Föreläsningsanteckningar för Programmering med MATLAB MVG300

Thomas Ericsson
Beräkningsmatematik
Chalmers/GU
2011

Innehåll

1 Introduktion

1.1 Vad är programmering och varför är det svårt?

Programmering består till stor del av problemlösning, man vill hitta en metod, ett recept för att lösa ett speciellt problem.
Problemet "gör en sockerkaka" kan t.ex. lösas så här
(från `recept.nu`, jag har inte testat receptet):

Ingredienser
2 ägg
2 dl socker
3 dl vetemjöl
1 1/2 tsk bakpulver
1/2 tsk vaniljsocker
50 g smör
1 dl mjölk
matfett och ströbröd till formen

Gör så här

1. Sätt ugnen på 175 grader. Smörj och bröa sockerkaksformen. Vispa ägg och strösocker vitt och pösigt. Blanda samman mjöl, bakpulver och vaniljsocker.
2. Smält smöret och häll i mjölken, så att allt blir ljummet. Häll mjölkblandningen i socker blandningen och rör om. Vispa ner mjölblandningen i smeten och häll den i formen.
3. Grädda kakan i 175 graders ugn i 28-30 minuter. Känn med en sticka om den är klar. Är stickan helt torr är kakan klar.

Kocken läser och utför instruktionerna i receptet.

I programmerings-sammanhang talar man inte om recept utan om algoritmer. En algoritm är en problemlösningsmetod.
En implementation av en algoritm är ett program.
Implementera betyder förverkliga, realisera.
En dator (motsvarar kocken) kör, exekverar programmet.

Lösningen av typtal är väldigt algoritmiskt, man behöver knappast kunna någon matematik, så låt oss studera ett typtal från min gymnasietid.

Finn alla rötter till:

$$x^3 - 8x^2 + 23x - 28 = 0$$

Det finns formler som ger rötterna till allmänna tredjegradsekvationer, men dessa lärde jag mig ej i gymnasiet (lärs inte ut vid GU heller). Så tricket för detta typtal är att vi vet att det finns en heltalsrot. Alla problem i denna typtalsmängd är konstruerade på detta sätt.

Eftersom polynomet har heltalskoefficienter måste heltalsroten dela den konstanta termen. Vi får alltså undersöka alla delare till -28, dvs. talen $\pm 1, \pm 2, \pm 4, \pm 7, \pm 14, \pm 28$ och det visar sig att 4 är en rot. Vi dividerar sedan polynomet med $x - 4$ och löser den resulterande andragradsekvationen.

Ovanstående är en kortfattad algoritmskiss skriven i naturligt språk (svenska). Ofta gör man skisserna i sk pseudokod, en blandning mellan engelska och programspråk.

För att bestämma delare till den konstanta termen kan vi dela denna med talen $1, 2, \ldots, 28$ och se vilken rest vi får. $28 = 3 \cdot 9 + 1$, resten ett, så 3 är inte en delare. (Detta är inte den mest effektiva algoritmen, men den duger för enkla problem.) Detta är exempel på en algoritm som utgör en del av den större algoritmen. Ofta bryter man ned en lösning i mindre delar på detta sätt, kallas "top-down design". Att lösa andragradsekvationen kräver ytterligare en liten algoritm.

I denna kurs är algoritmerna inte givna, att komma på dessa är den svåra biten. Att implementera en algoritm är enklare.
Nu när vi har en algoritmskiss kan man skriva ett kort MATLAB-program (inte inkluderat), som löser problemet.

Så här ser det ut när programmet körs, exekveras:

```
>> [r1, r2, r3] = tredjegrads_ekv(-8, 23, -28)
r1 = 4
r2 = 2 + sqrt(3)i
r3 = 2 - sqrt(3)i
```

Observera att programmet inte kan matematik och det kan inte datorn heller. Datorn förstår inte vad programmeraren vill, utan datorn gör som den är tillsagd. Datorn gör inte heller några rimlighetsbedömningar. Om det i sockerkaksreceptet hade stått 2000 ägg, så hade sannolikt kocken reagerat. En dator reagerar inte.
Det är programmeraren som måste kunna matematik.

I grundkurser i matematik finns standardproblem, typtal, där man kan lära sig algoritmer (beräkna determinanter, lösa linjära ekvationssystem, deriveringsregler etc). Man kan lösa dessa typtal om man lärt sig algoritmerna.

Problemen i högre matematik-kurser utgörs mest av "visa-problem". Bevisa att ...
Detta är sällan typtal och det finns ingen standardmetod för att bevisa satserna. En lärare på en sådan kurs tar kanske upp olika bevistekniker som t.ex. induktionsbevis, konstruktiva bevis och motsägelsebevis. Man får alltså lära sig att hantera diverse verktyg. Läraren torde också ta upp vanliga fallgropar, man måste skilja mellan implikation och ekvivalens, tillräckliga- och nödvändiga villkor etc.

Att ta fram en algoritm kan ofta liknas med att ta fram ett konstruktivt bevis för sats som man inte har sett tidigare.
Som programmerare måste man kunna ta sig an nästan vilket problem som helst. Man har liten användning av att bara kunna lösa en liten uppsättning standardproblem. Däremot är det bra att kunna lösa standardproblem, eftersom man då har fler verktyg att arbeta med.

Det riktigt svåra är som sagt att hitta en algoritm. När man är nybörjare kommer själva programspråket också att ställa till bekymmer, ungefär som när man lär sig ett främmande språk som engelska eller tyska. Ett programspråk är dock mycket enklare än ett naturligt språk. Om man har lärt sig programmera med ett språk brukar det inte vara så svårt att lära sig ett annat programspråk.

Undervisningen i denna kurs kommer att bestå av en beskrivning av verktygen (for- och if-satser t.ex) och många små exempel som visar hur dessa används. En del av dessa exempel är standardproblem. Jag kommer att varna för vanliga fallgropar.

Det kommer exempel på hur man kan skapa en algoritm, tyvärr går inte detta exempel att generalisera till alla problem. En vanlig metod när man är nybörjare är att fundera över hur man skulle lösa problemet för hand. När man har fått fram en fungerande algoritm, och först då, kan man implementera den som ett MATLAB-program.

Vi kommer också att titta på avlusning, tekniker för att hitta löss, buggar i ett program. Det finns syntaxfel (språkfel), och logiska fel, algoritmen är felaktig. Syntaxfelen brukar vara enkla att åtgärda. Man måste kunna hitta fel på egen hand, annars kan man inte programmera.

Den svåraste delen i programmering är att skapa algoritmen och det kan vara riktigt svårt, därefter kommer avlusningen. Att skriva MATLAB-koden brukar vara det enklaste (åtminstone i denna kurs).

Att döma av senaste kursutvärderingen så förväntade sig några kursdeltagare att bli lärda ett recept för att lösa programmeringsproblem, men det finns tyvärr inget. Detta är ingen typtalskurs.

1.2 Två goda råd

En typisk lab:
"skriv ett MATLAB-program som löser ovanstående problem".

Tre steg:

1. Läs igenom lab-PM och förstå problemet.
2. Tänk ut en algoritm. Använd papper och penna, rita bilder, fundera över hur du skulle göra för hand.
3. Implementera algoritmen, dvs. skriva MATLAB-koden.

Att lösa problemet för hand.
Ett exempel: hitta minsta talet i en vektor med heltal.

Titta på ett *litet* problem. Problemet skall dock vara generellt, så att vi inte hittar en algoritm för ett specialfall. I följande exempel har vi blandade tecken samt upprepningar.

```
5 5 7 -4 6 -10 11 -10 0 13 5
```

"Fuska" inte, en människa ser direkt att minsta talet är -10, en dator gör det inte. Vi måste hitta en systematisk procedur som fungerar för ett godtyckligt antal tal, t.ex. 1 000 000 tal.

Idé: vi går igenom talen från vänster till höger och håller reda på det hittills minsta talet. Vi måste också hålla reda på det aktuella talet när vi går igenom vektorn. Om det aktuella talet är mindre än det hittills minsta talet, uppdaterar vi detta.

När man skall skriva ett program måste först man behärska verktygen, loopar, if-satser etc. Det gör man genom att träna på enkla problem (mina övningar eller problem man själv hittar på).

1.3 Varför MATLAB?

Det finns tusentals programspråk. Varför MATLAB?

- Enkelt att komma igång.
- Lättanvänd grafik, enkelt att skapa GUIs.
  (GUI = Graphical User Interface, grafiskt användargränssnitt).
- Kraftfullt för numeriska beräkningar (vektorer, matriser).
- Används på många GU/Chalmers-kurser och till viss del i näringslivet.
- Önskemål vid kursutvärderingar.

En nackdel är att MATLAB inte har ett så välutvecklat typsystem. För att lära sig en del om datatyper (viktigt om man skall fortsätta med Java t.ex) innehåller 7.5 hp-kursen en gnutta C++.

C++ är ju eftertraktat när man söker arbete, så varför inte en kurs med enbart C++?

- Komplicerat och stort.
- Inte förlåtande, lätt att göra fel.
- Besvärligt med grafik och GUI.
- Java (liknar C++) kommer andra året på matematikerlinjen.

En arbetsgivare kräver nog att man behärskar objektorienterad programmering (OOP), mer om detta senare. Att lära sig allt detta klarar man knappast på en nybörjarkurs. Dessutom är MATLAB intressantare ur beräkningssynpunkt.

Det viktigaste och svåraste i en programmeringskurs är att lära sig problemlösningsmetodik. Att lära sig programspråkets grammatik utgör normalt en mindre och enklare del.
Om man har lärt sig programmera med ett språk brukar det inte vara så svårt att lära sig ett annat programspråk.

1.4 Exempel på några programspråk

Vissa språk passar bättre för vissa arbetsuppgifter än andra. MATLAB är t.ex. bekvämt när det gäller matrishantering, men det vore svårt (omöjligt) att skriva ett operativsystem, som Linux, i MATLAB. Man väljer då något annat språk (Linux är skrivet i C).

För att visa på likheter mellan olika språk följer här några kodexempel som beräknar en approximation till

$$\sum_{k=1}^{1000} \frac{1}{k}$$

Först Java-script, ett språk som kan användas i web-sidor. Lägg följande rader på en web-sida, **summa.html**.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head>
    <title>Summa-exempel</title>
  </head>
  <body>
    <script type="text/javascript">
      var s = 0.0;
      for(k = 1; k <= 1000; k++)
        s += 1.0 / k;
      document.write('1/1 + 1/2 + ... + 1/1000 = ' + s);
    </script>
  </body>
</html>
```

När man läser sidan (i Firefox) kommer följande text upp:

```
1/1 + 1/2 + ... + 1/1000 = 7.485470860550343
```


Python

```
>>> s = 0
>>> for k in range(1, 1001):
...   s += 1.0 / k
...
>>> s
7.4854708605503433
```

Tcl

```
tcl>set s 0
tcl>for {set k 1} {$k <= 1000} {incr k} \
        {set s [expr $s+1.0/$k]}
tcl>puts $s
7.48547086047
```

Lisp (kan man köra i emacs till exempel):

```
(setq sum 0)
(setq k 1)
(while (<= k 1000)
    (setq sum (+ sum (/ 1.0 k)))
    (setq k (+ k 1))
)
(eval sum)
```

Fortran90

```
program sum_ex
  integer          ::  k
  double precision ::  summa

  summa = 0.0
  do k = 1, 1000
    summa = summa + 1.0d0 / k
  end do

  print*, "1 + 1/2 + ... + 1/1000 = ", summa
end program sum_ex
```


Java

```
public class Summa {
  public static void main (String args[]) {
    double summa;

    summa = 0;
    for(int k = 1; k <= 1000; k++)
      summa += 1.0 / k;

    System.out.println("1 + 1/2 + ... + 1/1000 = "
                       + summa);
  }
}
```

Nu några mer ovanliga exempel.
PostScript, som ju i första hand ett språk för att generera text och bilder. När man skriver något på våra skrivare förpackas text och bild i ett PostScript-program som sedan exekveras av skrivaren.

```
%!
/s 0 def
/k 0 def

1000 {
  /k 1 k add def
  /s 1 k div s add def
} repeat
s ==
quit
```

eller kortare

```
%!
0 0
1000 {
  1 add dup 1 exch div 3 -1 roll add exch}
repeat
pop
==
quit
```

Om man arbetar lite mer kan få följande resultat:

Maple är ett exempel på ett symbolbehandlande program. Mathematica är ett annat. Maple kan anropas från MATLAB.

```
> s := sum(1 / k, k = 1..1000);
           s := Psi(1001) + gamma

> evalf(s, 100);
7.485470860550344912656518204333900176521679169708800
  366577362674995769934916520244095993443741184508 2

> expand(s);
533629132822947850455910456240429804096524722803842 6
009710134924845626888949710175750609790198503569140 9
088731550468098378442172117885009464302344326566022 5
021002784256328520814055449412104425101426727702947 7
471270891796396777961045322469242686646888828158207 1
984897105110796873249319155529397017508931564519976 0
857344730141832840117244122806490743077037366831700 5
580029365923508858936023528585280816075957473783665 5
413175508131522517   /
712886527466509305316638415571427292066835886188589 3
```



```
040452001991154324087581111499476444151913871586911 7
178170195752565129802640676210092514658710043051310 7
268626814320019660997486274593718834370501543445252 3
739745298963145674982128236956232823794011068809262 3
177088619795407912477545580493264757378299233527517 9
673524804246363805113703433121478174685087845348567 8
021888075373249921995672056932029099390891687487672 6
97950931603520000
```

I MATLAB kan man skriva:

```
>> s = sym(0)
s = 0

>> for k = 1:1000, s=s+1/k; end
>> s
s = 533629132822947850455910 45 osv.
```

Haskell, som denna kurs ersätter (för matematikerna).

```
Hugs> sum [ 1.0 / k | k <- [1..1000]]
7.48547086055034
```

J en efterföljare till APL

```
+/ % 1000 - i. 1000
```

I denna kurs kommer vi att studera MATLAB i detalj:

```
>> s = 0;
>> for k = 1:1000
     s = s + 1 / k;
   end
>> s
s =
   7.4855e+00
```

eller kortare

```
>> sum(1 ./ (1:1000))
ans = 7.4855e+00
```


Flertalet programspråk har inte så kraftfull hantering av matriser och vektorer som MATLAB har. Därför måste man behärska båda programmeringsstilarna. Vi kommer att börja med programmering på elementnivå.

I slutet av MVG300-kursen kommer *lite* C++:

```
#include <iostream>
using namespace std;

int main()
{
  double summa;

  summa = 0.0;
  for(int k = 1; k <= 1000; k++)
    summa += 1.0 / k;

  cout << "1 + 1/2 + ... + 1/1000 = " << summa << endl;

  return 0;
}
```

```
% g++ sum.cc
% a.out
1 + 1/2 + ... + 1/1000 = 7.48547
```

Den stora finessen med C++ är att det stödjer objekt-orienterad programmering (kommer vi inte att ta upp).

Några ord om OOP:

C++ har stöd för komplexa tal i sitt standardbibliotek, men antag att så inte vore fallet (Java har det inte t.ex). Vi skulle då kunna skapa en så kallad klass i vilken vi anger hur ett komplext tal skall lagras. Två uppenbara alternativ är att lagra real- och imaginärdel. Ett annat är att lagra r och φ om vi skriver talet som $re^{i\varphi}$.

Användaren av klassen behöver inte (och skall inte) veta hur talet lagras. Informationen döljs, man slipper bry sig om alla detaljer.

För att kunna räkna med talen krävs operationer som +, -, *, /, cos, sin, ... Klassen innehåller medlemsfunktioner (eller metoder, som det heter i Java) som utför sådana beräkningar. En klass är således ett sätt att paketera data och funktioner som opererar på dessa data.

En klass är en datatyp och en variabel av denna typ, ett komplext tal, är ett objekt. I C++ kan man skapa sin klass så att det är möjligt att t.ex. skriva `z = w + 2 * cos(x)` (där `x`, `z` och `w` är komplexa tal, objekt). Notera att `+`, `*` och `cos` även finns för reella tal. Man talar om polymorfism när samma funktionsnamn kan användas för olika typer.

Säg nu att vi vill arbeta med komplexa matriser. Vi kan då skapa en matrisklass som utnyttjar komplex-klassen vi redan skapat. Att en klass bygger på en annan klass kallar man arv. En viktig del av OOP är lära sig hur man bryter ner ett problem i klasshierarkier.

2 Variabler och uttryck

2.1 Variabler

Man vill kunna referera till ett minnesutrymme via ett namn, att använda adresser är för opraktiskt.
Variabelnamn får bestå av bokstäver, siffror och understreck, _ (den benämning som Svenska datatermgruppen, **http://www.nada.kth.se/dataterm**, föreslår).

Första tecknet måste vara en bokstav. Antalet tecken får högst vara 63. Längre namn trunkeras.

```
>> namelengthmax
ans =
    63
```

Detta är knappast en begränsning i praktiken. För långa namn ger oläsliga program.

MATLAB skiljer mellan versaler och gemena i variabelnamnen (eng. "case sensitive"). Detta kan man utnyttja för att ge ökad läsbarhet. Använd variabelnamn som säger något (eng. "mnemonic").

Exempel. Några tillåtna variabelnamn:
DelSumma, **dx_dy**, **n_fakultet**, **ute_temp**, **UteTemp**.
Otillåtna är **_var**, **3-dje-fallet**.
■
Man programmerar ofta på engelska, och flertalet exempel i denna kurs kommer att vara på engelska. Inte utan anledning:

```
>> poäng = 10
??? poäng = 10
      |
Error: The input character is not valid in MATLAB
       statements or expressions.
```

poang eller **poaeng** låter inte så bra.

Mina exempel kommer inte alltid att följa dessa regler, eftersom långa namn tar så stor plats på en OH-sida.
För att minska pappersåtgången kommer jag att redigera MATLABs utmatning. Följande sekvens

```
>> x = 1
x =
     1
```

kommer jag normalt att skriva:

```
>> x = 1
x = 1      % så på samma rad
```

I MATLAB skapas en variabel automatiskt vid tilldelning.
Om variabeln redan existerar så skrivs det gamla värdet över.
% inleder en kommentar.

```
>> x = -12.434   % tilldelning
x = -12.4340

>> x             % skriv ut värdet
x = -12.4340

>> x = 23        % ersätt det gamla
x = 23

>> y             % odefinierad
??? Undefined function or variable 'y'.
```

Blanktecken (mellanslag), eller slarvigt blanka, är ej signifikanta ovan, utan används för ökad läsbarhet. **x=-12.434** är tillåtet.

-12.434 är exempel på en flyttalskonstant. Här några exempel:

```
>> +12.34      % + tillåtet men onödigt
ans = 12.3400  % answer-variabeln

>> -0.000001
ans = -1.0000e-06
```



```
>> -1e-6       % kortare
ans = -1.0000e-06

>> e-6         % variabeln e minus 6
??? Undefined function or variable 'e'.

>> 23.48e156
ans = 2.3480e+157  % normaliserad form

>> 2,34           % , separerar uttryck
ans = 2
ans = 34
```

Om man inte lagrar resultat av en beräkning tilldelas värdet till **ans** (för answer).

2.2 Aritmetiska uttryck

Vi vill kunna kombinera konstanter och variabler med hjälp av aritmetiska operatorer. De vanligaste är **+**, **-**, *****, **/** och **^** .
+ och **-** förekommer i två situationer, som *unär* eller *binär* operator (eng. unary, binary).

Unär form: **-23.46**, **+34**, **-temp**.
Binär form: **23 + 45.67**, **x + y**, **x - 3.66**.

Operatorer har olika prioritet (eng. priority, precedence). Man kan ändra evalueringsordningen genom att använda parenteser. Här en lista från högsta till lägsta prioritet:

1. Parenteser **( )**
2. Unärt **+** och **-**
3. **^** (upphöjt till)
4. ***** och **/**
5. Binärt **+** och **-**

När två operatorer har samma prioritet, evalueras uttrycket från vänster till höger.

Här några uttryck. Siffrorna under operatorerna indikerar evalueringsordningen.

```
a + b * c    (a + b) * c    a / b / c    a / b * c
  2   1        1    2        1   2        1   2

a / (b / c)    -a + b        a^b^c       a^(b^c)
  2    1       1  2          1 2          2  1

a + b * c - d    a + (b * c) - d
  2   1   3        2    1    3    så () behövs ej

(a + b) * (c - d)     2 * (a + b) / c * d^3
   1    3    2          3    1    4   5  2

((a + b) * (c - d))^3
    1    3    2    4
```

Sätt inte ut onödiga parenteser. Koden blir svårare att läsa och det är lätt att skriva fel. Skriv

```
c0 + x * (c1 + x * (c2 + x * (c3 + x * (c4 + x))))
```

och inte t.ex.

