(x,y)
>> text(7, 0.5, 'cos(x)')
>> hold on
>> y = -sin(x);
>> plot(x,y,'g')
>> text(3, 0.5, '-sin(x)')

>> % med musklick
>> x = 0:0.1:3*pi;
>> y1 = cos(x);
>> y2 = -sin(x);
>> plot(x,y1,x,y2)
>> gtext('cos(x)')
>> gtext('-sin(x)')
```



Figur 4.9: Textplacering

4.8 Linjefärger, linjetyper och punktmarkörer

För varje kurva som plottas kan man välja vilken färg och vilket utseende den ska ha. Detta skrivs till *plot*-kommandot. För att plotta en svart streckad linje av värdena x och y skriver man $plot(x, y,' k :')$

Färgerna är bland annat

```
b - blå
g - grön
r - röd
c - cyan
m - magenta
y - gul
k - svart
w - vit
```

Linjetyperna är

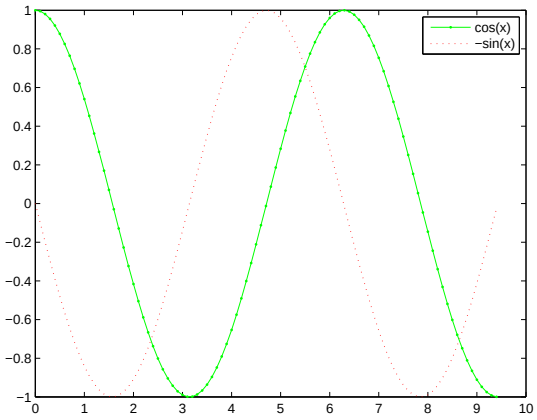
```
-  - heldragen
:  - prickad
-. - prick-streckad
-- - streckad
```

Punktmarkörerna är bland annat

```
. - punkt
o - cirkel
x - kryss
+ - plus
* - asterisk, snöflinga
```

I nedanstående exempel plottas två kurvor i figur 4.10. Den första kurvan blir grön prickstreckad och den andra rödprickad.

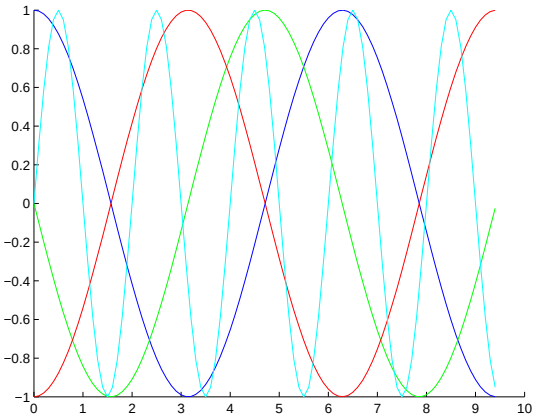
```
>> x = 0:0.1:3*pi;
>> y1 = cos(x);
>> y2 = -sin(x);
>> plot(x,y1,'g-.',x,y2,'r:')
>> legend('cos(x)','-sin(x)')
```



Figur 4.10: Två olika linjetyper och färger.

I följande exempel plottas fyra kurvor med fyra olika färger i figur 4.11. Här används vektorn *color* för att ange de olika färgerna som kurvorna plottas med.

```
>> x = [0:0.1:3*pi]';
>> Y = [cos(x) -sin(x) -cos(x) sin(pi*x)];
>> color = ['b','g','r','c'];
>> hold on
>> for i = 1:4
      y = Y(:,i);
      plot(x,y,color(i))
>> end
```



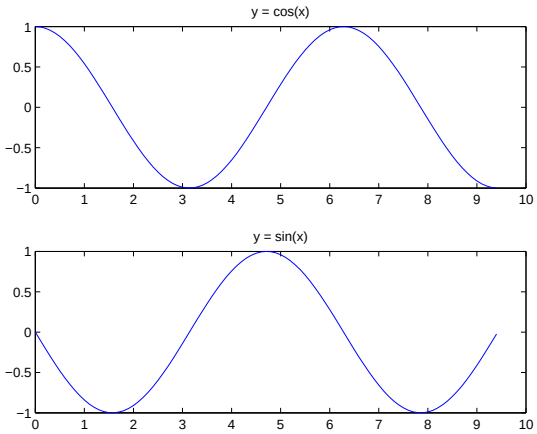
Figur 4.11: Fyra olika funktioner och färger.

4.9 Separata plottar i samma fönster

Det finns möjlighet att göra flera separata plottar i samma fönster med hjälp av *subplot*. Kommandot skrivs på formen $subplot(rader, kolumner, aktuell)$. *subplot* delar upp ett fönster i en matris med grafikfönster. Matrisens storlek är $rader \times kolumner$. *aktuell* anger i vilket grafikfönster som aktuell plott ska göras. Grafikfönstren numreras radvis från vänster till höger med den översta raden först därefter den nästöversta osv. Tänk på att *subplot* inte plottar utan bara delar upp ett fönster och att man därför måste skriva ett plottkommando efter *subplot*!

I följande exempel visas hur kurvorna
$y = cos(x)$ och $y = -sin(x)$ plottas i två rader i figur 4.12