```
c0+(x*(c1+(x*(c2+(x*(c3+(x*(c4+x))))))))

>> c0+(x*(c1+(x*(c2+(x*(c3+(x*(c4+x))))))))
??? c0+(x*(c1+(x*(c2+(x*(c3+(x*(c4+x))))))))
                                          |
Error: Expression or statement is incorrect--possibly
       unbalanced (, {, or [.
```

I en lab (i numerisk analys) ville man beräkna:

```
f = sqrt(1 + (1.5 * (1 - y^3)^(-2/3) * y)^2)
```

Här är några varianter:

```
f = sqrt((1 + (1.5 * ((1 - y^3)^(-2/3))) * y)^2)
f = sqrt((1 + ((1.5 * ((1 - (y^3))^(-2/3))) * y)^2))
```

Det är inte lätt att se vilka som är korrekta och flera studenter hade parentesfel på labben.

Riktiga rishögar kan man dela upp. Här från en riktig kod som jag har hjälpt till att optimera:

```
A=1/(160*Z(b)*x*Rc(j)^3)*(x^5-5*Z(b)*x^4+10*x^3*...
(Z(b)^2-Rc(j)^2-Rn(j)^2)+10*x^2*(3*Rn(j)^2*Z(b)-...
2*Rn(j)^3-2*Rc(j)^3+3*Rc(j)^2*Z(b)-Z(b)^3)+5*x*...
(6*Rn(j)^2*Rc(j)^2-6*Rn(j)^2*Z(b)^2+8*Rn(j)^3*...
Z(b)-3*Rn(j)^4-3*Rc(j)^4+8*Z(b)*Rc(j)^3-6*Rc(j)^2*...
Z(b)^2+Z(b)^4)-4*Rc(j)^5+15*Z(b)*Rc(j)^4-20*Rc(j)^3...
*(Z(b)^2-Rn(j)^2)+10*Rc(j)^2*(2*Rn(j)^3+Z(b)^3-3*...
Z(b)*Rn(j)^2)-Z(b)^5+10*Rn(j)^2*Z(b)^3-20*Rn(j)^3*...
Z(b)^2+15*Rn(j)^4*Z(b)-4*Rn(j)^5)*1/E(i)*S(i,k)*dx
```

`...` markerar fortsättningsrad och uttrycket innehåller vektorer (som vi inte har tittat på än). Vårt första exempel blir tämligen lättläst om vi skriver:

```
temp = 1.5 * y * (1 - y^3)^(-2/3)
f    = sqrt(1 + temp^2)
```

2.3 Några elementära funktioner

```
>> sin(0)
ans = 0

>> sin(pi)          % pi fördefinierat
ans = 1.2246e-16

>> cos(0)
ans = 1

>> cos(pi)
ans = -1

>> exp(1)
ans = 2.7183
```



```
>> format long      % byt utskriftsformat
>> exp(1)
ans = 2.718281828459046

>> exp(10)
ans = 2.202646579480672e+04

>> format           % default, standardformat
>> log(exp(1))

ans = 1

>> format compact   % färre blankrader
>> ln(1)
??? Undefined function or method 'ln' for
    input arguments of type 'double'.

>> log10(1000)
ans = 3

>> log2(8)
ans = 3

>> 4 * atan(1)
ans = 3.1416

>> cosh(0)
ans = 1

>> sqrt(2)
ans = 1.4142

>> abs(-3)
ans = 3
```


2.4 `which` och `whos`

```
>> a = 10;
>> which a
a is a variable.

>> cos(2)
ans = -0.416146836547142

>> which cos
built-in (/chalmers/sw/sup/matlab-2009b/toolbox/matlab/
          elfun/@double/cos)  % double method

>> cos = 10;
>> cos(2)
??? Index exceeds matrix dimensions.

>> which cos
cos is a variable.

>> clear cos
>> which cos
built-in (/chalmers/sw/sup/matlab-2009b/toolbox/matlab/
         elfun/@double/cos)  % double method

>> whos
  Name      Size            Bytes  Class     Attributes

  a         1x1                 8  double
  ans       1x1                 8  double
```

Kan använda GUIt i stället för `whos`.

2.5 Komplexa tal

MATLAB stödjer räkning med komplexa tal, här några exempel:

```
>> z = 1 + 3i        % notera i
z = 1.0000 + 3.0000i

>> w = 2 + 4j        % j, för fysiker, j-omega-metoden
w = 2.0000 + 4.0000i

>> u = z * w
u = -10.0000 +10.0000i

>> abs(z)
ans = 3.1623

>> z * conj(z)
ans = 10

>> real(z)
ans = 1

>> imag(z)
ans = 3

>> i = sqrt(-1)
i = 0 + 1.0000i

>> exp(i * pi)  % eller  exp(1i * pi)
ans = -1.0000 + 0.0000i
```

Om z är ett komplext tal så gäller att:

$$\cos z = \frac{e^{iz} + e^{-iz}}{2} \quad \text{och} \quad \sin z = \frac{e^{iz} - e^{-iz}}{2i}$$

Vi testar:

```
>> cos(z), (exp(i * z) + exp(-i * z)) / 2
ans = 5.4396 - 8.4298i
ans = 5.4396 - 8.4298i

>> sin(z), (exp(i * z) - exp(-i * z)) / (2 * i)
ans = 8.4716 + 5.4127i
ans = 8.4716 + 5.4127i
```

Vi fortsätter i samma anda:

$$\cosh z = \frac{e^z + e^{-z}}{2} = \cos(iz)$$

$$\sinh z = \frac{e^z - e^{-z}}{2} = -i\sin(iz)$$

```
>> cosh(z), cos(i * z)
ans = -1.5276 + 0.1658i
ans = -1.5276 + 0.1658i

>> sinh(z), -i * sin(i * z)
ans = -1.1634 + 0.2178i
ans = -1.1634 + 0.2178i
```

Det borde också gälla, med $z = re^{i\varphi}$, att

$$\log z = \log(re^{i\varphi}) = \log r + i\varphi$$

Om speciellt $r = 1$ borde gälla att

$$\log e^{i\varphi} = i\varphi$$

```
>> log(-1)
ans = 0 + 3.1416i

>> 2 * log(i)
ans = 0 + 3.1416i
```


3 Logiska uttryck

Ett logiskt uttryck har värdet sant eller falskt. I MATLAB (liksom i C) representeras falskt av värdet noll och sant av ett värde skilt från noll. I andra språk har man speciella symboler, som **`false`** och **`true`**. (MATLAB har dock två funktioner **`false`** och **`true`**.)
Om **`summa`** har värdet **`5`** så har det logiska uttrycket **`summa <= 10`** värdet **`1`** och **`summa > 10`** värdet **`0`**. **`<=`** och **`>`** är exempel på *relationsoperatorer* (eng. relational operators).

```
>> summa = 5;
>> summa <= 10
ans = 1

>> summa > 10
ans = 0
```

Här en tabell över relationsoperatorerna:

symbol	betydelse
`<`	mindre än
`>`	större än
`<=`	mindre än lika med
`>=`	större än lika med
`==`	lika med
`~=`	skilt från

Observera att **`=<`** samt **`=>`** ej är tillåtna.
Vi kan sätta samman de enkla uttrycken ovan med hjälp av logiska operatorer (eng. logical operators). Säg att vi vill kontrollera om **`x`** ligger i intervallet **`[2, 7]`**. Uttrycket, **`2 <= x <= 7`**, kontrollerar *inte* det vi vill, t.ex.

```
>> x = 10;
>> 2 <= x <= 7
ans = 1

>> x = -1;
>> -2 <= x <= 0
ans = 0
```


Vad som händer i dessa exempel är att uttrycken evalueras från vänster till höger. **`2 <= x`** har värdet ett, som är **`<= 7`**. **`-2 <= x`** har också värdet ett, men det gäller inte att **`1 <= 0`**. Vi noterar att man kan behandla de logiska konstanterna som vore de reella tal (inte möjligt i alla språk, t.ex. Fortran).

Så här får vi utföra kontrollen:

```
>> x = 10;
>> 2 <= x  &  x <= 7    % & betyder och
ans = 0

>> x = -1;
>> -2 <= x  &  x <= 0
ans = 1
```

Uttrycket **`x < -2 | 2 < x`** är sant när **`x`** är mindre än **`-2`** eller **`2`** är mindre än **`x`**. Det icke uteslutande eller betecknas alltså med **`|`**.

Negation betecknas med tilde, **`~`**.

```
>> ~0
ans = 1

>> ~1
ans = 0
```

Värdet av uttrycket, **`~(x <= -1)`**, är ett när **`x`** är större än **`-1`**.

Det är viktigt med parentesen. Eftersom man kan negera tal (och inte bara logiska värden) har **`~x`** ett värde för varje **`x`**. I vissa andra språk kan man *inte* skriva så här:

```
>> ~pi
ans = 0

>> ~sin(1.435)
ans = 0
```


Detta gör att både **`~(x <= -1)`** och **`~x <= -1`** är tillåtna uttryck, men det andra är nog inte det man tänkt sig, ty **`~x <= -1`** får värdet **`0`** för *alla* **`x`**. Varför? Jo, **`~x`** beräknas först. När **`x = 0`** så blir **`~x`** lika med **`1`** och **`1 <= -1`** är falskt. För nollskilda **`x`** så blir **`~x`** lika med **`0`**, som inte är mindre än **`-1`**.

Här en tabell över logiska operatorer och deras funktion:

a	b	a & b	a | b	~a	xor(a, b)
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

`xor` står för exclusive or (antingen eller). **`|`** står ju för det vanliga icke uteslutande eller.

Det finns två varianter av **`&`** samt **`|`**. I ett uttryck som **`a & b`** finns det ingen anledningen att kontrollera värdet på **`b`** om **`a`** är falskt (tänk på **`a`** och **`b`** som komplicerade uttryck). Analogt för **`a | b`** om **`a`** är sant.
För detta ändamål finns två "short-circuit"-varianter (med MATLABs) språkbruk. **`a && b`** respektive **`a || b`**. **`b`** räknas inte ut om värdet på det logiska uttrycket bestäms av värdet på **`a`**.

Detta är ett exempel på lat evaluering (lazy evaluation); uttryck beräknas först när/om de behövs, en teknik som är vanlig i funktionella språk, t.ex. Haskell.

Varning för C/C++-programmerare: I C/C++ står **`&&, ||`** för logiskt och, eller. **`&, |`** betecknar bitvis och, eller.

Bitvisa operationer finns även i MATLAB, t.ex. bitvis och, **bitand**. Här följer ett exempel. Exekveringen av följande rader (**printbits** har jag skrivit själv)

```
a = uint8(2^5 + 2^4 + 2^3 + 1)
b = uint8(2^7 + 2^3 + 2 + 1)

printbits(a)
printbits(b)
disp('    ----------------------------------')
printbits(bitand(a, b))
printbits(bitor(a, b))
```

ger resultatet

```
a = 57
b = 139
     0    0    1    1    1    0    0    1
     1    0    0    0    1    0    1    1
    ----------------------------------
     0    0    0    0    1    0    0    1
     1    0    1    1    1    0    1    1
```

uint8 konverterar ett heltal till ett åttabitars heltal utan tecken (**u** = unsigned).

Prioritetsordningen för de logiska operatorerna är:

~	högsta
&	
|	
&&	
||	lägsta

Exempel. Givet **x** och **y** så vill vi kontrollera om punkten **(x, y)** ligger i rektangeln med hörn i **(0, 0)**, **(2, 0)**, **(2, 1)** och **(0, 1)**. Följande logiska uttryck är sant om så är fallet:

```
0 <= x & x <= 2 & 0 <= y & y <= 1
```

alternativt

```
abs(x - 1) <= 1 & abs(y - 0.5) <= 0.5
```

Följande uttryck är sant om punkten ligger utanför rektangeln:

```
x < 0 | 2 < x | y < 0 | 1 < y
```

Man kan ha användning av De Morgans lagar, **~(p & q)** är ekvivalent med **(~p) | (~q)** (vilket kan skrivas **~p | ~q**, eftersom negation har högre prioritet än eller).
Analogt är **~(p | q)** ekvivalent med **~p & ~q**.

Så, att testa om **x** *inte* tillhör intervallet **[1, 4]** kan skrivas

```
x < 1 | 4 < x        eller (De Morgan)
~(x >= 1 & 4 >= x)   eller
~(1 <= x & x <= 4)
```

■

Exempel. Tolka

```
0 <= x & x <= 2 | 0 <= y & y <= 1   och
0 <= x | x <= 2 & 0 <= y | y <= 1
```

Jag har markerat evalueringsordningen (för **|** och **&**):

```
0 <= x & x <= 2 | 0 <= y & y <= 1
       1        3        2
```

är sant om $x \in [0, 2]$ eller om $y \in [0, 1]$ (en korsliknande mängd).

```
0 <= x | x <= 2 & 0 <= y | y <= 1
       2        1        3
```

är sant om $x \leq 2$ och $0 \leq y$ eller om $0 \leq x$ eller om $y \leq 1$. Så detta är alltid sant (rita ett mängddiagram). ■

3.1 Villkorssatser

Logiska uttryck är inte så intressanta om vi inte kan testa deras värden. Detta gör vi med villkorssatser (alternativsatser, if-satser). Här följer den enklaste varianten:

```
if logiskt_uttryck
  en eller flera satser
end
```

Om värdet på det logiska uttrycket är sant (värdet är skilt från noll), så utförs satserna. Om det är falskt (värdet noll) så utförs inte satserna. Satsen kallas ibland "if-then-sats". I vissa programspråk skrivs satsen nämligen något i stil med:

```
if ( logiskt_uttryck ) then   % notera then
  en eller flera satser
end if
```

Om vi återvänder till rektangelexemplet kan vi skriva

```
x = 1.8;
y = 0.66;
if 0 <= x & x <= 2 & 0 <= y & y <= 1
  disp('punkten ligger i rektangeln')
end
```

och i detta exempel skrivs texten ut. Ändrar vi **x** till 4, t.ex. så skrivs texten inte ut. Antag att vi vill att detta faktum skall skrivas ut. Vi använder då en "if-then-else-sats".

```
if 0 <= x & x <= 2 & 0 <= y & y <= 1
  disp('punkten ligger i rektangeln')
else
  disp('punkten ligger inte i rektangeln')
end
```

En god vana är att flytta in, indentera (eng. indent) satserna, för att öka läsbarheten. MATLABs editor gör det automatiskt. Det är svårt att läsa stora program som har rak vänstermarginal.

Ett inte alldeles ovanligt fel (onödig konstruktion) är att skriva

```
if logiskt_uttryck == 1
  satser
end
```

men det är som att säga "om sant är lika med sant så...". Det räcker att fråga, "om det är sant så...". Man skriver således

```
if x > 2
  ....
end
```

och inte

```
if (x > 2) == 1  % samma som x > 2 == 1
  ....
end
```

Exempel. Vi har en uppsättningen intervall, $[1, 3]$, $[4, 5]$, $[6, 14]$ och vill ge variabeln, **k** värdet ett, två eller tre beroende på i vilket intervall variabeln **x** ligger. Om **x** inte tillhör något intervall skall variabeln få värdet noll.

```
if 1 <= x & x <= 3       % om x ligger i [1, 3]
  k = 1;
elseif 4 <= x & x <= 5   % annars om x ligger i [4, 5]
  k = 2;
elseif 6 <= x & x <= 14  % annars om x ligger i [6, 14]
  k = 3;
else                     % annars
  k = 0;
end
```

Om man tänker på effektivitet, har man det vanligaste alternativet först etc. ■

Exempel. Säg att intervallen är $]-\infty, 3[$, $[3, 5[$, $[5, \infty[$. Vi kan då förenkla koden.

```
if x < 3
  k = 1;
elseif x < 5  % behöver ej kolla 3 <= x
  k = 2;
else          % behöver ej kolla 5 <= x
  k = 3;
end
```

■

Man kan ha if-satser inuti if-satser, nästlade if-satser (svengelska, nästade, eng. nested).

Exempel. Givet **(x, y)** vill vi avgöra om punkten ligger på eller innanför enhetscirkeln och om det är fallet avgöra vilken kvadrant punkten ligger i. Vi struntar i entydighetsproblemet när en punkt ligger på en av koordinataxlarna.

```
if x^2 + y^2 <= 1     % innanför enhetscirkeln?

  if 0 <= x           % högra halvplanet?
    if 0 <= y         % första kvadranten?
      kvad = 1;
    else
      kvad = 4;       % fjärde kvadranten
    end
  else                % vänstra halvplanet
    if 0 <= y         % andra kvadranten?
      kvad = 2;
    else
      kvad = 3;       % tredje kvadranten
    end
  end

else
  kvad = 0;           % utanför enhetscirkeln
end
```


Notera indenteringen! Jag har lagt in två blankrader som kanske ökar läsbarheten. ■

Ett alternativ till if-satsen är ibland switch-satsen (kallas case-sats i vissa språk), här två exempel:

Exempel.

```
direction = 'right';

switch direction
  case 'up'
    k = 1;
  case 'down'
    k = 2;
  case 'left'
    k = 3;
  case 'right'
    k = 4;
  otherwise
    warning('direction has an illegal value')
end
```

warning skriver ut meddelandet och var i programmet felet inträffade. ■

Exempel.

```
pick = 3;

switch pick
  case {1, 5}
    % räkna ...
    disp('first')
  case {2, 3}
    % räkna ...
    disp('second')
end
```

■

4 Repetitionssatser

4.1 For-satsen

Exempel. Säg att vi vill beräkna en approximation av

$$\sum_{k=1}^{1000} \frac{1}{k}$$

Ett jobbigt sätt att göra detta är följande:

```
>> summa = 1
summa = 1

>> summa = summa + 1 / 2
summa = 1.5000

>> summa = summa + 1 / 3
summa = 1.8333

>> summa = summa + 1 / 4
summa = 2.0833
```

En konstruktion som **summa = summa + term** skall INTE tolkas som en ekvation utan som följer: **summa = 1**, lagra talet ett i minnet som svarar mot variabeln **summa**. **summa = summa + 1/2**, **summa** i högerledet svarar mot det värde, ett, på **summa** som ligger i minnet. Detta värde adderas till **1/2**, vilket ge **3/2**, och detta nya värde lagras nu i minnet och skriver över det gamla värdet på **summa**. Så, ungefär $\texttt{summa}_{ny} = \texttt{summa}_{gammal} + \texttt{1 / 2}$.

Vi vill nu låta MATLAB beräkna summan automatiskt åt oss och vi använder en sk repetitionssats som kan ha utseendet:

```
for variabel = start:stopp
  en eller flera satser som skall upprepas
end
```

Variabeln kallas ofta loopvariabel, styrvariabel (eng. loop variable, loop counter). En for-sats kallas också for-loop, for-snurra, forslinga på "svenska".

Satserna mellan **for** och **end** kallas loopkropp (eng. loop body) och de utförs när variabeln antar värden **start**, **start + 1**, ..., **stopp**.

Här ett enkelt exempel:

```
>> for k = 1:4
     k           % skriv ut värdet på k
   end

k = 1
k = 2
k = 3
k = 4

>> for k = -3:-1
     k
   end
k = -3
k = -2
k = -1

>> for k = 5:5   % start = stopp
     k
   end
k = 5


>> for k = 7:5
     k           % utförs ej, ty start > stopp
   end
```

Satserna i loopkroppen får använda värdet på loopvariabeln i beräkningar, men de får (bör) inte ändra på loopvariabelns värde (förbjudet i vissa språk).

Vi kan nu beräkna vår summa:

```
>> summa = 0;                  % ; ger ingen utskrift
>> for k = 1:1000
     summa = summa + 1 / k;    % ; ger ingen utskrift
   end

>> summa                       % värdet på summan
summa = 7.4855

>> format long
>> summa
summa = 7.485470860550343  % fler decimaler
```

Jag skrev raderna ovan direkt i MATLABs kommandofönster. Normalt skriver man in sina program med en editor. En god vana är att flytta in, indentera loppkroppen, för att öka läsbarheten. MATLABs editor gör det automatiskt.
Låt oss beräkna samma summa baklänges, $1/1000 + 1/999 + \ldots + 1/2 + 1$.

```
>> summa = 0;
>> for k = 1:1000
     summa = summa + 1 / (1001 - k);
   end
>> summa
summa = 7.485470860550341
```

Framåt- och bakåtsumman skiljer sig lite åt (eftersom vi inte räknar exakt):

```
>> 7.485470860550341 - 7.485470860550343
ans = -2.664535259100376e-15
```

I kursen i numerisk analys kommer vi att se vilken summa som ger den bästa approximationen.
En alternativ lösning på ovanstående problem är att utnyttja en steg-parameter (får ej vara noll):

```
for variabel = start:steg:stopp
  en eller flera satser som skall upprepas
end
```


Här några enkla exempel:

```
>> for k = 1:2:4
     k              % k = 4 skrivs ej ut
   end
k = 1
k = 3

>> for k = 1:-1:-2  % negativt steg
     k
   end
k = 1
k = 0
k = -1
k = -2
```

Med denna teknik kan bakåtsumman beräknas som följer:

```
>> summa = 0;
>> for k = 1000:-1:1
     summa = summa + 1 / k;
   end
```

Det är vanligt att man har loopar inuti loopar, nästlade loopar (eng. nested loops). Vi vill beräkna summan:

$$\left[\frac{1}{1^2}\right] + \left[\frac{1}{1^2} + \frac{1}{2^2}\right] + \left[\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2}\right] + \ldots + \left[\frac{1}{1^2} + \frac{1}{2^2} + \ldots + \frac{1}{n^2}\right]$$

Vi skriver uttrycket med vanlig notation:

$$\sum_{j=1}^{n} \left(\sum_{k=1}^{j} \frac{1}{k^2} \right)$$

och ser att den inre summan beror av iterationsvariabeln i den yttre summan. Nu kommer en ineffektiv lösning i MATLAB där vi varje gång beräknar **term** lika med $\sum_{k=1}^{j} 1/k^2$.


```
>> summa = 0;
>> n = 10;
>> for j = 1:n
     term = 0;
     for k = 1:j
       term = term + 1 / k^2;
     end
     summa = summa + term;
   end
>> summa
summa = 1.4118e+01
```

Detta är ineffektivt eftersom

$$\sum_{k=1}^{j+1} \frac{1}{k^2} = \sum_{k=1}^{j} \frac{1}{k^2} + \frac{1}{(j+1)^2}$$

Här följer en effektivare lösning som utnyttjar denna iakttagelse:

```
>> summa = 0;
>> n     = 10;
>> term  = 0;
>> for j = 1:n
     term  = term + 1 / j^2;  % inre summa
     summa = summa + term;    % yttre summa
   end
>> summa
summa = 1.4118e+01
```

4.2 While-satsen

for-satser används normalt när man vet hur många iterationer som skall utföras. Ibland är detta antal okänt i förväg, som i följande exempel: Vi vet att den harmoniska serien är divergent. Hur många termer krävs för att summan skall överstiga 10? Här en algoritmskiss:


```
summa = 0
term_nummer = 0
upprepa så länge som summan är <= 10
  term_nummer = term_nummer + 1
  summa = summa + 1 / term_nummer
slut på upprepningen
skriv ut antalet termer, term_nummer
```

Detta är exempel på en sk "while-loop" och konstruktionen finns i MATLAB. En annan loop-konstruktion, som ej direkt stödjs av MATLAB (men som finns i andra språk) är repeat-until, "upprepa till dess att".

```
summa = 0
term_nummer = 0
upprepa
  term_nummer = term_nummer + 1
  summa = summa + 1 / term_nummer
till dess att summan är > 10
skriv ut antalet termer, term_nummer
```

I denna variant har vi testen på slutet av loopen. I while-loopen testar vi alltid först. Detta gör att loopkroppen i en while-loop inte behöver exekveras, men en repeat-until-loop utförs alltid minst en gång.

```
>> summa = 0;
>> k = 0;
>> while summa <= 10
     k = k + 1;
     summa = summa + 1 / k;
   end

>> summa
summa = 10.000043008275778

>> k
k = 12367
```

Hur väl stämmer detta med följande gränsvärde?

$$\lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} - \log n = \gamma \approx .577215664901532860 \text{ Eulers konstant}$$

```
>> summa - log(k)
ans = 0.577256094533722  % stämmer rätt bra
```

Hur många termer krävs för att summan skall överstiga 1000000? Vi kan få en uppskattning av värdet genom att använda gränsvärdet:

$$10^6 - \log n \approx \gamma \Rightarrow n \approx e^{(10^6-\gamma)}$$

Detta ger ett enormt stort tal, försöker vi räkna ut n i MATLAB får vi overflow. så

```
>> n = exp(1e6 - .577215664901532860)
n = Inf
```

Så här kan vi göra

$$\log_{10} n \approx \log_{10}\ e^{(10^6-\gamma)} = (10^6 - \gamma)\log_{10} e$$

i MATLAB:

```
>> log10_n = (1e6-.577215664901532860) * log10(exp(1))
log10_n = 4.342942312216737e+05

>> int_part = floor(log10_n)   % ny funktion
int_part = 434294

>> dec_part = log10_n - int_part
dec_part = 0.231221673660912

>> 10^dec_part
ans = 1.703027550110583
```

så att

$$n \approx 1.7 \cdot 10^{434294}$$

vilket är ett stort tal. Antag att divisionen dominerar tidsmässigt, med kanske $3ns$ (nano-sekunder). I själva verket tar MATLAB mer tid.

Om vi hade utfört divisioner från "Big Bang" (för ca $13.7 \cdot 10^9$ år sedan) hade vi hunnit utföra

$$13.7 \cdot 10^9 \cdot 365 \cdot 24 \cdot 3600/3 \cdot 10^{-9} \text{ divisioner.}$$

```
>> 13.7e9 * 365 * 24 * 3600 / 3e-9
ans = 1.4401e+26
```

en droppe i havet.

Det är enklare att göra fel när man arbetar med while-loopar än när man använder for-loopar. Man har väsentligen två felkällor:

- initiering och uppdatering av variabler
- formulering av det logiska villkoret

```
initiera variabler
while logiskt uttryck
  räkna, uppdatera variabler
end
```

Här är några varianter av summa-problemet:

```
summa = 0;
k = 0;                      % initieringen passar inte
while summa <= 10
  summa = summa + 1 / k;    % division med noll
  k = k + 1;                % ihop med uppdateringen
end
```

Följande kod ger rätt värde på **summa**, men fel värde på **k** (om vi inte tänker oss för).

```
summa = 0;
k = 1;
while summa <= 10
  summa = summa + 1 / k;
  k = k + 1;                % öka efter användning
end
k                           % skriv ut k
```


Detta fel, om man nu gör det, att öka **k** en gång för mycket är exempel på ett sk "off by one error", man ökar **k** en gång för mycket.
Vi kan ju enkelt kompensera för denna ökning, genom att subtrahera ett från **k** efter loopen. Denna typ av fel karakteriseras av att man utför något en gång för mycket eller litet.

Här ytterligare en felaktig variant.

```
summa = 1;              % summa = 1
k = 1;
while summa <= 10
  summa = summa + 1 / k;
  k = k + 1;
end
```

och här en korrekt variant:

```
summa = 1;              % summa = 1
k = 1;
while summa <= 10
  k = k + 1;
  summa = summa + 1 / k;
end
```

För att undvika dessa fel bör man alltid kontrollera första- och sista iterationen. Man bör också kontrollera en allmän iteration "mitt i".

Så, i föregående exempel:
Begynnelsevärden **summa = 1** och **k = 1**. I första iterationen kommer **k** att sättas till två och **summa** blir **1 + 1 / 2**. Korrekt!

I sista iterationen ökas **summa** med **k** så att **k** är det sista värdet som används i summan. Om vi skriver ut **summa** och **k** efter loopen får vi matchande värden.

I en allmän iteration bildar vi **k = k + 1;** samt
summa = summa + 1/k; så att **summa = summa + 1/(k + 1)** vilket är korrekt (givet korrekta begynnelsevärden).

Exempel. Beräkna en partialsumma av Taylorutvecklingen av $\log(1+x)$.

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n-1}\frac{x^n}{n}$$

Här följer några olika varianter.

```
n = 10;
x = 1.24526;  % till exempel
summa = 0;
for k = 1:n
  summa = summa + (-1)^(k - 1) * x^k / k;
end
```

Det är sällan en god idé att skriva in matematiska formler direkt i program. Man kan få problem med beräkningsfel (vi räknar ju inte exakt) och dels kan man få långsam kod. I exemplet, när **n = 10**, spelar detta ingen roll. Det skall sägas att Taylorutvecklingar inte används för att beräkna approximationer av elementära funktioner.

Hur beräknas $x^k, k > 0$? Vi vet inte riktigt hur MATLAB gör detta, men sannolikt något i stil med följande:

- Genom upprepad multiplikation om k är ett litet positivt heltal. Så x^3 beräknas nog som $x \cdot x \cdot x$.
- För större heltalsvärden kam man gör något bättre. x^8 skulle kunna beräknas som: $t = x \cdot x$ (så $t = x^2$), $t = t \cdot t$ (så $t = x^4$) och $t = t \cdot t$ (så $t = x^8$). Detta kräver tre multiplikationer i ställer för sju som upprepad multiplikation gör.
- Om k inte är ett heltal (eller om k är ett stort heltal) brukar man använda omskrivningen

$$x^k = e^{\log(x^k)} = e^{k \cdot \log x}$$

Teckenväxlingen kan erhållas via $(-x)^k, k = 0, 1, \ldots$:

```
n     = 10;
x     = 1.24526;  % till exempel
summa = 0;
for k = 1:n
  summa = summa - (-x)^k / k;  % notera första minus
end
```

Vi vill kanske inte använda "upphöjt till" (tar lite tid), vanlig multiplikation är snabbare. Om `n = 10` spelar det dock ingen som helst roll vilket alternativ vi använder.
Här är ett alternativ utan upphöjt till:

```
sgn = 1      % sign är upptaget

sgn = -sgn   % sgn = -1
sgn = -sgn   % sgn =  1
sgn = -sgn   % sgn = -1
```

Här lite kod:

```
n     = 10;
x     = 1.24526;
xk    = x;
summa = 0;
sgn   = 1;
for k = 1:n
  summa = summa + sgn * xk / k;
  xk    = x * xk;
  sgn   = -sgn;
end
```

Eller ännu kortare

```
xk    = x;
summa = 0;
for k = 1:n
  summa = summa + xk / k;
  xk    = -x * xk;    % teckenväxling
end
```


Ännu lite snabbare är

```
xk    = x;
summa = 0;

minus_x = -x;
for k = 1:n
  summa = summa + xk / k;
  xk    = minus_x * xk;
end
```

Vi kan göra på detta sätt, eftersom `-x` är konstant i loopen, det är invariant. Mer allmänt så är det onödigt att beräkna ett konstant uttryck flera gånger. I ovanstående exempel spelar det ingen roll, men det gör det i följande:

```
tic                                % starta tidtagning
s = 0;
for k = 1:10000000
  s = s + exp(sin(-1))^exp(1) / k; % räknas ut 10000000
end                                % gånger
toc                                % avsluta tidtagning
```

Här följer ett enkelt exempel på kodoptimering:

```
tic
temp = exp(sin(-1))^exp(1);        % räknas ut EN gång
s    = 0;
for k = 1:10000000
  s = s + temp / k;                % återanvänd
end
toc
```

Den första loopen tar 6.4 s och den andra 0.15 s, en uppsnabbning på drygt 40 gånger.
Men kom ihåg:

Make it right before you make it faster.

"Snabbt men fel" är av föga intresse.

Exempel. Det är vanligt att man måste beräkna min/max i en loop. Här ett litet problem (programmet kan göras effektivare, hur?). Vi vill hitta min/max av summan, `s`, och för vilket `k` min/max antas.

$$s = \sum_{k=0}^{20} \frac{x^k}{k!}$$

```
x     = -8.5;          % given value
summa = 1;             % summa = x^0 / 0! = 1
min_summa = summa; min_k = 0;
max_summa = summa; max_k = 0;

for k = 1:20
  summa = summa + x^k / factorial(k);
  if summa < min_summa
    min_summa = summa;
    min_k = k;
  elseif max_summa < summa
    max_summa = summa;
    max_k = k;
  end
end

min_k, min_summa
max_k, max_summa
```

Resultatet blir

```
min_k = 7
min_summa = -338.2279

max_k = 8
max_summa = 337.5919
```


Man skall inte ändra på loopvariabeln i en for-loop. Det är inte ens tillåtet i vissa språk.

```
for k = 1:4
  k
  k = k + 3
end

k = 1     % första iterationen
k = 4
k = 2     % nästa iteration
k = 5
k = 3     % nästa iteration
k = 6
k = 4     % nästa iteration
k = 7
```

Om man behöver en loop där index ändras mer dynamiskt använder man en while-loop.

```
k = 1;                    % till exempel
while inte_färdig
  k = k + något_man_räknat_fram;
  ...
end
```

Ibland inträffar undantagshändelser i en loop. Ett sätt att lösa det är med logiska variabler, här ett enkelt exempel som illustrerar principen:

```
fail = false;
k    = 0;

while k < n & ~fail
  k = k + 1;
  work ....
  if något går fel
    fail = true;
  end
end
```

Om ett fel inträffade så innehåller **k** index för den iteration som gick fel.

En enklare lösning får man med **break**-satsen.

```
for k = 1:n
  work ....
  if något går fel
    break
  end
end
```

Om **break**-satsen exekveras kommer **k** att innehålla index för den iteration som gick fel. Om inget gick fel så är **k = n** efter loopen. Om man, i efterhand, måste kunna kontrollera om något gick fel eller inte i sista iterationen, kan man göra så här:

```
fail = false;
for k = 1:n
  work ....
  if något går fel
    fail = true;
    break
  end
end
```


Observera att **break** bryter från den innersta **for**-satsen (om man har nästlade loopar). **break** kan också användas i while-loopar.

```
for j = 1:3
  j
  for k = 1:2
    k
    if j == 2
      break
    end
 end
end
```

ger (väsentligen) utskriften

```
j = 1
k = 1
k = 2

j = 2
k = 1  <- break

j = 3
k = 1
k = 2
```

En annan, besläktad, konstruktion är **continue**, som gör att man hoppar över resten av aktuell iteration, och fortsätter med nästa:

```
for j = 1:3
  if j == 2
    continue
  end
 j
end

ger

j = 1
j = 3
```


return-satsen, slutligen avbryter exekveringen (och hoppar tillbaks till anropande funktion).

```
>> type cont

for j = 1:3
  if j == 2
    return
  end
 j
end
disp('after the loop')

>> cont
j = 1
```

Detta kan vara användbart vid t.ex. avlusning. Observera att **break** hade gett uthopp ur loopen, men **disp**-raden hade exekverats.

MATLAB-funktionerna **warning** och **error** kan vara användbara i detta sammanhang.

```
  if något går fel
     warning('Felmeddelande')
       eller
     error('Felmeddelande')
  end
```

Båda funktionerna skriver ut felmeddelandet och positionen i koden. **error** avbryter dessutom exekveringen.

4.3 Från problem till MATLAB-kod, ett exempel

Följande exempel försöker visa hur vägen från problemställning till färdig MATLAB-kod kan se ut. Problemet, en Diofantisk ekvation, är också utformat så att det ger lite ledning till en laboration.
Diofantos, grekisk matematiker som levde år [200,214]-[284,298].

Utmärkande för en Diofantisk ekvation är man bara tillåter heltalslösningar.

Exempel. En Pythagoreisk trippel är tre positiva heltal, (p, q, r) som satisfierar $p^2+q^2 = r^2$. Två lösningar är $(3, 4, 5)$ och $(5, 12, 13)$.

Nu till programmeringsproblemet:

Exempel. Hitta 10 lösningar, (x, y), till Pells ekvation:

$$x^2 - n \cdot y^2 = 1$$

n är ett positivt heltal (men inte en jämn kvadrat). I denna uppgift är $n = 8$ och x samt y skall vara positiva heltal. Det finns en heltalslösning $(1, 0)$, men den behöver programmet inte beräkna.

Vi ignorerar det faktum att det finns en hel del matematiska resultat om Pells ekvation, utan i stället försöker vi resonera oss fram till ett MATLAB-program som löser problemet.

För att skriva ett program måste man ha en algoritmidé, vi börjar inte med att skriva MATLAB-kod. Ett sätt att få en idé är att fundera på hur man skulle löst problemet för hand.

Första idén: vi testar med några tal (gissar) och hittar kanske lösningen $(3, 1)$: $3^2 - 8 \cdot 1^2 = 1$. Man kan kanske hitta några till på detta sätt, men man lär inte hitta den tionde, $(22619537, 7997214)$. Det är också svårt att skapa ett program utifrån gissningsidén och körtiden blir omöjlig att uppskatta.
Vi måste vara mer systematiska.

Andra idén (brute force): vi testar alla heltalspar. Om det finns minst tio lösningar (vi får lite på den, jag, som konstruerat uppgiften) kommer algoritmen att fungera. Så, vi tar $y = 1$ och sedan $x = 1, 2, 3 \ldots$. Sedan tar vi $y = 2$ och sedan $x = 1, 2, 3 \ldots$ etc.

Denna algoritm har flera brister. $x = 1, 2, 3 \ldots$ kommer inte att kunna implementeras, det finns ju oändligt många positiva heltal. Vi kan testa med $x = 1, 2, 3, \ldots,$ *stort heltal*, men vad är "*stort*". Om talet är för litet kommer vi inte att hitta de tio lösningarna. Säg att vi låter, "*stort heltal*" vara 10^6, lite godtyckligt. Då kan ett program se ut som:

```
n = 8;
for y = 1:1e6
  for x = 1:1e6
    if x^2 - n * y^2 == 1
      disp([x, y])  % för snyggare utskrift
    end
  end
end
```

Det andra bekymret är exekveringstiden. If-satsen kommer att utföras 10^{12} gånger. Jag har uppskattat tiden till 18.5 timmar på en lab-dator. Dessutom kommer vi bara att hitta åtta lösningar.

Tredje idén: vi måste utnyttja det faktum att x inte kan vara godtyckligt om vi vet värdet på y. Det gäller ju att:

$$x^2 = 1 + n\,y^2$$

så $x < \sqrt{n} \cdot y$ måste vara uppfyllt. Detta är nu en för svag uppskattning, vi måste göra en bättre. Om y är stort så måste $x \approx \sqrt{n}\,y$. Detta verkar lovande, för då skulle vi helt kunna bli av med loopen för x. Om vi känner y så vet vi vad x måste vara. Lite mer matematik krävs dock, $\approx$ känns lite osäkert.

$$x = \sqrt{1 + n\,y^2} = \sqrt{n\,y^2} \cdot \sqrt{\frac{1}{n\,y^2} + 1} =$$

$$\sqrt{n}\,y \left(1 + \frac{1}{2\,n\,y^2} - \frac{1}{8\,n^2\,y^4} + \ldots\right) = \sqrt{n}\,y + \frac{1}{2\sqrt{n}\,y} - \ldots$$

Eftersom $y \geq 1$ så gäller att $x = \sqrt{n}\,y + \epsilon$ där $0 < \epsilon < 1/(2\sqrt{8}) \approx 0.18$. Eftersom $\sqrt{n}$ inte är ett heltal är alltså det enda tänkbara x-värdet $\text{ceil}(\sqrt{n}\,y)$ (ceil, för ceiling, avrundat uppåt).

Först nu är det dags att börja programmera. Här följer ett MATLAB-program där vi har en while-loop för att slippa avgöra vad "*stort tal*" skall ha för värde.

```
n     = 8;
sn    = sqrt(n);     % för att spara tid
antal = 0;           % antal lösningar
y     = 0;
while antal < 10
  y = y + 1;
  x = ceil(sn * y);  % tänkbart värde på x
  if x^2 - n * y^2 == 1
    antal = antal + 1;
    disp([x, y])
  end
end
```

Exekveringstiden är 1.5 s och här är de tio första lösningarna:

```
           3           1
          17           6
          99          35
         577         204
        3363        1189
       19601        6930
      114243       40391
      665857      235416
     3880899     1372105
    22619537     7997214
```


Det skall sägas att vi har lite tur att programmet klarar av att räkna ut alla tio lösningarna. $x^2 = 22619537^2 = 511643454094369$ är större än det största representerbara heltalet (med 32-bitars heltal är detta `intmax = 2147483647`.

Så x^2 räknas ut som ett flyttal och siffrorna räcker just till för att kvadraten, och övriga värden, skall kunna beräknas exakt. Programmet klarar inte av att räkna ut nästa lösning, eftersom antalet siffror inte räcker till.

Man kan göra en del rolig matematik med ovanstående (om man använder vektorer). Låt $y_k, k = 1, \ldots, 10$ vara y-värdena i tabellen ovan. Om man i MATLAB bildar y_{k+1}/y_k får man följande tabell

```
6.000000000000000
5.833333333333333
5.828571428571428
5.828431372549019
5.828427249789740
5.828427128427128
5.828427124854547
5.828427124749380
5.828427124746284
```

Knappast en slump! Nästa y-värde borde alltså vara ungefär

$$5.828427124746284 \cdot 7997214 \approx 46611179.000000728852776$$

och motsvarande x

$$\sqrt{8} \cdot 46611179 \approx 131836322.99999999620742$$

och testar man i Maple (som räknar exakt med heltal) får man

```
> 131836323^2 - 8 * 46611179^2;
                1
```

men i MATLAB får man värdet noll på grund av avrundningsfel.