```
>> x = 0:0.1:3*pi;
>> y = cos(x);
>> subplot(2,1,1)
>> plot(x,y)
>> title('y = cos(x)')
>> y = -sin(x);
>> subplot(2,1,2)
>> plot(x,y)
>> title('y = sin(x)')
```

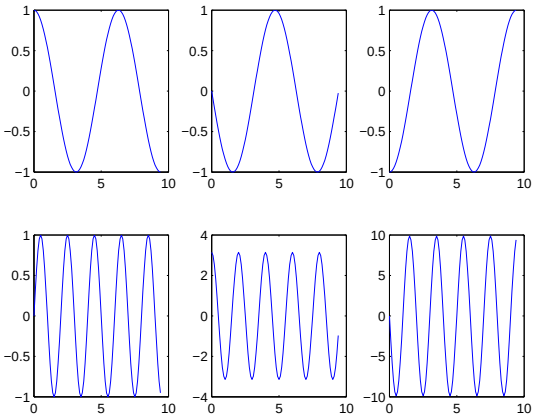


Figur 4.12:

Efter att dessa satser har körts är det aktuella fönstret det nedersta, dvs på rad 2.

För att plotta 6 funktioner separat används här både vektorer, *for*-slinga och *subplot*. Resultatet visas i figur 4.13

```
>> x = [0:0.1:3*pi]';
>> Y = [cos(x) -sin(x) -cos(x) ...
 sin(pi*x) pi*cos(pi*x) -pi^2*sin(pi*x)];
>> for i = 1:6
       y = Y(:,i);
       subplot(2,3,i)
       plot(x,y)
>> end
```



Figur 4.13: Sex olika funktioner.

Det aktuella fönstret är nu det nedersta längst till höger, dvs på rad 2 i kolumn 3.

4.10 Figurfönster

När ett plottkommando körs öppnas ett nytt figurfönster automatiskt om det inte redan finns ett. Eftersom varje fönster numreras kan man komma åt ett fönster via dess ordningsnummer. Man skriver då $figure(ordningsnummer)$.

För att öppna ett nytt figurfönster kan man skriva $figure$.

4.11 Stänga figurfönster

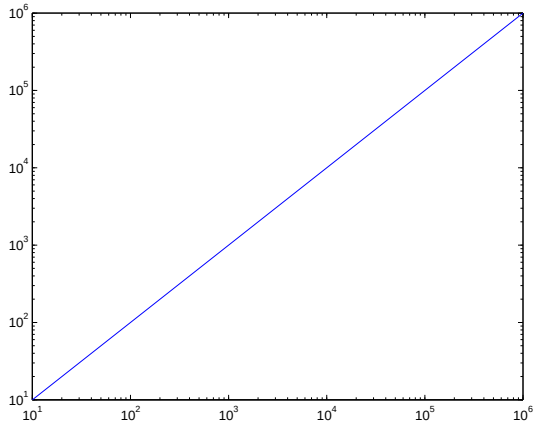
close och clf stänger det aktiva figurfönstret. *close all* stänger alla figurfönster.

4.12 loglog-plot

För att plotta en kurva i ett logaritmiskt fönster används kommandot *loglog*. Den logaritmiska skalningen används för både x- och y-axeln.

Exempel:

```
>> potens = [1:6];
>> x = 10.^potens;
>> y = logspace(1,6,length(potens));
>> loglog(x,y)
```



Figur 4.14: loglog-plot

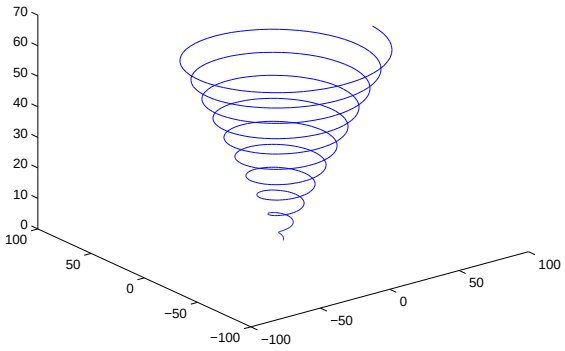
Vektorerna x och y innehåller exakt likadana element men har skapats på två olika sätt.

Det finns även *semilogx* och *semilogy*.

4.13 Plotta en linje i 3D

När man vill plotta en linje i det 3-dimensionella rummet kan man använda sig av *plot*3. För att plotta spiralen i figur 4.15 kan man skriva