```
>> 17380816062160329 - 17380816062160328  % OBS: olika
ans = 0
```


4.4 Beräkningskomplexitet

(eng. computational complexity). En algoritm (eng. algorithm, se Wikipedia för etymologin) är en metod för att lösa ett problem. Ett program är en implementation (realisering) av algoritmen.

En viktig del i algoritmkurser är att studera algoritmens beräkningskomplexitet, dvs. hur algoritmens minnesbehov och tidsåtgång varierar med problemstorleken.
På dessa sidor tar vi endast upp tidsaspekten.

Exempel. Att lösa $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ där $\boldsymbol{A}$ är en $n \times n$-matris kräver ungefär $n^3/3$ additioner och multiplikationer. Om det tar tiden τ att utföra ett (+, *)-par är tidsåtgången, $T(n)$, därför

$$T(n) = \frac{n^3 \tau}{3}$$

■

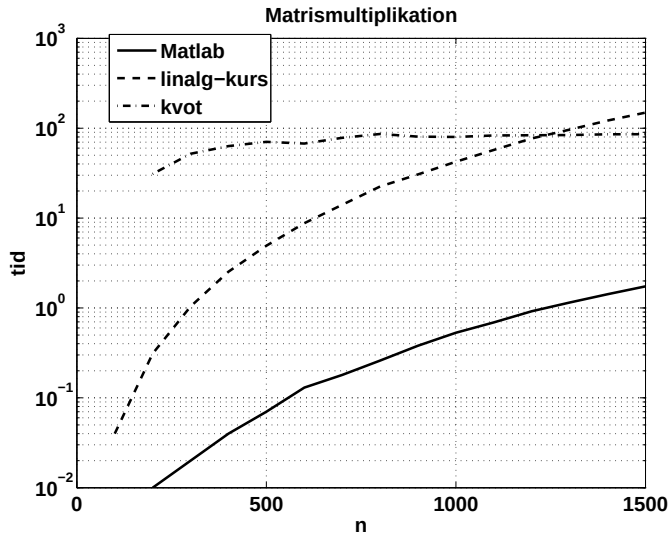
Vi kan använda sådan information för att

- Uppskatta beräkningstiden
- Se hur stort problem vi kan lösa
- Välja bland olika algoritmer

I praktiken räcker det inte att veta hur snabbt datorn utför ett (+, *)-par eftersom minnet spelar så stor roll. Två program som utför lika många operationer kan skilja sig avsevärt i tid eftersom minnet spelar in.

Exempel. Låt A, B och C vara $n \times n$-matriser. Den vanliga algoritmen, från kursen i linjär algebra, tar n^3 multiplikationer och $n^2(n-1)$ additioner. Den rutin som används i MATLAB kräver lika många räkneoperationer, men den utnyttjar minnet på ett effektivare sätt.

Här en bild



MATLABs egen rutin är nästan 100 gånger snabbare än den enkla varianten. ■

Exempel. Nästlade loopar kan ta tid:

```
Program 1       Program 2         Program 3

for i = 1:n     for i = 1:n       for i = 1:n
  work            for j = 1:n       for j = 1:n
end                 work              for k = 1:n
                  end                   work
                end                     end
                                      end
                                    end
```

Antag att `work` tar tiden τ i samtliga fall. Tiderna blir då

$$T_k(n) = n^k\tau,\ k = 1, 2, 3$$

Antag att $n = 10^6$ och att $\tau = 10^{-8}s$. Då blir

$$T_1(n) = 0.01s,\ T_2(n) \approx 2.8 \text{ timmar},\ T_3(n) \approx 317 \text{ år}.$$

Antalet nästlade loopar *spelar stor roll*. Observera dock att tre loopar, som i Program 1, *efter varandra*, endast tar $0.03s$. ■

Det är vanligt (vanligare i datalogi än i numerisk analys) att man använder stort ordo (från latin, ordning, eng. big O) för ange komplexiteten. I detta sammanhang säger vi att $T(n) = O(n^p)$ eller $T(n) = \mathcal{O}(n^p)$ om $T(n) \leq M \cdot n^p,\ M > 0$ när $n \to \infty$.

Exempel. Säg att $T(n) = c_3n^3 + c_2n^2 + c_1n + c_0$ där c_k är reella konstanter och $T(n) > 0, n > 0$. Det gäller att $T(n) = \mathcal{O}(n^3)$ ty

$$T(n) = c_3n^3 + c_2n^2 + c_1n + c_0 = n^3\left[c_3 + \frac{c_2}{n} + \frac{c_1}{n^2} + \frac{c_0}{n^3}\right] \leq$$

$$n^3\left[c_3 + \frac{|c_2|}{n} + \frac{|c_1|}{n^2} + \frac{|c_0|}{n^3}\right] \leq Mn^3$$

för tillräckligt stort n. ■

Exempel. Den sorteringsalgoritm som används i MATLAB har (sannolikt) komplexiteten $\mathcal{O}(n \log n)$ där n är antalet element i vektorn som skall sorteras. Detta gör att det går väldigt snabbt att sortera tal.

$$n \log n = \frac{n \log_{10} n}{\log_{10} e} \Rightarrow \mathcal{O}(n \log n) = \mathcal{O}(n \log_{10} n)$$

Så, sorteringstiden växer lite snabbare än linjärt med n.
Att sortera 10^6 tal tar ungefär $0.3s$. ■

I numerisk analys är det mindre vanligt med ordo-notation i detta sammanhang. Att lösa $Ax = b$, när A är symmetrisk, tar ungefär $n^3/6$ (+, *)-par (halva tiden jämfört med det osymmetriska fallet), $\mathcal{O}(n^3/3) = \mathcal{O}(n^3/6) = \mathcal{O}(n^3)$ men det är skillnad på 10 minuter och 20 minuter.

I exemplet med matrismultiplikation så är båda metoderna $\mathcal{O}(n^3)$ men den ena tar hundradelen så mycket tid.

På samma sätt vill man skilja mellan två metoder med $T_1(n) = n^3$ och $T_2(n) = 1000n^2$. Den första metoden är $\mathcal{O}(n^3)$ och den andra $\mathcal{O}(n^2)$, men den första metoden är snabbare när $n < 1000$, fastän det är en n^3-metod.

Däremot kan man tillåta säg att approximera $n^2(n-1) \approx n^3$ när det gäller matrismultiplikation.

5 Fält - vektorer och matriser, en första bekantskap

Detta är en inledning till vektorer och matriser, så att vi kan klara av intressantare laborationer. Det kommer mer efter hand.
Ett fält eller en lista är en följd av element. I MATLAB-sammanhang talar man oftare om vektorer (endimensionellt fält) och matriser (tvådimensionellt fält, eng. one- eller two-dimensional array).
MATLAB står ju för MATrix-LABoratory och man ser MATLABs styrka vid matrisräkning.

5.1 Vektorer

Precis som i linjäralgebrakursen har vektorn ett namn och vi indexerar de individuella elementen. Först några sätt att skapa vektorer:

```
>> vek = [3 6 -8]    % skapa en radvektor
vek = 3     6    -8

>> vek = [3, 6, -8]  % komma går också bra
vek = 3     6    -8

>> v = [2+3, 4*7, -3]  % kan ha numeriska uttryck i
v = 5    28    -3

>> v = [2 +3, 4*7, -3]  % Varning!
v = 2     3    28    -3

>> v = [2 + 3, 4*7, -3] % Varning
v = 5    28    -3

>> vek(1)            % index inom ( )
ans = 3

>> vek(2)
ans = 6
```

```
>> vek(0)            % indexfel
??? Subscript indices must either be real positive
    integers or logicals.

>> vek(4)            % indexfel
??? Index exceeds matrix dimensions.

>> vek(4) = 146      % vektorn utvidgas
vek = 3     6    -8   146

>> vek(2) + 3 * sqrt(-vek(3))  % numeriska uttryck
ans =                          % som vanligt
   14.4853
```

Notera att vektorer och matriser är matematisk begrepp snarare än datalogiska objekt. Vi (jag) vill alltså ha första index ett och inte noll som i C/C++. I den numeriska språket Fortran är första index ett, som standard, men man kan välja startindex, kan tom vara negativt.

I numerisk analys är det vanligast med kolonnvektorer (kommer nedan) med de tar så stor plats att skriva ut, så i denna genomgång använder jag mest radvektorer.

```
>> vek_tr = vek'     % transponera
vek_tr =
     3
     6
    -8
   146

>> vek_tr(4)         % ett index även för kolonnvektor
ans = 146
```


Här ett annat sätt att skapa kolonner:

```
>> kol = [2; -5; 7]  % ; = radbyte
kol =
     2
    -5
     7

>> x = 1:4           % vanlig typ av vektor
x =                  % jämför  for k = 1:4
     1     2     3     4

>> x = 1:4'
x = 1     2     3     4

>> x = (1:4)'        % behövs ( )
x =
     1
     2
     3
     4

>> y = 5:-2:-4
y = 5     3     1    -1    -3

>> y = 5:-2:6
y =
   Empty matrix: 1-by-0

>> nollor = zeros(1, 3)
nollor = 0     0     0

>> ettor = ones(1, 3)
ettor = 1     1     1
```



```
>> r = rand(1, 3)   % likformig fördelning på [0, 1)
r =
   4.8626e-01   4.2040e-01   8.5471e-01

>> r = randn(1, 3)  % normalfördelning
r =                 % randn(3) blir en 3 x 3-matris
   7.2579e-01  -5.8832e-01   2.1832e+00
```

5.2 Några räkneoperationer för vektorer

De vanliga vektoroperationerna fungerar som vanligt.

```
>> a = 1:3
a = 1     2     3

>> b = 4:6
b = 4     5     6

>> c = a + 2 * b
c = 9    12    15

>> (1:4) + a               % dimensionerna måste stämma
??? Error using ==> plus
Matrix dimensions must agree.

>> a + a'                  % och orienteringen
??? Error using ==> plus
Matrix dimensions must agree.


>> a * b
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```



```
>> a' * b                  % sk ytterprodukt. MATRIS
ans =
     4     5     6
     8    10    12
    12    15    18

>> a * b'                  % inner(skalär)-produkt
ans = 32
```

Mer om vektorer kommer senare, men nu lite om matriser.

5.3 Matriser

Matriser fungerar ungefär som vektorer,

```
>> M = [1 2; 3 4]    % ; = ny rad
M =
     1     2
     3     4

>> M(1, 2)           % matris(rad, kolonn)
ans = 2

>> M(2, 1) = 10      % ändra värde
M =
     1     2
    10     4

>> M'                % transponat
ans =
     1    10
     2     4

>> M(2, 4) = -77     % utvidgning
M =
     1     2     0     0
    10     4     0   -77
```

```
% Uttryck
>> sqrt(abs(M(1, 2) - M(2, 1) * M(2, 2)))
ans = 3.1623

>> v = [1 2 3 0]';
>> M * v            % matris-vektor multiplikation
ans =
     5
    11

>> [1 2] * M       % rad från vänster
ans = 7    10     0  -154

>> [1 2] * M * v
ans = 27

>> [1 2]' * M      % samma regler som vanligt
??? Error using ==> mtimes
Inner matrix dimensions must agree.

>> v = rand
v = 0.5135

>> R = [cos(v), sin(v); -sin(v), cos(v)]
R =
    0.8711    0.4912
   -0.4912    0.8711

>> R' * R          % matrismultiplikation
ans =
     1     0
     0     1

>> 2 * R           % elementvis som vanligt
ans =
    1.7421    0.9824
   -0.9824    1.7421
```


Man kan få flera siffror utskrivna genom att byta utskriftsformat (det ändrar inte på det interna binära formatet dock):

```
>> pi_vek = pi * [1e-10, 1, 1e10];
>> format short
pi_vek =
   1.0e+10 *                    % <-- OBS
    0.0000    0.0000    3.1416

>> format short e               % använder jag oftast
>> pi_vek
pi_vek =
   3.1416e-10   3.1416e+00   3.1416e+10

>> format long
>> pi_vek'                      % transponat
ans =
   1.0e+10 *                    % <-- OBS
   0.000000000000000
   0.000000000314159
   3.141592653589793

>> format long e                % tar mycket plats
>> pi_vek'                      % obs transponat
ans =
     3.141592653589793e-10
     3.141592653589793e+00
     3.141592653589793e+10

>> format bank                  % kronor och ören
>> pi_vek
pi_vek =
          0.00          3.14 31415926535.90

>> format hex   % internt hexadecimalt format
>> pi_vek
pi_vek =
   3df596bf8ce7631e   400921fb54442d18   421d4223fc1f977
```


6 Stränghantering

En teckensträng (sträng, eng. string) är en vektor av tecken. I MATLAB lagras vektorn som en vektor av motsvarande teckenkoder.

```
>> a_string = 'Matlab'
a_string = Matlab

>> double(a_string)  % teckenkoder
ans = 77    97   116   108    97    98
```

Om man tittar i manualbladet för ascii (se datoravsnittet) så framgår att M har teckenkoden 77, a har koden 97 etc. Det stämmer alltså. Man kan också gå från koder till tecken:

```
>> v = [84 104 111 109 97 115];

>> char(v)
ans = Thomas
```

Man kan också göra roliga saker som:

```
>> 'a':'z'
ans = abcdefghijklmnopqrstuvwxyz

>> 'A':'Z'
ans = ABCDEFGHIJKLMNOPQRSTUVWXYZ

>> 'A':'z'
ans =
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
```

I labben kan du behöva konvertera mellan stora och små bokstäver. Så här kan man göra. När jag bildar `s - 'a' + 'A'` subtraherar jag en skalär från en vektor. Otillåtet i kursen i linjär algebra, men tillåtet i MATLAB (matematik och MATLAB-syntax är ibland olika saker). Skalären subtraheras elementvis från elementen i vektorn. Mer om sådant senare.


```
>> s = 'a':'z';
>> char(s - 'a' + 'A')   % teckenaritmetik
ans = ABCDEFGHIJKLMNOPQRSTUVWXYZ

s = 'A':'Z';
>> char(s - 'A' + 'a')
ans = abcdefghijklmnopqrstuvwxyz

% Dock
>> char(a_string  - 'a' + 'A')
ans = -ATLAB    % notera -

% Mera allmänt
>> upper(a_string)
ans = MATLAB

>> lower(a_string)
ans = matlab
```

Skilj mellan tecknet `'0'` och siffran `0`.

```
>> s = '0':'9'   % tecken
s = 0123456789

>> double(s)     % teckenkoder
ans = 48  49  50  51  52  53  54  55  56  57

>> s - '0'       % siffror
ans = 0   1   2   3   4   5   6   7   8   9
```

Här en strängmatris:

```
>> S = ['a', 'b', 'c'; 'd', 'e', 'f']
S =
abc
def

>> S(1,2)
ans = b

>> Sa = ['abc'; 'def']   % eller
Sa =
abc
def

>> Sa(1,2)
ans = b
```

Man kan utföra numeriska operationer med strängdata, men det är kanske inte så meningsfullt:

```
>> S * S'
ans =
       28814       29696
       29696       30605
```

Värdena svarar inte mot några tecken.

7 Fält och loopar

Nu till kombinationen av fält och loopar. Vi kommer att göra detta mycket kortare så småningom. Man måste dock behärska den klumpiga tekniken också, eftersom det är den enda som står till buds i C och Java t.ex.
Vi kommer nu att se på mer fullständiga program. Det är vanligt att man programmerar på engelska, med engelska variabelnamn och kommentarer. Jag kommer att använda engelska variabelnamn men svenska kommentarer. Normalt skriver jag kommentarerna på engelska också. Eftersom dessa sidor är skrivna i ett stort typsnitt, har jag ibland lite kryptiska variabelnamn för att slippa bryta raderna.
Beräkna längden (normen) av en vektor.

```
v = randn(100, 1);       % 100 värden
summa = 0;
for k = 1:length(v)      % length(v) = 100
   summa = summa + v(k) * v(k);
 end
summa = sqrt(summa);
norm(v) - summa          % 1.7764e-15 i detta fall
```

Beräkna antalet element, i en slumpvektor, som är mindre än en halv (simulering av slantsingling).

```
tosses = rand(100000, 1);
heads  = 0;
for k = 1:length(tosses)
  if tosses(k) < 0.5
    heads = heads + 1;
  end
end
heads   % = 50140 i detta fall
```

Beräkna det största elementet, samt motsvarande rad- och kolonnindex, i en matris.


```
M     = magic(30);  % en magisk kvadrat
max_M = M(1, 1); max_row = 1; max_col = 1; % platsbrist

for row = 1:size(M, 1)
  for col = 1:size(M, 2)
    if M(row, col) > max_M
      max_M   = M(row, col);
      max_row = row;
      max_col = col;
    end
  end
end

max_M               % 900 i detta exempel
max_row             % 30
max_col             % 8
M(max_row, max_col) % blir alltså 900
```

Sortera elementen i en vektor, i växande ordning, med den mycket enkla algoritmen "selection sort".

```
v = [7   6   2   5   8  10   3   4   9   1];
n = length(v);
for j = 1:n-1
  min_val = v(j);
  min_pos = j;
  for k = j+1:n        % för resterande element
    if v(k) < min_val  % hittat mindre?
      min_val = v(k);  % spara värde
      min_pos = k;     % och index
    end
  end
  if min_pos ~= j
    temp_v     = v(j);       % kasta om
    v(j)       = v(min_pos);
    v(min_pos) = temp_v;
  end
end
```



```
>> v
v = 1   2   3   4   5   6   7   8   9  10
```

Omkastningen kan också skrivas

```
    v(min_pos) = v(j);    % kasta om
    v(j)       = min_val;
```

Givet en kvadratisk matris, beräkna diagonal-, rad- och kolonnsummor.

```
n         = 5;
M         = magic(n);     % en magisk kvadrat
row_sums  = zeros(n, 1);  % allokera minne
col_sums  = zeros(n, 1);
diag_sum1 = 0;
diag_sum2 = 0;            % andra huvud-diagonalen

for j = 1:n
  for k = 1:n
    row_sums(j) = row_sums(j) + M(j, k);
    col_sums(j) = col_sums(j) + M(k, j);
  end
  diag_sum1 = diag_sum1 + M(j, j);
  diag_sum2 = diag_sum2 + M(j, n - j + 1);
end

% Detta går också bra
%   row_sums(k) = row_sums(k) + M(k, j);
%   col_sums(k) = col_sums(k) + M(j, k);

>> row_sums'   % transponat för att spara plats
ans = 65    65    65    65    65
>> col_sums'
ans = 65    65    65    65    65
>> diag_sum1
diag_sum1 = 65
>> diag_sum2
diag_sum2 = 65
```

```
>> n^2 * (n^2 + 1) / (2 * n)
ans = 65            % det magiska värdet
```

Nu till det obligatoriska palindromexemplet.

> palindrome n [Gk palindromos running back again, fr. palin back, again + dramein to run; akin to Gk polos axis, pole - more at POLE, DROMEDARY](1629): a word, verse, or sentence (as "a man a plan a canal panama") or a number (such as 1881) that reads the same backward or forward

Sirap i Paris.
Ni talar bra latin.
Bor edra gråvita fat i vår garderob?
A man, a plan, a canal, Panama!
A dog! A panic in a pagoda!
Stressed was I ere I saw desserts.
Eva, can I stab bats in a cave?
Ma is as selfless as I am.
Al lets Della call Ed Stella.
Cigar? Toss it in a can, it is so tragic.
"Naomi, sex at noon taxes," I moan.
Saippuakivikauppias (Finska för "försäljare av kaustik soda").

Låt oss skriva ett program för den enkla varianten, när vi har avlägsnat alla skiljetecken och endast har små bokstäver. Här är några varianter, där jag dessutom introducerar några nya tekniker.


```
str   = 'cigartossitinacanitissotragic';
palin = true;               % logisk variabel
k     = 1;                  % index
n     = length(str);
ndiv2 = n / 2;
np1   = n + 1;

while palin & k <= ndiv2
  if str(k) ~= str(np1 - k)
    palin = false;
  end
  k = k + 1;
end

if palin
  disp('the string is a palindrome')
else
  disp('the string is not a palindrome')
end
```

Här är en mindre strukturerad variant (man skall inte hoppa för mycket i program):

```
str   = 'cigartossitinacanitissotragic';
palin = true;
for k = 1:length(str) / 2
  if str(k) ~= str(end + 1 - k)  % OBS: end
    palin = false;
    break;                       % hoppa ur loopen
  end
end
palin
```


Här en mer komplicerad variant som kan hantera en allmän sträng:

```
str   = 'Cigar? Toss it in a can, it is so tragic.'
left  = 1;           % pekar på vänster bokstav
right = length(str); % pekar på höger bokstav
palin = true;

while palin & left < right
  if ~isletter(str(left))       % inte en bokstav?
    left = left + 1;            % tag nästa
  elseif ~isletter(str(right))  % inte en bokstav?
    right = right - 1;          % tag nästa
  else                          % två bokstäver

    if lower(str(left)) == lower(str(right))
      left  = left + 1;         % tag nästa
      right = right - 1;        % tag nästa
    else
      palin = false;
    end  % if

  end  % if
end  % while
palin
```


8 Funktioner

Det finns en praktisk gräns för hur stora program man kan hantera ("monolithic program"). Man vill stycka upp ett program i mindre delar, i funktioner. Detta har flera fördelar:

- Kan dela upp ett program i mer hanterbara delar. Koden blir enklare att förstå, underhålla och bygga ut. Man behöver inte förstå hela koden i ett svep, utan kan koncentrera sig på de stora dragen.
- Kan återanvända kod (t.ex. **sin(x)**).
- Kan dölja variabler, behöver inte tänka på namnkonflikter (eng. information hiding).
- Kan dölja funktioner (en funktion kan vara lokal till en annan funktion).

Det vi har använt hittills är sk script-filer, som man också kan använda för att dela upp program. Alla variabler är dock globala (åtkomliga från alla andra script-filer) vilket gör det svårt att skriva stora program med script.

En MATLAB-funktion liknar en matematisk funktion, $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$. I programmering brukar $\boldsymbol{x}$ kallas inparameter eller inargument. $\boldsymbol{y}$ kallas utparameter, utargument eller resultat. Man kan strunta i in/ut om det är uppenbart vad som avses.
I programmering finns funktioner som inte tar några inargument, t.ex. **rand()** som returnerar ett slumptal. Notera också att vi kan få olika resultat fastän vi har samma inparameter. Det finns också funktioner som inte tar några argument alls. Anropet, **figure** (eller **figure()**), skapar ett nytt plot-fönster. **figure** kan dock ta argument och returnera värden.
Liksom i matematik är det vanligt att funktioner tar mer än en inparameter. I MATLAB-programmering kan man ha flera in- och utparametrar. MATLABs **eig**-funktion kan t.ex. användas för att beräkna egenpar till det generaliserade egenvärdesproblemet:

$$\boldsymbol{Ax} = \boldsymbol{\lambda Bx}$$

Anropet är `[X, L] = eig(A, B)` (fyra matriser) eller `lambda = eig(A, B)` (två matriser och en vektor) om vi bara behöver egenvärdena.

Ibland anropar vi funktioner utan att tänka på det. När vi löser ett linjärt ekvationssystem skriver vi `x = A \ b` . Alternativt kan man skriva `x = mldivide(A, b)` (matrix-left-divide).

Symbolen \ är knuten till funktionen `mldivide`. När `A` inte är kvadratisk löses problemet i minstakvadratmening.
Vi kan addera två tal genom att skriva `plus(a, b)` eller enklare `a + b`. Det är samma syntax när vi adderar vektorer och matriser. När en operators (t.ex. \) funktion bestäms av datatypen talar vi om operatoröverlagring. Detta är möjligt i t.ex. MATLAB, C++ samt Fortran90, men inte i Java.

8.1 En enkel funktion

Formen på en funktion som tar ett argument och returnerar ett argument är:

```
function ut_parameter = funktions_namn(in_parameter)
 räkna, räkna, räkna ...
ut_parameter = resultat;
```

- Man ger funktionen ett värde, man returnerar ett resultat, genom att tilldela `ut_parameter` ett värde.
- Man måste alltid returnera ett värde (om funktionen har en `ut_parameter`).
- Man sparar funktionen på filen `funktions_namn.m`
- Notera att en funktion inte kommer åt några variabler "utifrån" annat än `in_parameter` (med det vi kan nu).
- Variabler som man använder inuti funktionen är skyddade för åtkomst utifrån. Det enda (med det vi kan nu) sättet för funktionen att returnera ett resultat är via `ut_parameter` (eller via datorgrafik eller utskrifter).

Exempel. Skriv en funktion som givet en matris beräknar och returnerar skillnaden mellan matrisens största och dess minsta element.

```
function max_minus_min = max_diff(A)
max_A = A(1, 1);   min_A = A(1, 1);

for row = 1:size(A, 1)
  for col = 1:size(A, 2)
    max_A = max(max_A, A(row, col));
    min_A = min(min_A, A(row, col));
  end
end
max_minus_min = max_A - min_A;
```

```
>> M = magic(4)
M = 16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> skillnad = max_diff(M)  % kan använda andra namn
skillnad = 15

>> max_diff(1:4)           % resultat -> ans
ans = 3

>> max_diff(-4)
ans = 0

>> max_diff(2:6) + 2 * max_diff(-5:-2:-10)  % uttryck
ans = 12

>> max_diff()  % inget argument, max_diff ger samma
??? Input argument "A" is undefined.

>> max_diff(1:3, 4:5)      % två argument
??? Error using ==> max_diff
Too many input arguments.
```


Observera att namnen på in- och utaparameter är lokala till funktionen. Vi kan använda andra namn när vi anropar funktionen. Om vi inte tillhandahåller en variabel för resultatet, så lagras det i `ans`. Det är dock inte tillåtet att strunta i inparametern eller att ge för många. Vi kan använda funktionen i numeriska uttryck.

En funktion kan ha noll eller flera inparametrar och noll eller flera utparametrar, som i exemplen nedan. Dessa exempel visar enbart hur man använder ett varierande antal parametrar, de är inte tänkta att visa hur man skriver bra kod.

Här en funktion som inte har några parametrar:

```
function date_time1
disp(datestr(now, 'yyyy-mmm-dd, HH:MM:SS'))

>> date_time1()
2009-Nov-19, 18:53:58

>> date_time1
2009-Nov-19, 18:54:00
```

En funktion som har en utparameter.

```
function str = date_time2
str = datestr(now, 'yyyy-mmm-dd, HH:MM:SS');

>> tid = date_time2
tid = 2008-Jan-14, 15:09:40
>> tid
tid = 2008-Jan-14, 15:09:40
```

En funktion som har en inparameter.

```
function date_time3(form)
if form == 'date'
  disp(datestr(now, 'yyyy-mmm-dd'))
elseif form == 'time'
  disp(datestr(now, 'HH:MM:SS'))
end
```



```
>> date_time3('date')
ans = 2009-Nov-19

>> date_time3('time')
ans = 18:56:04
```

En funktion som har två utparametrar.

```
function [d, t] = date_time4
d = datestr(now, 'yyyy-mmm-dd');
t = datestr(now, 'HH:MM:SS');

>> [a, b] = date_time4
a = 2008-Jan-14
b = 15:11:37
```

En funktion som har två inparametrar och två utparametrar.

```
function [d, t] = date_time5(d_form, t_form)
if d_form == 'us'
  d = datestr(now, 'mmm-dd-yyyy');
else
  d = datestr(now, 'yyyy-mmm-dd');
end

if t_form == 'us'
  t = datestr(now, 'HH:MM:SS AM');
else
  t = datestr(now, 'HH:MM:SS');
end

>> [a, b] = date_time5('us', 'us')
a = Jan-14-2008
b = 3:18:34 PM
```

När en funktion tar mer än en inparameter är ordningen viktig; första värdet man skickar in kommer att vara värdet på första inparametern etc.

Namnen är oviktiga i detta sammanhang. Om vi har funktionen
function r = func(a, b)
och gör anropet **res = func(3, 19)** så kommer
a = 3 och **b = 19** inuti funktionen. Det gäller även i följande fall:

```
a   = 19
b   = 3
res = func(b, a)
```

Namnen i anropet och inuti funktionen är inte kopplade på något sätt. Om vi ändrar på **a** eller **b** inuti funktionen så ändras *inte* variabler, med samma namn, utanför funktionen.
Det är dessutom så att parametrarna inte ändras, här ett exempel:

```
>> type func_ex     % lista en funktion

function r = func_ex(a, b)
a = a + 1;
b = 10 * b;
r = a + b;

>> par1 = 20;
>> par2 = 30;
>> res = func_ex(par1, par2)
res = 321

>> par1
par1 = 20  % har inte ändrats
>> par2
par2 = 30  % har inte ändrats
```

Detta beror på att MATLAB analyserar funktionen. Om man i funktionen ändrar på en inparameter, skapar MATLAB en kopia som funktionen arbetar på. Denna kopia avlägsnas när vi återvänder från funktionen.
Att skapa kopior tar tid och minne, så om vi *inte* ändrar på en inparameter i funktionen, så skapar MATLAB inte en kopia av denna inparameter.

Om vi vill ändra på en inparameter brukar man göra som i följande larviga exempel:

```
>> type add_one
function p = add_one(p)  % OBS: samma p
p = p + 1;

>> p = 3
p = 3

>> add_one(p)
ans = 4

>> p = add_one(p)
p = 4
```

MATLAB är lite speciellt i detta avseende. I andra språk har detta lösts på effektivare sätt, man slipper då en (i vissa fall) allokering av temporärt minne och onödiga kopieringar.

Här följer ytterligare några exempel på funktioner.
Exempel. Skriv en funktion som givet en vektor, v, och ett heltal, $m \geq 1$, som argument, beräknar och returnerar en vektor av glidande medelvärden (eng. moving averages) enligt (n är antalet element i v):

$$\frac{1}{m}\left[\sum_{k=1}^{m} v_k,\ \sum_{k=2}^{m+1} v_k, \ldots,\ \sum_{k=n-m+1}^{n} v_k\right]$$

Rutinen skall också returnera det vanliga medelvärdet av v.
Vi noterar att:

$$\sum_{k=j+1}^{j+m} v_k = v_{j+m} - v_j + \sum_{k=j}^{j+m-1} v_k$$


```
>> type mov_aver

function [vec_aver, aver] = mov_aver(v, m)
n = length(v);
if m < 1 | m > n
  error('m < 1 or m > length(v)')
end

vec_aver = zeros(n - m + 1, 1);  % allokera utrymme

s = 0;
for k = 1:m
  s = s + v(k);
end
vec_aver(1) = s;  % första summan

% beräkna resterande summan mha formeln
for j = 1:n - m
  vec_aver(j + 1) = v(j + m) - v(j) + vec_aver(j);
end

vec_aver = vec_aver / m;

% vanliga medelvärdet
aver = 0;
for k = 1:n
  aver = aver + v(k);
end
aver = aver / n;

% En enkel test
>> for m = 1:5
     [vec_aver,aver]=mov_aver(1:5, m); vec_aver', aver
   end

ans = 1     2     3     4     5
aver = 3
```



```
ans = 1.5000   2.5000   3.5000   4.5000
aver = 3

ans = 2     3     4
aver = 3

ans = 2.5000e+00   3.5000e+00
aver = 3

vec_aver = 3
aver = 3
```

■

Exempel. Skriv en funktion, **is_prime(n)**, som avgör om **n** är ett primtal.

```
function result = is_prime(n)
if n <= 0 | round(n) ~= n
  error('n must be a positive integer')
elseif n == 1          % special case
  result = false;
else
  result = true;
  for factor = 2:floor(sqrt(n))
    if rem(n, factor) == 0 % mod is another alternative
      result = false;
      return
    end
  end
end
```

Det finns en färdig rutin, **isprime**, i MATLAB som accepterar allmännare argument och som dessutom kontrollerar storleken på argumentet (tillåter bara $n \leq 2^{32}$). Skriv **type isprime** för att se koden. ■

Funktioner kan anropa andra funktioner. En funktion kan till och med anropa sig själv. Detta kallas rekursion och behandlas i slutet av den större kursen. Här ett exempel där en funktion anropar en annan:

Exempel. Skriv en funktion som approximerar $\int_a^b f(x)\,dx$ med trapetsmetoden (inte en speciellt bra metod).
Trapetsmetoden approximerar integralen med summan av areorna av $n-1$ parallelltrapetser. Med $n \geq 2$ och $h = (b-a)/(n-1)$, $x_k = a+(k-1)h, k = 1, \ldots, n$, så ges approximationen av formeln:

$$\int_a^b f(x)\,dx \approx h\left[\frac{f(x_1)}{2} + f(x_2) + \cdots + f(x_{n-1}) + \frac{f(x_n)}{2}\right]$$

Här kommer filen **trapeze.m**:

```
function I = trapeze(a, b, n)
if n < 2
  error('n must be >= 2.')
end

h  = (b - a) / (n - 1);
I  = 0.5 * (f(a) + f(b));
xk = a;

for k = 2:n-1
  xk = xk + h;
  I  = I + f(xk);
end

I = h * I;
```

och här är filen **f.m** som definierar integranden:

```
function y = f(x)
y = sin(x) / x;
```


Svaret med 16 decimaler är **0.659329906435511833**. Här några testkörningar

```
>> trapeze(1, 2, 2)
ans = 6.480598491103686e-01

>> trapeze(1, 2, 20)
ans = 6.592989203183923e-01

>> trapeze(1, 2, 200)
ans = 6.593296239739588e-01

>> trapeze(1, 2, 2000)
ans = 6.593299036362444e-01
```

■

Vår rutin vore mer användbar om funktionen inte alltid måste heta **f**. Det fixar vi i de två följande avsnitten.

8.2 Funktionshandtag

Om man har enkla funktioner (inga for-satser etc.) finns det några kortvarianter. Vi tar upp den viktigaste varianten, *anonyma funktioner* (eng. anonymous functions). Det generella utseendet är

```
  funktionshandtag =
     @(kommaseparerad lista av argument) uttryck
```

Listan får vara tom eller bestå av endast ett argument. Här några exempel:

```
>> f = @(x) exp(-x^2);
>> f(1)
ans = 3.6788e-01

>> f(sqrt(-1))
ans = 2.7183e+00
```



```
>> f
f = @(x) exp(-x^2)

>> class(f)
ans = function_handle

>> minmax = @(x) [min(x), max(x)];

>> minmax(1:4)
ans = 1     4

>> x = linspace(0, 1);
>> minmax(sin(x) .* cos(x))  % elementvis *
ans = 0   4.9999e-01

>> two_var = @(x, y) exp(-x^2 - y);
>> two_var(1, 2)
ans = 4.9787e-02

>> a = 10;
>> note = @(x) x + a;
>> note(1)
ans = 11

>> a = 10000;
>> note(1)
ans = 11      ! OBS, a = 10

>> time = @() datestr(now, 'HH:MM:SS');
>> time()
ans = 21:15:10
>> time()
ans = 21:15:11

>> time  % dock, dvs utan ()
time = @() datestr(now, 'HH:MM:SS')
```


Man kan även skapa ett funktionshandtag på följande vis:

```
  funktionshandtag = @funktionsnamn
```

Ett funktionshandtag fungerar som en variabel (i flera fall) och kan lagras i en cellvektor (som vi kommer till så småningom) t.ex.

```
% Define test function

if test == 1
  test_f = @cos;  % Nytt
elseif test == 2
  test_f = @sin;
elseif test == 3
  test_f = @(x) exp(-x) * x;
end
```

Med **test = 1;** blir **test_f(pi)** lika med -1.
Snyggare är att använda cellvektorer:

```
% inga blanka
>> test_funcs = {@cos, @sin, @(x)exp(-x)*x}
test_funcs =
    @cos    @sin    @(x)exp(-x)*x

>> for k = 1:3
     test_funcs{k}(pi / 2)
   end

ans = 6.1232e-17
ans = 1
ans = 3.2654e-01
```

8.3 Funktioner som parametrar

Det är mycket vanligt i tillämpningar att skicka funktioner som parametrar. Fördelen är att användaren av t.ex. en integrationsrutin kan välja ett eget namn på integrandrutinen, namnet är inte bestämt av den som skrivit integrationsrutinen. Här följer vår förbättrade trapetsrutin, som nu även tar en funktion som parameter. Det enda vi behöver är första raden i rutinen, som nu lyder:

```
function I = trapeze(f, a, b, n)
```

Nu några tester:

```
>> trapeze(@f, 1, 2, 2000)
ans = 6.593299036362444e-01

>> trapeze(@my_integrand, 1, 2, 2000)
ans = 1.352572717657960e-01

>> type my_integrand

function y = my_integrand(x)
y = exp(-x^2);
```

Det går bra med anonyma funktioner också:

```
>> handtag = @(x) exp(-x^2);

>> trapeze(handtag, 1, 2, 2000)
ans = 1.352572717657960e-01

>> trapeze(@(x)exp(-x^2), 1, 2, 2000)
ans = 1.352572717657960e-01
```


8.4 Globala variabler

En litet bekymmer med att använda standardiserad programvara är att funktionen vi skickar (integranden) har en given parameterlista, bestämd av den som skrivit integrationsrutinen. Det gör att man inte utan vidare kan bifoga data till sin rutin, data som kanske behövs för att beräkna funktionsvärdena.
Ett vanligt sätt att lösa denna typ av problem är att använda globala variabler. Denna teknik skall inte missbrukas, eftersom det lätt leder till oläslig kod.
Säg att vi har ett huvudprogram, en scriptfil:

```
global pressure temperature  % deklarera först

pressure    = 1.23e6;        % definiera sedan
temperature = 237.55;

I = trapeze(@a_function, -23.56, 18.54, 1000)
```

Här följer funktionen:

```
function y = a_function(x)
global pressure temperature

% här kan vi använda pressure och temperature
% för att beräkna funktionsvärdet
y = ...
```

En fördel med ovanstående lösning är att vi ändrar värdet på t.ex. **pressure** på *ett* ställe. Om man skriver in värdet, **1.23e6**, både i huvudprogram och funktion glömmer man kanske att ändra i funktionen när man senare ändrar i huvudprogrammet.

8.5 Flera funktioner i samma fil

När man skriver större program blir det lätt många funktioner och därmed filer att hålla reda på. I vissa fall kan man lagra flera funktioner i samma fil. Detta exempel visar på strukturen. Funktionerna är lagrade på filen **prime.m** (i detta fall).

```
function y = prime(x)  % primary function
                       % file name  prime.m
 ...
function [z, y] = another_function(x)  % a sub function
 ...
function out = a_third_function(x)  % a sub function
 ...
```

Endast den primära funktionen är synlig utifrån (kan anropas av funktioner som *inte* ligger i filen). Funktionerna i filen kan anropa varandra. Om det finns en funktion på en egen, separat, fil med samma namn som en subfunktion (**sub** säg), och en funktion i filen anropar **sub**, så väljs subfunktionen.

Det går att driva detta ett steg till. Man kan ha funktioner som är lokala till andra funktioner (eng. nested functions). Det finns också "private functions" (som ligger i en katalog **private**). Dessa funktioner kan anropas från en funktion en nivå ovanför den privata katalogen. Slutligen har vi överlagrade funktioner (MATLAB har visst stöd för objektorienterad programmering). Vi tar inte upp någon av dessa varianter i denna kurs.

9 Avlusning

En stor del av en programmerares tid brukar upptas att hitta och korrigera fel i program. Denna process kallas avlusning (eng. debugging). Felen beror oftast på programmeraren, men kan även orsakas av fel i kompilatorer, programsystem som MATLAB och kanske av hårdvaran. I större programmeringsprojekt skriver en programmerare 10-20 korrekta rader per dag. Antalet beror givetvis på kodkvalitet, tillämpning, språk etc.

MATLAB är rätt tacksamt när det gäller att hitta fel, C++-program är *mycket* svårare att felsöka (index- och pekarfel fångas normalt inte). Det finns speciella program, avlusare (eng. debuggers) som kan underlätta felsökandet (för erfarna programmerare).

Det vanligaste felet för nybörjaren är "språkfel" (felaktig syntax).

```
>> si = sin[0.1]   % fel typ av parenteser
?? si = sin[0.1]
           |
Error: Unbalanced or misused parentheses or brackets.
```

Dessa är normalt lätta att åtgärda. Mycket svårare är logiska fel och då speciellt sådana som inte orsakar något felmeddelande utan bara ett felaktigt resultat (om man nu får ett felaktigt resultat när man testar). Programmet kan ju vara felaktigt, i stora delar, men producera korrekta resultat i några enkla testfall. *Om* man får ett felmeddelande, är man (kanske) i en bättre situation.

```
>> x = a(n);   % indexfel, varför?
??? Index exceeds matrix dimensions.
```

Arbeta baklänges i programmet. Varför fick vi indexfel. Är vektorn för kort eller är **n** för stort? Kanske har vi fel variabelnamn.

Om programmet avslutar normalt, men med fel resultat: Kontrollera indata, GIGO, "Garbage In Garbage Out".
Skriv ut resultat under exekveringen (tag bort semikolon).

Ibland har vi facitvärden och kan jämföra, om inte, kan vi kanske göra en rimlighetskontroll. Har vi ett testproblem där vi känner resultatet?
Lägg in kontroller. Använd gärna MATLABs, **warning**, som ger en "traceback". Finns även **error**, som avbryter körningen. Kan även använda **return** för att avbryta körningen, och bara testa en del av koden.

```
>> type warn_ex

function warn_ex
% code ...
  another_func(2, 3)
% code ...

function another_func(x, y)
% code ...
if x^2 + y^2 > 1   % should not happen
  warning('x^2 + y^2 > 1, x^2 + y^2 = %e', x^2 + y^2)
end
% code ...

>> warn_ex
Warning: x^2 + y^2 > 1, x^2 + y^2 = 1.300000e+01
> In warn_ex>another_func at 9
  In warn_ex at 3
```

Man kan lägga in sådana kontroller när man skriver programmet, "defensive programming". Om en kontroll är tidskrävande slår man på den med hjälp av en debug-flagga.

```
  if debug_expensive
    % Time consuming computation

    if cannot_happen
      warning( ...
    end

  end
```


Murphys lag:

> If there's more than one possible outcome of a job or task, and one of those outcomes will result in disaster or an undesirable consequence, then somebody will do it that way.

Programmera för "alla" eventualiteter (åtminstone om någon annan än du själv skall använda koden).
Ibland undrar man om en viss funktion ens exekveras. När det gäller if-satser undrar man ibland om något/vilket alternativ som utförs. Man kan ha formulerat något villkor fel eller glömt något. Man kan göra en egen trace, t.ex.:

```
function func1(a)
disp('*** in func1')

if a < 0
  disp('*** func1: a < 0')
  ... code
else
  disp('*** func1: a >= 0')
  ... code
end
```

Man kan också lägga in utskrifter som talar om var man befinner sig i koden:

```
disp('x before solve')
x = A \ b
 ...
disp('x before correction')
x = x + corr
```

Det vanligaste felet (åtminstone i traditionella språk) är nog indexfelet (i vektorer och matriser). Index kan vara otillåtet (då får vi ett felmeddelande, i MATLAB i alla fall) eller tillåtet men felaktigt.