```
>> fi = 0:0.1:20*pi;
>> x = fi.*cos(fi);
>> y = fi.*sin(fi);
>> z = fi;
>> plot3(x,y,z)
```

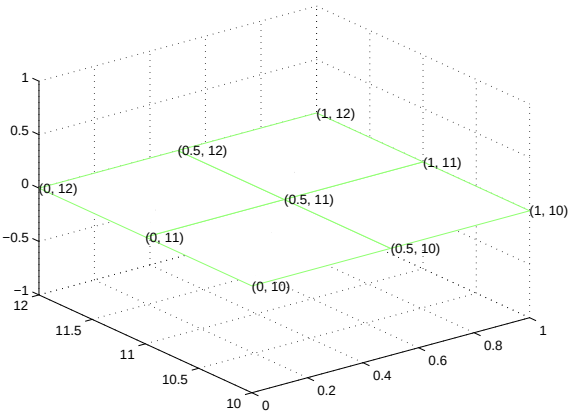


Figur 4.15: Spiral

4.14 Plotta en yta i 3D

För att plotta en funktion $z = f(x_i, y_j)$ där $x_i \in [xmin \quad .. \quad xmax]$ och $y_j \in [ymin \quad .. \quad ymax]$ i ett 3-dimensionellt rum kan man använda t.ex. *mesh* eller *contour*. Tänk på att det till varje koordinatpar (x_i, y_j) finns ett värde $z = f(x_i, y_j)$ och att det leder till att en yta plottas. Det innebär dessutom att för varje x_i så finns alla y-värden och för varje y_j så finns alla x-värden. På grund av detta konstruerar man därför matriser av x- respektive y-värdena. Kommandot *meshgrid* används här med fördel.

Säg att man vill plotta en yta, $f(x, y) = 0$ i $x \in [0, 1]$ och $y \in [10, 12]$ där koordinatparen är som i figur 4.16.



Figur 4.16: meshgrid

För att få koordinatparen används *meshgrid* där indata är x-vektorn $[0, 0.5, 1]$ samt y-vektorn $[10, 11, 12]$

```
>> x = 0:0.5:1;
>> y = 10:12;

>> [X,Y] = meshgrid(x,y);
```

Koordinatparen återfinns nu i matriserna X och Y.

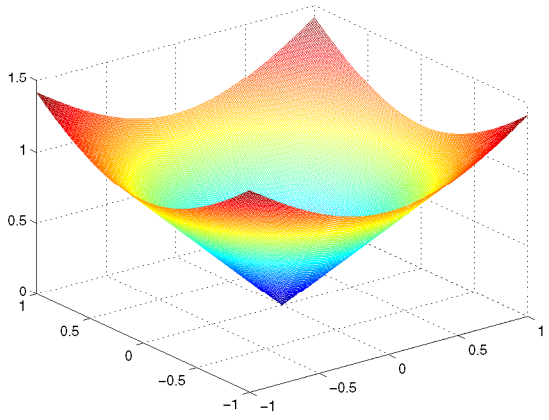
```
>> X
X =
    0    0.5000    1.0000
    0    0.5000    1.0000
    0    0.5000    1.0000
>> Y
Y =
    10    10    10
    11    11    11
    12    12    12
```

För att få den önskade ytan, som visas i figur 4.16 skrivs

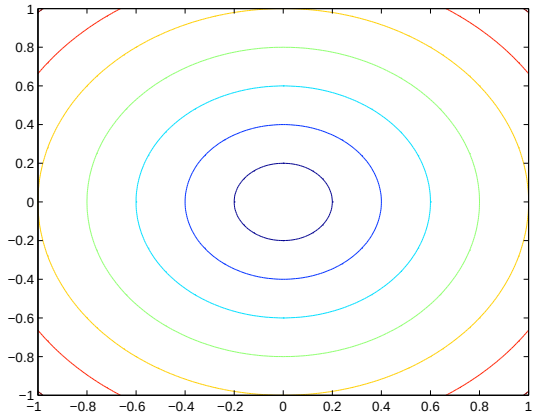
```
>> Z = zeros(size(X));
>> mesh(X,Y,Z)
```

Ett ytterligare exempel är när man ska plotta en konisk yta. Här visas plottar med kommandona *mesh* och *contour* i figur 4.17 respektive 4.18.

```
>> x = -1:0.01:1;
>> y = x;
>> [X, Y] = meshgrid(x,y);
>> % konisk yta
>> Z = sqrt(X.^2 + Y.^2);

>> mesh(X,Y,Z)
>> contour(X,Y,Z)
```



Figur 4.17: Konisk yta med *mesh*



Figur 4.18: Konisk yta med *contour*

Sakregister