Exempel. Tag ut de sista 7 elementen ur vektorn **vec**.

```
last_elem = vec(end-7:end); % tar ut 8 element
```

vec(m:n) med **m** $\leq$ **n** tar ut **m - n + 1** element, inte **m - n** element. ■
Ett mycket vanligt fel i traditionella språk att utföra en loop en gång för mycket/litet. Det är inte lika vanligt i MATLAB pga av MATLABs kraftfulla matrishantering.
Felet har ett eget namn: "off-by-one error (oboe)". Ett typiskt fall är räknare i samband med while-loopar:

```
ind = 0;       % eller 1
while ind < n  % eller ind <= n
  ind = ind + 1;
  vec(ind) ...

% eller

  vec(ind) ...
  ind = ind + 1;
end
```

Testa första samt sista iterationen. Om en loop skall gå ett givet antal varv brukar det vara mindre risk för fel med for-loopar.

Ett vanligt fel i numerisk analys (t.ex. i trapetsformeln) är följande:

Exempel. Vi delar in intervallet, $[a, b]$, i en uppsättning lika långa delintervall, (x_j, x_{j+1}). Intervallernas ändpunkter är $a = x_1 < x_2 < x_3, \ldots < x_n = b$. Hur långt är varje delintervall? Vanliga svar: $(b-a)/(n-1), (b-a)/n, (b-a)/(n+1)$.
Lösning: tänk specialfall (som är tillräckligt allmänt, så att det täcker upp alla fall). Med $n = 3$ har vi $a = x_1 < x_2 < x_3 = b$. Tre x-värden ger två intervall (ett mindre), så rätt svar är $(b-a)/(n-1)$.

Eftersom man har olika numrering i olika sammanhang kan det bli ännu mer förvirrat: $a = x_0 < x_2 < x_3, \ldots < x_{n-1} = b$, $a = x_0 < x_2 < x_3, \ldots < x_{n+1} = b$ etc. ■

Avlusning av funktioner. Kontrollera in- och utparametrar vad avser ordningsföljd, antal och typ. Variabler som är lokala till en funktion är lite besvärliga att inspektera. **keyboard**-kommandot kan vara praktiskt.

```
  function ...

    keyboard
% återvänd med att skriva return (sex tecken)
```

Man stannar på **keyboard**-raden och får tillgång till variablerna i funktionen. Prompten byts till **K»**. Man kan utföra beräkningar, plotta etc. på vanligt sätt. För att fortsätta exekveringen skriver man **return** (man skriver verkligen strängen). Om man har en loop som anropar **keyboard** och vill avsluta i förtid, skriver man **dbquit**, exekveringen av m-filen avslutas då.

Några avslutande ord om avlusning:

> I really hate this damned machine
> I wish that they would sell it.
> It never does quite what I want
> But only what I tell it.
> anon.

10 Testning av program

> A bug-free program is an abstract, theoretical concept. Dennie Van Tassel
>
> All sufficiently complex programs have bugs. Anonymous
>
> Program testing can be used to show the presence of bugs, but never to show their absence! Edsger Dijkstra
>
> An effective way to test code is to exercise it at its natural boundaries. Brian Kernighan

Man har givetvis testat sitt program under avlusningsfasen, men större system kräver en speciell testningsfas. Här följer några saker att tänka på. En bra och utförligare beskrivning finns hos Wikipedia, sök efter "Software testing".

Vi tänker oss ett relativt litet MATLAB-program som utför numeriska beräkningar (att testa ett program som Firefox är ett helt annat företag). Man brukar välja att testa slumpdata, men effektivare är att förfara som Brian Kernighan föreslår. Man måste t.ex. kunna sortera en vektor som har *ett* element, kunna lösa ett linjärt ekvationssystem, $\boldsymbol{Ax} = \boldsymbol{b}$, där $\boldsymbol{A}$ är en 1×1-matris. Kontrollera resultat av loopar som går noll eller ett varv. Blir alla variabler definierade fastän villkoret i en while-loop aldrig blir uppfyllt?

```
for k = m:n  % problem om m > n?
 ...
end

while villkor
  x = ...
end

här använder vi x
```


När man testar numeriska rutiner kan man i viss utsträckning använda slumpdata, men slump-problem behöver inte vara typiska ($\boldsymbol{Ax} = \boldsymbol{b}$ från PDE) och de brukar inte vara speciellt svåra. Man kan normalt tänka ut svårare testproblem.

Exempel. Vi vill testa norm-rutinen (för vektorer) i MATLAB och några MATLAB-cloner.
Octave, **http://www.gnu.org/software/octave**.
Scilab, **http://www.scilab.org**.

```
>> x = rand(2, 1)          % I Matlab
x =
   9.2217e-01
   5.1609e-01
>> norm(x) - sqrt(sum(x.^2))
ans = 2.2204e-16

octave:8> x = rand(2, 1)   % I Octave
x =
  0.94248
  0.78092

octave:9> norm(x) - sqrt(sum(x.^2))
ans = 0

--> x = rand(2, 1)         % I Scilab
 x  =
    0.2113249
    0.7560439

-->norm(x) - sqrt(sum(x.^2))
 ans  = 0.
```

Så inga problem. Problem kan det dock bli om man känner till programmens svaga punkter.


```
>> norm(1e200)
ans = 1.0000e+200
>> norm(1e-200)
ans = 1.0000e-200

octave:11> norm(1e200)
ans = Inf
octave:12> norm(1e-200)
ans = 0

-->norm(1e200)
 ans  = Inf
-->norm(1e-200)
 ans  = 0.
```

Så varken Octave eller Scilab klarar av att räkna ut normen av vektorer med stora eller små tal, fastän det inte är några egentliga problem. I exemplet har dessutom vektorn bara ett element. Både Octave och Scilab använder sannolikt en algoritm enligt

```
norm(x): sqrt(sum(x.^2))
```

Problemet är att **1e200^2** ger overflow och **1e-200^2** underflow, så MATLAB använder en listigare algoritm som skalar indata. ■

Exempel. Sinusberäkning för stora argument. Facit med Maple:

$$\sin 10^{20} \approx -0.64525128526578084420581171131 3$$

```
>> sin(1e20) + 0.64525128526578084420581171131 3
ans = 0

octave:25> sin(1e20) + 0.64525128526578084420581171131 3
ans =  -1.01670605993712e-01

-->sin(1e20) + 0.64525128526578084420581171131 3
 ans  = - 1.016706059937121E-01
```

Slumpargument ger sannolikt inga problem alls. ■

Exempel. Här ett sista test med determinanter:

```
>> det(diag([1e-200 1e-200 1e200 1e200]))
ans = 1.0000e-00

octave:41> det(diag([1e-200 1e-200 1e200 1e200]))
ans =  1.0000e+00

-->det(diag([1e-200 1e-200 1e200 1e200]))
 ans  = 0.000E+00     % OBS, noll
```

Om man kastar om ordning med de stora talen först ger Scilab **INF**.
■

Man bör försöka testa alla delar av ett program. I en enkel test utnyttjar man kanske bara en delmängd av de funktioner man skrivit och en del av alternativen i if-satser.

11 Dokumentation

Större program och sådana som används av andra än programmeraren måste dokumenteras (dock brukar även programmeraren glömma detaljer). Det skall sägas att min föreläsningsexempel inte är så väldokumenterade, eftersom typsnittet gör att jag för begränsa antalet tecken. Det blir också lite svårläst när har begränsat utrymme. Dessutom fungerar jag, förhoppningsvis, som en levande kommentar under föreläsningen.

Man dokumenterar av flera anledningar.

- När man utvecklar program, man glömmer snabbt.
- Program som har en livslängd på flera år kommer att modifieras av någon (dig eller andra).
- För att förstå vad program gör. För att kunna återanvända kod.

Typ av dokumentation:

- Kommentarer i koden, ett minimum.
- Manualblad (i en unix-miljö).
- I större programsystem, on-line-hjälp, html-sidor, manualer.

Vi koncentrerar oss på kommentarer i koden.

- Inledande kommentarer. Vad gör funktionen. Beskrivning av in- och ut-variabler. Begränsningar. Felmeddelanden. Programmerare, version, datum. Utnyttja funktionen hos MATLABs **help**-kommando.
- Kommentarer i koden.

help my_func listar de inledande kommentarerna i koden (fram till första ickekommentar).

```
function res = my_func(arg)
% This ....    listas
%              listas
n = length(arg);         % listas EJ
%                          listas EJ
```


Här ett hoptryckt exempel på kommentarer i början av en Fortran-funktion:

```
NAME
  DPOTRF - compute the Cholesky factorization of a real
           symmetric positive definite matrix A

SYNOPSIS
  SUBROUTINE DPOTRF( UPLO, N, A, LDA, INFO )
    CHARACTER      UPLO
    INTEGER        INFO, LDA, N
    DOUBLE         PRECISION A( LDA, * )

PURPOSE
  DPOTRF computes the Cholesky factorization of a real
  symmetric positive definite matrix A.
  The factorization has the form
     A = U**T * U,   if UPLO = 'U', or
     A = L  * L**T,  if UPLO = 'L',
  where U is an upper triangular matrix and L is
  lower triangular.

  This is the block version of the algorithm, calling
  Level 3 BLAS.


ARGUMENTS
  UPLO  (input) CHARACTER*1
        = 'U':  Upper triangle of A is stored;
        = 'L':  Lower triangle of A is stored.

  N     (input) INTEGER
        The order of the matrix A.  N >= 0.

  A     (input/output) DOUBLE PRECISION array, dimension
        (LDA,N). On entry, the symmetric matrix A. If
        UPLO = 'U', the leading N-by-N upper triangular
        part of A contains the upper triangular part
```



```
        of the matrix A, and the strictly lower triangula
        part of A is not referenced. If UPLO = 'L', the
        leading N-by-N lower triangular part of A
        contains the lower triangular part of the matrix
        A, and the strictly upper triangular part
        of A is not referenced.

        On exit, if INFO = 0, the factor U or L from the
        Cholesky factorization A = U**T*U or A = L*L**T.

  LDA   (input) INTEGER
        The leading dimension of the array A.
        LDA >= max(1,N).

  INFO  (output) INTEGER
        = 0:  successful exit
        < 0:  if INFO = -i, the i-th argument had an
              illegal value
        > 0:  if INFO = i, the leading minor of order
              i is not positive definite, and the
              factorization could not be completed.
VERSION
  LAPACK version 3.0, 15 June 2000
```

I vissa fall lägger man namnet på programmeraren och en versionshistorik.
När man dokumenterar kod skall man inte bara översätta koden till naturligt språk.

```
% Increase k by 2
k = k + 2;
```

är en meningslös kommentar, en person som läser koden kan ju normalt programmera. En bättre kommentar kan se ut så här (där **...** är en lämplig fortsättning).

```
% Take care of the even case. We don't have to
% consider odd numbers, since ...
k = k + 2;
```


Förklara vad ett programsegment gör. Enklare algoritmer och datastrukturer beskrivs i koden, mer komplicerade kräver separat dokumentation.
Man kan få gratisdokumentation genom att använda vettiga variabelnamn, t.ex. **begin_word**, **end_word**, **BeginWord**, **EndWord**, **WordBegin**, **WordEnd**,

```
function words = SentenceToWords(sentence)
 ...
function words = sentence_to_words(sentence)
 ...
function words = sentence2words(sentence)
 ...
function words = FindWords(sentence)
 ...
function words = GetWords(sentence)
```

Alltför långa variabelnamn blir svårlästa. Konsonanter är viktigare än vokaler (i slutet av namn).

```
function words = Extrct_Wrds(sentence)
 ...
function words = strng2wrds(string)
 ...
```

Det är bra om man är konsekvent vid namngivning. Jag brukar använda stora bokstäver för matriser och små för vektorer och skalärer t.ex.

Var *väldigt* försiktig med att använda tecken som kan förväxlas.

```
0, o, O, Q   (noll, bokstäver)
1, l, i, I   (ett, bokstäver)
```

Tänk också på att programmet kanske läses med ett annat typsnitt, där det är mindre (ingen) skillnad mellan dessa bokstäver. Om man använder ett typsnitt utan serifer (sans-serif) brukar det vara liten skillnad mellan tecknen.
Här ett exempel i Helvetica: 0, o, O, Q, 1, l, i, I.

12 Mer om vektorer och matriser, vektorisering

Vi har redan sett en del vektorer och matriser, men MATLAB kan mycket mer. En del av följande operationer definieras inte i kursen i linjär algebra. Skilj på matematik och programspråk. I ett programspråk kan man implementera bekväm notation som inte har en direkt motsvarighet i matematik. Använd *inte* syntax från MATLAB i matematik. I MATLAB kan man t.ex. skriva `x = A \ b`, vänsterdivision med matris, men man får *inte* skriva

$$x = \frac{b}{A}$$

i en matematiskt text.

Vi börjar med några *elementvisa* operationer.

```
>> a = 1:3;   % lite data
>> b = 4:6;

>> a .* b
ans = 4    10    18

>> a ./ b
ans = 2.5000e-01   4.0000e-01   5.0000e-01

>> a .\ b
ans = 4.0000e+00   2.5000e+00   2.0000e+00
```

Man skall lära sig att använda punkter där de behövs. Skriv *inte* `c = a + 2 .* b`. Varför?

1. Den som läser koden får intrycket av programmeraren inte vet att multiplikation med skalär *definitionsmässigt* utförs elementvis. Det ser amatörmässigt ut.

2. Om det hade stått `c = a + v .* b;` hade jag tänkt "`v` måste vara en vektor, eftersom man använder elementvis multiplikation". Koden "ljuger" alltså.
3. Det kan introducera fel. Uttryck kan bli definierade som med korrekt syntax hade varit odefinierade. Se exempel nedan.

Om man avlägsnar punkterna från divisionerna får man fortfarande resultat. Den första kan tolkas i termer av minstakvadratproblem $\min_\xi ||b\,\xi - a||_2$.

```
>> a / b
ans =
   4.1558e-01

>> a \ b           % ingen uppenbar tolkning
ans =
            0            0            0
            0            0            0
   1.3333e+00   1.6667e+00   2.0000e+00
```

Nu till några intressantare operationer:

```
>> a .^ b
ans = 1    32   729

>> ones(size(a)) ./ a   % invertera
ans =
   1.0000e+00   5.0000e-01   3.3333e-01

>> size(a)
ans = 1     3

>> 1 ./ a               % kortare
ans =
   1.0000e+00   5.0000e-01   3.3333e-01
```



```
>> a ^ 2               % samma som a * a
??? Error using ==> mpower
Matrix must be square.

>> a .^ 2
ans = 1     4     9

>> sum(a)  % även sum(a') blir 6
ans = 6

>> sqrt(sum(a.^2))  % Normexemplet
ans = 3.7417

>> a.^2 .* b + 1
ans = 5    21    55

>> 1:3+2
ans = 1     2     3     4     5

>> (1:3)+2
ans = 3     4     5
```

De sista operationerna visar hur en skalär kan ta rollen av en vektor (är inte tillåtna i linjär algebra-kursen).
Här ett exempel som visar att man kan få fel resultat om man alltid skriver punkter.
Säg att man hade tänkt att bilda `c = a + s * b;` där `s` är en skalär men att `s` har råkat bli en vektor:

```
>> s = 10:10:30     % fel
s =
    10    20    30

% många rader senare ...

>> c = a + s .* b   % oavsiktligt definierat
c =
    41   102   183
```



```
% Hade vi använt korrekt syntax, så:

>> c = a + s * b    % ej definierat. Aha, ett fel!
??? Error using ==> mtimes
Inner matrix dimensions must agree.
```

Detta är nu, i ärlighetens namn, inte bombsäkert:

```
>> b = b'          % misstag, säg
b =
     4
     5
     6

>> c = a + s * b  % vad är det som händer?
c =
   321   322   323
```

Säg att vi vill skapa en tabell av sinus-värden när vinkeln varierar mellan 0 och 90 grader i steg om 10 grader. Vi kan klara oss utan loopar eftersom sinus accepterar vektorargument och returnerar en vektor av sinus-värden.

```
>> v = 0:10:90
v =
   0  10  20  30  40  50  60  70  80  90

>> sin((pi / 180) * v)  % pi är fördefinierat
ans =
  Columns 1 through 8
         0    0.1736    0.3420    0.5000    0.6428    0.7
  Columns 9 through 10
    0.9848    1.0000
```

`cos`, `sqrt`, `exp`, `log` (finns ej `ln`), `log10` etc. arbetar också elementvis.

12.1 Indexering

Det finns flera sätt att indexera sig i vektorer, dels det vanliga, **vektor(uttryck)** där uttrycket evalueras till ett positivt heltal. Man har ofta anledning att känna storleken på en vektor och att hitta sista elementet i en vektor. Här några varianter:

```
>> v = (1:6)+0.1  % lite data
v = 1.1000  2.1000  3.1000  4.1000  5.1000  6.1000

>> n = length(v)
n = 6

>> size(v)
ans = 1     6

>> size(v, 2)
ans = 6

>> size(v')      % OBS transponat
ans = 6     1

>> size(v', 1)
ans = 6

>> n = size(v, 2);
>> v(n)
ans = 6.1000

>> v(end)
ans = 6.1000

>> v(end - 4)
ans = 2.1000
```

Observera att **end** har en annan betydelse när det inte används som index. Man har ofta anledning att arbeta på delar av vektorer. Här några exempel:


```
>> v(2:3)
ans = 2.1000    3.1000

>> v23 = v(2:3)
v23 = 2.1000    3.1000

>> v41 = v(4:-1:1)
v41 = 4.1000    3.1000    2.1000    1.1000

>> v(2:3) + v(3:-1:2)
ans = 5.2000    5.2000

>> vz = v(4:1:1)
vz = Empty matrix: 1-by-0
```

Det är vanligt att arbeta på, eller extrahera element som uppfyller logiska villkor. Man kan då indexera sig med logiska vektorer. Ett index ett (true) plockar ut ett element, men index noll (false) ignorerar elementet.

```
>> pick = v > 4
pick = 0     0     0     1     1     1

>> v(pick)
ans = 4.1000    5.1000    6.1000

>> v(v > 4)
ans = 4.1000    5.1000    6.1000

>> v(v > 10)
ans = Empty matrix: 1-by-0

>> v([0 0 0 1 1 1])
??? Subscript indices must either be real
    positive integers or logicals.

>> v(logical([0 0 0 1 1 1]))       % typkonvertering
ans = 4.1000    5.1000    6.1000   % ovanligt
```



```
>> v([false false false true true true]) % ovanligt
ans = 4.1000    5.1000    6.1000

>> v(v < 2 | v > 5)
ans = 1.1000    5.1000    6.1000

>> v(v.^2 <= 6)
ans = 1.1000    2.1000
```

Det finns två logiska funktioner som reducerar en vektor av logiska element till ett enda.

```
>> v > 0
ans = 1     1     1     1     1     1

>> all(v > 0)
ans = 1

>> all(v > 2)
ans = 0

>> any(v > 2)
ans = 1
```

För att undvika missförstånd kan detta användas i if-satser:

```
>> if v > 2
     disp('true')  % all values must be true
   else
     disp('false')
   end
false
```

är samma som

```
>> if all(v > 2)
     disp('true')
   else
     disp('false')
   end
```


Vill man att then-delen skall utföras om minst ett värde är större än två skriver man

```
>> if any(v > 2)
   etc
```

Ibland behöver man de index i vektorn för vilka elementen uppfyller ett visst villkor. Vi kan då använda funktionen **find**, men först en primitiv variant:

```
>> ind = 1:length(v);
>> ind(v < 2 | v > 5)
ans = 1     5     6

>> find(v < 2 | v > 5)
ans = 1     5     6
```

Vissa funktioner returnerar index:

```
>> r = rand(1, 5)
r =
    0.5343    0.7448    0.8109    0.4732    0.0698

>> [min_r, ind] = min(r)
min_r = 0.0698
ind = 5

>> [max_r, ind] = max(r)
max_r = 0.8109
ind = 3

>> [sort_r, ind] = sort(r)
sort_r = 0.0698    0.4732    0.5343    0.7448    0.8109
ind = 5     4     1     2     3

>> r(ind)
ans = 0.0698    0.4732    0.5343    0.7448    0.8109
```

min hittar första minimum. Hur hittar vi index för alla, om det finns kopior?

```
>> v
v = 3     1     2     6     1     5     4     1

>> [min_v, ind] = min(v)
min_v = 1
ind = 2

>> ind = find(v == min(v))  % eller min_v
ind = 2     5     8

>> v(ind)
ans = 1     1     1
```

Några gamla exempel med kortare lösningar:

```
>> tosses = rand(100000, 1);
>> heads  = sum(tosses < 0.5)
heads = 50359

>> s = 'Cigar? Toss it in a can, it is so tragic.';
>> letters = s('a'<=s & s<='z' | 'A'<=s & s<='Z')
% eller   letters = s(isletter(s))
letters = CigarTossitinacanitissotragic

>> low_case = lower(letters)
low_case = cigartossitinacanitissotragic
>> palin = all(low_case == low_case(end:-1:1))
palin = 1
```

12.2 Nu till matrisfallet

```
>> A = [1 2; 3 4];
>> B = [3 4; 1 2];
>> A * B
ans =
     5     8
    13    20
```



```
>> A + B
ans =
     4     6
     4     6

>> A .* B
ans =
     3     8
     3     8

>> A ./ B
ans =
   3.3333e-01   5.0000e-01
   3.0000e+00   2.0000e+00

>> A .\ B
ans =
   3.0000e+00   2.0000e+00
   3.3333e-01   5.0000e-01

>> A / B   % ungefär A * inv(B)
ans =
     0     1
     1     0

>> A \ B   % ungefär inv(A) * B
ans =
  -5.0000e+00  -6.0000e+00
   4.0000e+00   5.0000e+00

>> A^2
ans =
     7    10
    15    22
```



```
>> A.^2
ans =
     1     4
     9    16

>> A.^A
ans =
     1     4
    27   256

>> sqrt(A)
ans =
   1.0000e+00   1.4142e+00
   1.7321e+00   2.0000e+00

>> sqrt(-A)
ans =
           0 + 1.0000e+00i           0 + 1.4142e+00i
           0 + 1.7321e+00i           0 + 2.0000e+00i

>> R = rand(3)
R =
   9.5013e-01   4.8598e-01   4.5647e-01
   2.3114e-01   8.9130e-01   1.8504e-02
   6.0684e-01   7.6210e-01   8.2141e-01

>> R = rand(3, 2)     % rand
R =
   4.4470e-01   9.2181e-01
   6.1543e-01   7.3821e-01
   7.9194e-01   1.7627e-01

>> R = randn(3, 2)    % OBS randN
R =
  -1.9790e-02   2.5730e-01
  -1.5672e-01  -1.0565e+00
  -1.6041e+00   1.4151e+00
```



```
>> D = diag(1:2:5)
D =
     1     0     0
     0     3     0
     0     0     5

>> diag(D)  % inte bara för diagonalmatriser
ans =
     1
     3
     5

>> D = diag(1:2:5, -1) + diag(1:2:5, 1)
D =
     0     1     0     0
     1     0     3     0
     0     3     0     5
     0     0     5     0

>> I = eye(3)
I =
     1     0     0
     0     1     0
     0     0     1

>> B = magic(3)
B =
     8     1     6
     3     5     7
     4     9     2

>> IB = inv(B)
IB =
   1.4722e-01  -1.4444e-01   6.3889e-02
  -6.1111e-02   2.2222e-02   1.0556e-01
  -1.9444e-02   1.8889e-01  -1.0278e-01
```

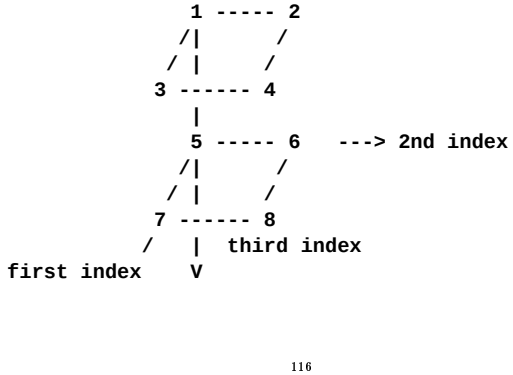
```
>> B * IB
ans =
   1.0000e+00            0  -1.1102e-16
  -2.7756e-17   1.0000e+00            0
   6.9389e-17            0   1.0000e+00

>> IB * B
ans =
   1.0000e+00            0  -2.7756e-17
            0   1.0000e+00            0
            0   1.1102e-16   1.0000e+00

>> ones(2, 3)
ans =
     1     1     1
     1     1     1

>> zeros(2)
ans =
     0     0
     0     0

>> S = reshape(1:6, 2, 3)
S =
     1     3     5
     2     4     6

>> sum(S)
ans =
     3     7    11

>> sum(S')
ans =
     9    12
```

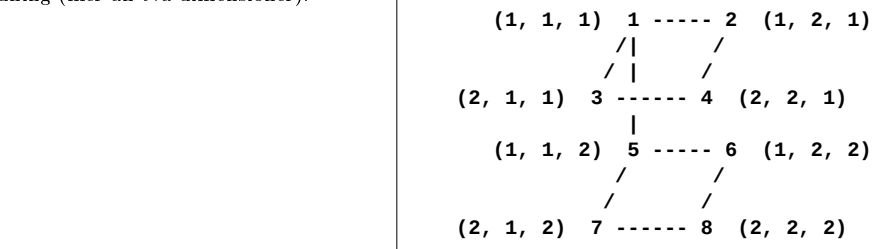


```
>> sum(S, 2)
ans =
     9
    12

>> sum(sum(S))
ans =
    21

>> cumsum(1:7)
ans =
     1     3     6    10    15    21    28

>> M = magic(3)
M =
     8     1     6
     3     5     7
     4     9     2

>> sort(M)
ans =
     3     1     2
     4     5     6
     8     9     7

>> M(:)'
ans =
     8     3     4     1     5     9     6     7     2

>> s = sort(ans)
s =
     1     2     3     4     5     6     7     8     9
```


Det finns matriser av högre ordning (mer än två dimensioner):

```
>> A1 = [1 2;3 4]
A1 =
     1     2
     3     4

>> A2 = [5 6; 7 8]
A2 =
     5     6
     7     8

>> A(:,:,1) = A1;
>> A(:,:,2) = A2;

>> A
A(:,:,1) =
     1     2
     3     4
A(:,:,2) =
     5     6
     7     8
```






```
>> M = magic(5)
M =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> M(:, 2)
ans =
    24
     5
     6
    12
    18

>> M([2 5], :)
ans =
    23     5     7    14    16
    11    18    25     2     9

>> M([2 5], [2 4])
ans =
     5    14
    18     2
```

```
>> M(:, end)
ans =
    15
    16
    22
     3
     9

>> M(end, :)
ans =
    11    18    25     2     9

>> M(end, end)
ans = 9

>> M([1 3], [end-3:end])
ans =
    24     1     8    15
     6    13    20    22

>> M(1:2, 3:4) = M([2 5], [2 4])
M =
    17    24     5    14    15
    23     5    18     2    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Alternativt:

```
>> [m, n] = size(M)
m = 5
n = 5

>> M(m, :)
ans = 11    18    25     2     9
```



```
>> T = triu(magic(4))
T =
    16     2     3    13
     0    11    10     8
     0     0     6    12
     0     0     0     1

>> T(:, 4:-1:1)
ans =
    13     3     2    16
     8    10    11     0
    12     6     0     0
     1     0     0     0

>> T(4:-1:1, :)
ans =
     0     0     0     1
     0     0     6    12
     0    11    10     8
    16     2     3    13

>> T(4:-1:1, 4:-1:1)
ans =
     1     0     0     0
    12     6     0     0
     8    10    11     0
    13     3     2    16
```



```
>> x = 1:3
x = 1     2     3

>> y = -2:0
y = -2    -1     0

>> [X, Y] = meshgrid(x, y)
X =
     1     2     3
     1     2     3
     1     2     3
Y =
    -2    -2    -2
    -1    -1    -1
     0     0     0
```

Kan beräknas på följande vis:

```
>> x = x(:)'; X = x(ones(length(y), 1), :)
X =
     1     2     3
     1     2     3
     1     2     3

>> y = y(:); Y = y(:, ones(1, length(x)));
```

```
>> M = magic(4)
M =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

>> M > 11
ans =
     1     0     0     1
     0     0     0     0
     0     0     0     1
     0     1     1     0
```



```
>> M(M > 11)
ans =
    16
    14
    15
    13
    12

>> i = find(M > 11)
i =
     1   % index i kolonnvis ordning
     8   % M(index) och
    12   % M(index_vektor) fungerar
    13
    15

>> m = M(:);

>> m(i)
ans =
    16
    14
    15
    13
    12

>> [j, k] = find(M > 11);
>> j'
ans = 1     4     4     1     3

>> k'
ans = 1     2     3     4     4
```

```
>> A = magic(2)
A =
     1     3
     4     2

>> b = [1; 3]
b =
     1
     3
>> C = [A, b; b', 7]
C =
     1     3     1
     4     2     3
     1     3     7

>> b = (1:3)';

>> F = [b b(3:-1:1) [b([3 1]); 10]]
F =
     1     3     3
     2     2     1
     3     1    10
```

Här en tabell över $\sqrt{x}$:

```
>> x = (1:10)';
>> [x, sqrt(x)]
ans =
    1.0000    1.0000
    2.0000    1.4142
    3.0000    1.7321
    4.0000    2.0000
    5.0000    2.2361
    6.0000    2.4495
    7.0000    2.6458
    8.0000    2.8284
    9.0000    3.0000
   10.0000    3.1623
```


Här ett ineffektivt sätt att bygga upp en matris:

```
>> M = [];
>> for k = 1:5
     M = [M, rand(5, 1)]; % bara för att göra något
   end
```

Ett alternativ är

```
>> M = [];
>> for k = 1:5
     M(:, k) = rand(5, 1);
   end
```

Så här kan man iterera över kolonnerna i en matris eller en vektor

```
>> for col = M
     work ...
   end
```

Är dock ineffektivt

```
a = randn(1, 1000000);
tic
s1 = 0;
for k = 1:1000000
  s1 = s1 + a(k);
end
toc

Elapsed time is 0.011008 seconds.

tic
s2 = 0;
for k = a
  s2 = s2 + k;
end
toc

Elapsed time is 2.592978 seconds.
```


12.3 Strängkonkatenering

Konkatenera, att sätta samman (eng. concatenate). Jämför kedjekurva, catenaria från latinets catena, kedja (blir en cosh-kurva).

```
>> fn = 'Thomas'
fn = Thomas

>> en = 'Ericsson'
en = Ericsson

>> namn = [fn, ' ', en]
namn = Thomas Ericsson

>> val = -12.3456
val = -12.3456

>> title_in_plot = ...
  ['Curve with val = ', num2str(val), '.']
title_in_plot =
Curve with val = -12.3456.

% Säg att man har plottat och vill ha en rubrik
>> title(['Curve with val = ', num2str(val), '.'])

>> disp(['The value is ', num2str(val)])
The value is -12.3456

% Man kan styra formatet i detalj, mera om detta senare.
>> disp(['The value is ', num2str(val, '%6.1f')])
The value is -12.3

>> disp(['The value is ', num2str(val, '%9.2e')])
The value is -1.23e+01

>> num2str(1.23456e20, '%f')
ans = 123456000000000000000.000000
```


Slutligen lite lek med den välkända Sator-kvadraten (se Wikipedia för detaljer):

```
>> S = ['sator'; 'arepo'; 'tenet']
S =
sator
arepo
tenet

>> S = [S; S([2 1], 5:-1:1)]
S =
sator
arepo
tenet
opera
rotas

>> W = char(' ' * ones(5, 9));
>> for k = 1:5, W(k, 1:2:end) = S(k, :); end
>> W
W =
s a t o r
a r e p o
t e n e t
o p e r a
r o t a s
```

Vi har plats över. I vår soteringsalgoritm bytte vi plats på två element via en temporär variabel. Så här kan man gör på en rad:

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

>> v([1 5]) = v([5 1])
v = 5     2     3     4     1     6
```

Här följer några lite svårare exempel som använder metoderna ovan. Vi försöker *vektorisera* koden, dvs. arbeta på vektor/matrisnivå och inte med loopar på elementnivå.

Exempel. Beräkna egenvärden och egenvektorer till **A** nedan och sortera egenvärdena i växande ordning.

```
>> A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

>> [X, L] = eig(A)
X =
   -0.4472    0.0976   -0.6330    0.6780   -0.2619
   -0.4472    0.3525    0.5895    0.3223   -0.1732
   -0.4472    0.5501   -0.3915   -0.5501    0.3915
   -0.4472   -0.3223    0.1732   -0.3525   -0.5895
   -0.4472   -0.6780    0.2619   -0.0976    0.6330
L =
   65.0000         0         0         0         0
         0  -21.2768         0         0         0
         0         0  -13.1263         0         0
         0         0         0   21.2768         0
         0         0         0         0   13.1263
>> [l, ind] = sort(diag(L));
>> [l, ind]
ans =
  -21.2768    2.0000
  -13.1263    3.0000
   13.1263    5.0000
   21.2768    4.0000
   65.0000    1.0000
```



```
>> X = X(:, ind) % sortera även egenvektorerna
X =
    0.0976   -0.6330   -0.2619    0.6780   -0.4472
    0.3525    0.5895   -0.1732    0.3223   -0.4472
    0.5501   -0.3915    0.3915   -0.5501   -0.4472
   -0.3223    0.1732   -0.5895   -0.3525   -0.4472
   -0.6780    0.2619    0.6330   -0.0976   -0.4472
```

■

Exempel. Vi placerar en stor mängd partiklar i kvadraten $(-1,1)\times(-1,1)$. Partiklarna kan röra sig i de fyra väderstrecken (men inte diagonalt t.ex). De fyra riktningarna är lika sannolika, och en partikel flyttar sig alltid samma sträcka, **d**. Om en partikel ligger på randen eller i det yttre av området så absorberas den. Animera händelseförloppet. Rita också ut masscentrums rörelse (partiklarna har alla samma massa) och plotta hur antalet kvarvarande partiklar varierar med tiden (antalet tidssteg).

```
num_particles = 1000;          % number of particles
x = 0.5 - rand(num_particles, 1); % original positions
y = 0.5 - rand(num_particles, 1);

figure(1)
hold off
plot(x, y, 'ko')
axis equal
axis([-1 1 -1 1])

d          = 0.02;
time_steps = 10000;
remain     = zeros(time_steps, 1);

for time_step = 1:time_steps
  len_x = size(x, 1);
  remain(time_step) = len_x;

  r = rand(len_x, 1);
```



```
  ix = r < 0.25;
  x(ix) = x(ix) - d;   % move left

  ix = 0.25 <= r & r < 0.5;
  x(ix) = x(ix) + d;   % move right

  ix = 0.5 <= r & r < 0.75;
  y(ix) = y(ix) - d;   % move down

  ix = 0.75 <= r;
  y(ix) = y(ix) + d;   % move up

  % how many inside the square?
  inside = find(abs(x) < 1 & abs(y) < 1);
  if isempty(inside)  % any remaining?
    break
  end
  x = x(inside);
  y = y(inside);

  plot(x, y, 'ko', mean(x), mean(y), 'r*')
  axis equal
  axis([-1 1 -1 1])
  drawnow
end

% Number of remaining particles as a function of time.
% 1:time_step, since we may have a premature exit
% (in which case the zero number is not plotted).
figure(2)
plot(remain(1:time_step))
```

■

Exempel. Här ett exempel inspirerat av "the Galton board" (Sir Francis Galton, 1822-1911). Se Wikipedia för detaljer och en bild. Kulor får falla genom en spikbräda, kulorna faller till vänster eller höger vid varje spik. Kulorna släpps ned vid samma ställe på brädan. Kulhögen vid brädans nederdel får formen av en normalfördelningskurva.

```
% The Galton board
rows    = 500;       % # of rows in the board
n_balls = 50000;
d       = 1 / rows;

x = zeros(n_balls, 1); % x-coordinates of the balls
for k = 1:rows
  r = 0.5 - rand(n_balls, 1);
  ix = r < 0;
  x(ix) = x(ix) - d;
  ix = ~ix;
  x(ix) = x(ix) + d;
end

figure(1)
hist(x, rows); % coordinates, # of bins
```

■

Exempel. Här är ett exempel där vi skapar en speciell matris av matriser, en sk Kroneckerprodukt. Säg att A och B är två matriser, då definieras Kroneckerprodukten som

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}$$

Man kan t.ex. använda produkten för att lösa Sylvesters ekvation

$$AX + XB = C$$

där vi antar att A, B, C är kända kvadratiska matriser, av dimension n, och att vi söker matrisen X.

Vidare antar vi att $\boldsymbol{A}$ och $\boldsymbol{B}$ har sådana egenvärden att problemet har entydig lösning. Låt $\boldsymbol{vec}(\boldsymbol{X})$ vara kolonnvektorn vi får genom att stapla kolonnerna i $\boldsymbol{X}$ ovanpå varandra. Man kan bevisa att $\boldsymbol{AX} + \boldsymbol{XB} = \boldsymbol{C}$ har en lösning som ges av:

$$(\boldsymbol{I} \otimes \boldsymbol{A} + \boldsymbol{B}^T \otimes \boldsymbol{I})\ \boldsymbol{vec}(\boldsymbol{X}) = \boldsymbol{vec}(\boldsymbol{C})$$

där $\boldsymbol{I}$ betecknar enhetsmatrisen av dimension $\boldsymbol{n}$. Detta kan bli ett stort system, koefficientmatrisen har ju dimension $\boldsymbol{n}^2$.
Här kommer lite kod, först en allmän rutin för Kroneckerprodukter:

```
function K = kron_prod(A, B)

[rows_A, cols_A] = size(A);
[rows_B, cols_B] = size(B);
% preallocate
K = zeros(rows_A * rows_B, cols_A * cols_B);

rows_B_m1 = rows_B - 1;  % for efficiency
cols_B_m1 = cols_B - 1;

block_row = 1;
for row_A = 1:rows_A
  block_col = 1;
  for col_A = 1:cols_A
    K(block_row:block_row + rows_B_m1,   ...
      block_col:block_col + cols_B_m1) = ...
        A(row_A, col_A) * B;
    block_col = block_col + cols_B;
  end
  block_row = block_row + rows_B;
end
```

och här Sylvesterkoden (för det kvadratiska fallet enbart)

```
function X = sylvester(A, B, C)
n = length(A);
I = eye(n);
X = (kron_prod(I, A) + kron_prod(B', I)) \ C(:);
X = reshape(X, n, n);
```


och slutligen en testkörning:

```
>> A = rand(2); B = rand(2); C = rand(2);
>> X = sylvester(A, B, C);
>> A * X + X * B - C
ans =
  -1.1102e-16   1.8041e-16
  -6.6613e-16  -8.3267e-16
```

MATLAB har färdiga rutiner för båda dessa uppgifter, **kron** och **lyap**, för Lyapunov, som inte gör som ovan (MATLABs är mycket snabbare). Vår **kron_prod** är dock snabbare än MATLABs variant, t.ex.

```
>> A = rand(30); B = rand(30);

>> tic; for k = 1:10, K1 = kron_prod(A, B); end; toc
Elapsed time is 0.227508 seconds.

>> tic; for k = 1:10, K2 = kron(A, B); end; toc
Elapsed time is 0.475439 seconds.

>> norm(K1 - K2, 1)  % kolla skillnaden
ans = 0

>> A = rand(50, 23); B = rand(34, 45);

>> tic; for k = 1:10, K1 = kron_prod(A, B); end; toc
Elapsed time is 0.468834 seconds.

>> tic; for k = 1:10, K2 = kron(A, B); end; toc
Elapsed time is 0.725575 seconds.

>> norm(K1 - K2, 1)
ans = 0
```

Vi hade också kunnat göra en specialvariant av **kron_prod** när vi vet att en matris är enhetsmatrisen.

Exempel. Antag att vi vill lösa ett linjärt ekvationssystem, $\boldsymbol{Ax} = \boldsymbol{b}$, där $\boldsymbol{A}$ och $\boldsymbol{b}$ innehåller komplexa tal. Visa hur vi kan lösa ett dubbelt så stort reellt system och implementera metoden i MATLAB (lite onödigt, eftersom MATLAB kan hantera komplexa matriser och den reella varianten tar mer tid). Låt $\boldsymbol{A}_R = \mathrm{real}(\boldsymbol{A})$ och $\boldsymbol{A}_I = \mathrm{imag}(\boldsymbol{A})$ och analogt för $\boldsymbol{b}$ och $\boldsymbol{x}$. Vi får

$$\boldsymbol{Ax} = \boldsymbol{b} \Leftrightarrow (\boldsymbol{A}_R + i\boldsymbol{A}_I)(\boldsymbol{x}_R + i\boldsymbol{x}_I) = \boldsymbol{b}_R + i\boldsymbol{b}_I \Leftrightarrow$$

$$\begin{cases} \boldsymbol{A}_R\boldsymbol{x}_R - \boldsymbol{A}_I\boldsymbol{x}_I = \boldsymbol{b}_R \\ \boldsymbol{A}_R\boldsymbol{x}_I + \boldsymbol{A}_I\boldsymbol{x}_R = \boldsymbol{b}_I \end{cases} \Leftrightarrow \begin{bmatrix} \boldsymbol{A}_R & -\boldsymbol{A}_I \\ \boldsymbol{A}_I & \boldsymbol{A}_R \end{bmatrix} \begin{bmatrix} \boldsymbol{x}_R \\ \boldsymbol{x}_I \end{bmatrix} = \begin{bmatrix} \boldsymbol{b}_R \\ \boldsymbol{b}_I \end{bmatrix}$$

Här lite MATLAB-kod:

```
% some input
n = 500;
A = randn(n)    + 1i * randn(n);
b = randn(n, 1) + 1i * randn(n, 1);

% extract A_R, A_I
A_R = real(A);
A_I = imag(A);

R = [A_R, -A_I; A_I, A_R];
r = [real(b); imag(b)]; % form real rhs

x_r = R \ r;  % solve real system

% convert back to complex
x_c = x_r(1:n) + 1i * x_r(n+1:end);
```

■

13 Cellfält, poster och mängder

En nackdel med vanliga vektorer och matriser *kan* vara att elementen måste vara av samma typ. Detta problem löses av cellfältet, där ett element kan vara en skalär, ett annat en matris, ett tredje en sträng etc.

Ytterligare en nackdel med vanliga vektorer *kan* vara att man refererar till elementen via index. Man kan inte skapa en person-vektor och indexera med adress, ålder etc. Det går med en post-variabel. Andra namn på post är record, struct (på svengelska).

Först några enkla exempel:

```
>> c{1} = 3   % { } i stället för ( )
c = [3]

>> c{2} = magic(3)
c = [3]    [3x3 double]

>> c{3} = 'a string'
c = [3]    [3x3 double]    'a string'

>> c{2}
ans =
     8     1     6
     3     5     7
     4     9     2

>> c{2}(2, 3)
ans = 7

>> c{3}
ans = a string

>> c{3}(4)
ans = t
```

```
>> C{1, 1} = 'a';   C{1, 2} = 'ab';  % går inte med [ ]
>> C{2, 1} = 'abc'; C{2, 2} = 'abcd';
>> C
C =  'a'      'ab'
     'abc'    'abcd'

>> C{1, 3} = c
C =  'a'      'ab'      {1x3 cell}
     'abc'    'abcd'            []

>> C{1, 3}{2}(2, 3)
ans = 7
```

Här är en annan syntax

```
>> c = {'a' 'ab' 'abc' 'abcd'}
c = 'a'    'ab'    'abc'    'abcd'

>> F = {'a' magic(3) rand(1, 4)}
F =
    'a'    [3x3 double]    [1x4 double]
```

En överraskning är kanske följande:

```
>> c = {'a', 'ab', 'abc', 'abcd'};
>> ind = [1 3]
>> res = c{ind}
res = a

>> [res1, res2] = c{ind}
res1 = a
res2 = abc

>> c{ind}        % returnerar innehållet
ans = a
ans = abc
>> res = c(ind)  % returnerar cellerna
res = 'a'    'abc'
>> class(res)    % ger datatypen (ungefär)
ans = cell
```


Data i en post nås via ett namn och inte via ett index.

```
>> person.namn = 'Anders Andersson'
person =
    namn: 'Anders Andersson'

>> person.adress = 'Gibraltargatan 24'
person =
      namn: 'Anders Andersson'
    adress: 'Gibraltargatan 24'

>> person.tel = '1234567'
person =
      namn: 'Anders Andersson'
    adress: 'Gibraltargatan 24'
       tel: '1234567'

>> person.tel
ans =
1234567
```

Vi kan även ha poster av poster och vektorer (cellfält) av poster osv.

```
>> personer(1).namn   = 'Anders';
>> personer(1).adress = 'Gata';
>> personer(1).tel    = '1234567';

>> personer(2).namn   = 'Karin';
>> personer(2).adress = 'Torg';
>> personer(2).tel    = '7654321';

1x2 struct array with fields:
    namn
    adress
    tel

>> personer(2).tel
ans = 7654321
```



```
>> personer(2).fel = '22'  % fel skapas i  alla
personer =                 % element
1x2 struct array with fields:
    namn
    adress
    tel
    fel
```

Använder vi ett cellfält, så får bara andra elementet, i **personer**, medlemmen **fel**.

13.1 Mängder

MATLAB har stöd för räkning med mängder av tal. Mängderna representeras som vektorer. Här några exempel:

```
>> set1 = unique([4 1 2 1 6 -5 3 10])
set1 = -5     1     2     3     4     6    10

>> set2 = unique([-5, 1:6])
set2 = -5     1     2     3     4     5     6

>> union(set1, set2)
ans = -5     1     2     3     4     5     6    10

>> intersect(set1, set2)
ans = -5     1     2     3     4     6

>> setdiff(set1, set2)
ans = 10

>> setdiff(set2, set1)
ans = 5
```



```
>> ismember([2 5 1 7 8 12 -33], set1)
ans = 1     0     1     0     0     0     0
```

Man behöver inte använda **unique** först:

```
>> union([1 2 1], [1 1 2 2 3 3])
ans = 1     2     3
```

Man kan också ha cellvektorer av strängar:

```
>> s1 = {'a', 'aa', 'bb', 'b', 'a', 'c', 'ccc', 'a'}
s1 = 'a'  'aa'  'bb'  'b'  'a'  'c'  'ccc'  'a'

>> unique(s1)
ans = 'a'  'aa'  'b'  'bb'  'c'  'ccc'

>> s2 = {'aa', 'aa', 'bbb', 'b', 'a', 'c', 'ccc', 'c'}
s2 = 'aa'  'aa'  'bbb'  'b'  'a'  'c'  'ccc'  'c'

>> union(s1, s2)
ans = 'a'  'aa'  'b'  'bb'  'bbb'  'c'  'ccc'

>> setdiff(s1, s2)
ans = 'bb'

>> setdiff(s2, s1)
ans = 'bbb'
```

14 Enkel grafik

MATLAB har ett stort antal funktioner för att rita kurvor, ytor och andra grafiska objekt. Grafikmanualen är på flera hundra sidor. I den här kursen kommer vi att titta på en *liten* del av vad MATLAB kan.

Att plotta y = f(x)

```
x = linspace(0, 10);     % 100 x-värden från 0 till 10.
% linspace(0, 10, 50) ger 50 värden
plot(x, exp(-0.1 * x) .* cos(x))  % heldragen kurva
grid on                  % stödlinjer
xlabel('x')              % lite text utmed x-axeln
ylabel('y')              % y-axeln
title('En enkel kurva')  % och en rubrik
```

Att plotta punkter:

```
plot(x, y, 'o')   % med ringar
plot(x, y, 'r*')  % med röda *
```

Mer än en kurva
Det finns väsentligen två alternativ, det första är att plotta flera kurvor med ett plot-kommando:

```
% två kurvor
plot(x, exp(-0.1 * x) .* cos(x), x, sin(x))

% två kurvor, en röd heldragen och en blå streckad
plot(x, exp(-0.1 * x) .* cos(x), 'r-', x, sin(x), 'b--')
```

Man kan också "hålla kvar" en plot med kommandot **hold on**. Om man kör sitt program flera gånger får man inte glömma att slå av hold-funktionen, med **hold off**, annars kommer man att accumulera fler och fler plottar i samma fönster. Jag utgår i följande exempel i från av vi inte har gett **hold on**-kommandot.


```
% plotta en kurva (plot rensar först fönstret)
plot(x, exp(-0.1 * x) .* cos(x))
hold on                          % håll kvar
plot(x, sin(x))                  % bygg på

hold off                         % slå av hold
% rensa och plotta sen kurvan
plot(x, exp(-0.1 * x) .* cos(x), 'r-')
hold on                          % håll kvar
plot(x, sin(x), 'b--')           % bygg på
```

Mer än ett fönster (eller flera axlar i ett fönster)
figure skapar ett nytt fönster. Det senaste öppnade blir aktivt (där hamnar nästa plot). Man kan göra ett fönster aktivt genom att klicka på det.
Alternativt kan man skriva t.ex. **figure(2)**, där 2 är fönstrets nummer. Detta ser också till att fönstret lägger sig överst, så att man inte behöver leta efter det. Finns det inget fönster med nummer två, så skapas det. Jag använder ofta följande kommandosekvens.

```
figure(1)  % aktivera fönster ett och lägg det överst
hold off   % slå av hold
plot(...   % ge lämpligt plotkommando
hold on    % håll kvar plotten
plot(...   % nytt plotkommando
```

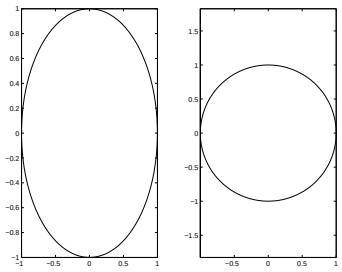
Man kan också rita flera axlar (plottar) i ett fönster (man behöver inte ha alla kommandona på en rad). **help subplot** för detaljer.

```
figure(2)
x = linspace(1, 10);
subplot(221); plot(x, log(x));     axis equal; grid on
subplot(222); semilogx(x, log(x)); grid on
subplot(223); semilogy(x, exp(x)); grid on
subplot(224); loglog(2.^x, exp(x));grid on
```


Att få korrekt axelskalning
Följande kodsegment är tänkt att rita en cirkel:

```
f = linspace(0, 2 * pi);
plot(cos(f), sin(f))
```

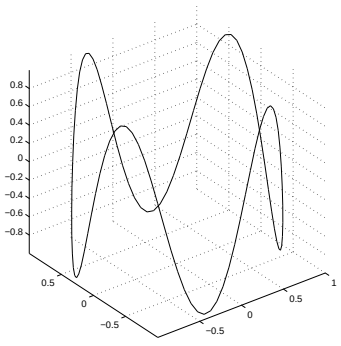
Den vänstra bilden visar resultatet. Cirkeln ser ut som en ellips eftersom MATLAB försöker att utnyttja fönstret maximalt och fylla ut det. Beroende på fönstrets storlek kan ellipsens storaxel vara parallell med x-axeln i stället.



Om jag vill få en korrekt bild måste jag skala om axlarna. Om jag ger kommandot **axis equal** efter att ha plottat får jag bilden till höger. Observera att **axis square** inte gör samma sak (kommandot ger axlar som upptar lika många centimeter i x- och y-led).

Att plotta kurvor i rummet och i planet
Om vektorerna x och y innehåller punkter på kurvan kommer **plot(x, y)** att förbinda punkterna (i ordning) med räta linjesegment och vi får en kurva i planet (som i exemplet ovan). Har vi också en z-vektor ger **plot3(x, y, z)** en kurva i rummet. Här en rymdkurva som svänger likt en sinus-kurva i z-led och som går som en cirkel i x-y-planet.


```
f = linspace(0, 2 * pi);
plot3(cos(f), sin(f), sin(4 * f))  % fyra perioder
axis equal
grid on
```

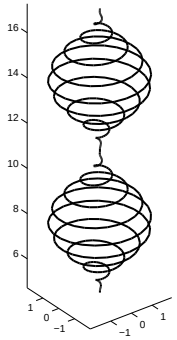


Ökar vi antalet svängningar i z-led räcker kanske inte de 100 punkterna som linspace ger oss utan kurvan blir hackig, eftersom man ser de enskilda linjesegmenten.

```
% blir hackigt vid min och max
plot3(cos(f), sin(f), sin(10 * f))
f = linspace(0, 2 * pi, 200);       % be om 200 punkter
plot3(cos(f), sin(f), sin(10 * f))  % ger inga hack
```

Här en spiral med bredare linje.

```
n   = 500;
phi = linspace(15 * pi, 55 * pi, n);
r   = 1 + sin(0.1 * phi);
figure(1)
plot3(r .* cos(phi), r .* sin(phi), 0.1 * phi, ...
      'Linewidth', 2)
axis equal
```



Att plotta polygoner i planet

Vi kan använda det vanliga plot-kommandot för att rita randen på en polygon, så detta avsnitt visar hur man fyller en polygon med färg. När vi fyller polygoner behöver vi inte sluta randkurvan. Det räcker med att definiera hörnen (i ordning, med- eller moturs).

```
% en fylld, röd, enhetskvadrat
fill([0 1 1 0], [0 0 1 1], 'r')
```

Numreringen är viktig, följande plot ger en fluga (till en kostym):

```
fill([0 1 1 0]', [0 1 0 1]', 'b')
```

Här ritar jag två polygoner med ett kommando (enhetskvadraten samt denna translaterad med vektorn (1, 1)). Kommandot är ett specialfall av **fill(X, Y, 'r')**. Först kolonnen i **X** samt första kolonnen i **Y** definierar hörnen i första polygonen. Andra kolonnen i **X** resp. **Y** definierar andra polygonen etc.

```
x = [0 1 1 0]';  % notera ', transponat
y = [0 0 1 1]';
fill([x x+1], [y y+1], 'r')
```


Om man inte vill ha några kantlinjer skriver man:

```
fill([x x+1],[y y+1], 'r', 'Edgecolor', 'None')
```

Vill vi ha gröna, breda kantlinjer skriver vi:

```
fill([x x+1], [y y+1], 'r', 'Edgecolor', 'g', ...
     'Linewidth', 5)
```

Att plotta en funktionsyta, z = f(x, y)

Antag att vi vill rita punktmängden (ytan)

$$M = \{(x, y, z) : z = f(x, y), (x, y) \in D\},$$

$$D = \{(x, y) : xmin \leq x \leq xmax,\ ymin \leq y \leq ymax\}$$

så D är ett rektangulärt område i x-y-planet.
Ett sätt att göra detta i MATLAB är att först bilda ett rutnät (grid) i mängden D. För att förstå hur detta fungerar, exekverar vi följande MATLAB-rad och inspekterar matriserna:

```
[X, Y] = meshgrid(-1:0.5:1, 1:4)
X =
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
   -1.0000   -0.5000         0    0.5000    1.0000
Y =
     1     1     1     1     1
     2     2     2     2     2
     3     3     3     3     3
     4     4     4     4     4
```

Antag nu att matrisen **Z**, innehåller funktionsvärdena, **f(X(j, k), Y(j, k))**. Kommandot

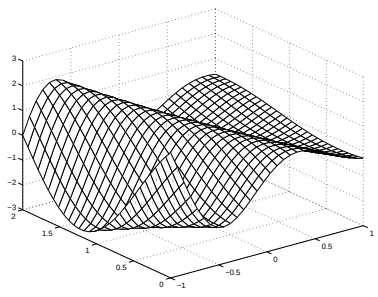
```
mesh(X, Y, Z)
```

ritar då en approximation, i form av ett nät, av ytan. Ju bättre upplösning man har i rutnätet desto jämnare yta får man.

Samtidigt tar matriserna mer plats och grafiken blir slöare (om man vill snurra på ytan till exempel). Tänk också på att om Du fördubblar antalet punkter i x- och y-led så innehåller matriserna fyra gånger så många element.
Matrisen **Z** kan byggas upp med loopar, men ett snabbare och enklare sätt är att använda elementvisa operationer. Testa t.ex. följande rader:

```
x = linspace(-1, 1, 30);
y = linspace(0, 2, 30);
[X, Y] = meshgrid(x, y);
Z = exp(-X) .* sin(4*X + 2*Y); % OBS .*
figure                  % Skapa nytt fönster
mesh(X, Y, Z)
rotate3d          % Slå på rotation
figure            % (kan klicka på rotationspilen också)
surf(X, Y, Z)     % Ett annat sätt att rita ytan
shading interp    % Jämna ut färgerna (löst uttryckt)
rotate3d
```

Den första bilden blir så här (jag har inte tagit med den andra, eftersom den blir så stor).



Regnbågsfärgen i ovanstående andra bilden kan ändras (man kan ha en färg och lägga på belysning t.ex.) men vi tar inte upp detta i denna kurs.

Exempel. En viktig kvantitet i flervariabelkursen är gradienten, $\nabla f = \text{grad} f = [f'_x, f'_y]$ om f är en rellvärd funktion av två reella variabler. Låt oss åskådliggöra gradienterna i en del av planet då:

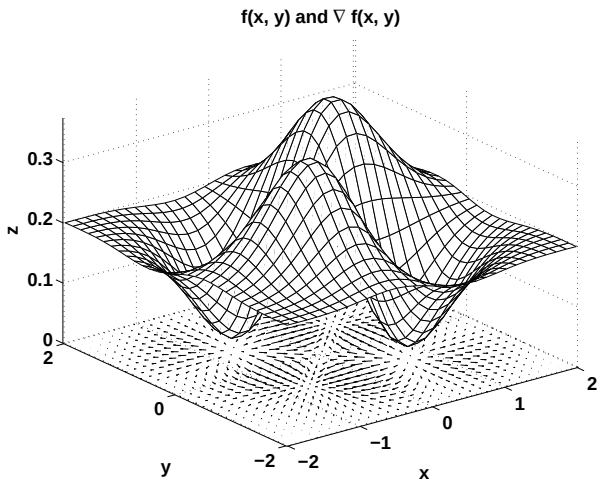
$$f(x, y) = 0.2 + e^{-x^2-y^2} \sin(xy),\ -2 \leq x,\ y \leq 2$$

$$\nabla f = e^{-x^2-y^2}[y\cos(xy) - 2x\sin(xy),\ x\cos(xy) - 2y\sin(xy)]$$

Lokala extrempunkter har vi när $x = \pm y$ och $\arctan x^2 = 0.5$. Origo är en sadelpunkt.

```
[X, Y] = meshgrid(linspace(-2, 2, 30));
E  = exp(-X.^2 - Y.^2);
C  = cos(X .* Y);
S  = sin(X .* Y);
Fx = E .* (Y .* C - 2 * X .* S);
Fy = E .* (X .* C - 2 * Y .* S);
F  = 0.2 + E .* S;

figure(1)
hold off
mesh(X, Y, F)
hold on

quiver(X, Y, Fx, Fy)  % för gradienterna

title('f(x, y) and \nabla f(x, y)', ...  % obs \nabla
      'Fontsize', 16, 'Fontweight', 'Bold')
xlabel('x', 'Fontsize', 16, 'Fontweight', 'Bold')
ylabel('y', 'Fontsize', 16, 'Fontweight', 'Bold')
zlabel('z', 'Fontsize', 16, 'Fontweight', 'Bold')
set(gca, 'Fontsize', 16, 'Fontweight', 'Bold')
axis tight

% bra om man skall inkludera en bild i en rapport
print -deps gradient1.eps
```

Man skall sitta vid datorn för att detta skall bli riktigt bra.

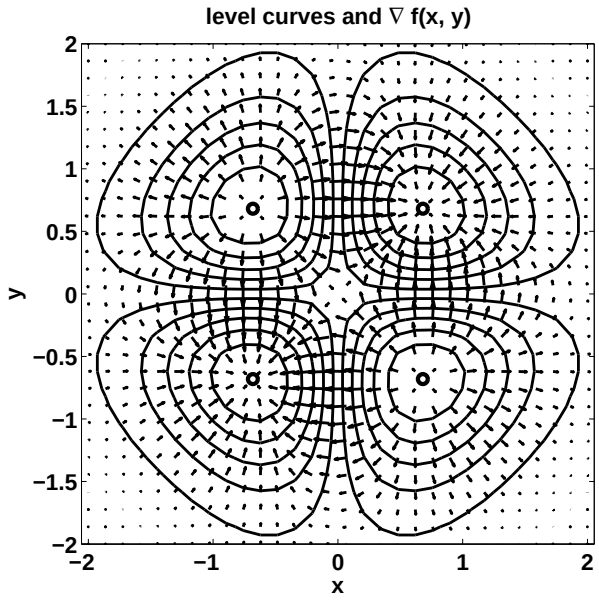


Låt oss rita ut en nivåkurva och övertyga oss om att gradienterna är ortogonala mot tangenterna till nivåkurvan. Jag har ökat på linjebredder för att de skall synas på OH-sidan (och för att visa att det går).

```
x_opt = sqrt(atan(0.5));   y_opt = x_opt;
figure(2)
hold off

% för nivåkurvor
contour(X, Y, F, 10, 'Linewidth', 2, 'EdgeColor', 'k')
hold on

quiver(X, Y, Fx, Fy, 'Linewidth', 2, 'Color', 'k')
```



```
% plotta stationära punkter
plot(x_opt * [1 1 -1 -1], y_opt * [1 -1 1 -1], 'ko', ...
     'Markersize', 7, 'Linewidth', 3)

title('level curves and \nabla f(x, y)', ...
      'Fontsize', 16, 'Fontweight', 'Bold')
xlabel('x', 'Fontsize', 16, 'Fontweight', 'Bold')
ylabel('y', 'Fontsize', 16, 'Fontweight', 'Bold')
set(gca, 'Fontsize', 16, 'Fontweight', 'Bold')
axis equal
print -deps gradient2.eps
```



■

15 Prestanda

De viktigaste åtgärderna för att få en snabb MATLAB-kod är:

- använd en snabb algoritm
- försök att utnyttja MATLABs kompilerade rutiner
- vektorisera koden, försök att arbeta på matris/vektor-nivå
- preallokera minne
- tänk på evalueringsordningen

Här följer några exempel:

```
n = 3000;
A = rand(n);
B = rand(n);

tic
 for j = 1:n
   for k = 1:n
     A(j, k) = A(j, k) + B(j, k);
   end
 end
toc

tar 0.11s

tic
  A = A + B;
toc
tar 0.02s
```

Antag att vi gör ett numeriskt experiment och måste samla på oss en stor mängd vektorer för senare analys (samma vektor i exemplet nedan).


```
A = [];
x = rand(n, 1);
tic
 for k = 1:n
   A(:, k) = x;
 end
toc
tar 96s

tic
A = zeros(n); % preallocate
 for k = 1:n
   A(:, k) = x;
 end
toc
tar 0.06s

A = [];
tic
 for k = 1:n
   A(:, n - k + 1) = x;
 end
toc
tar 0.06s
```

I följande exempel är `W` en 10000×15-matris och `x` är en kolonnvektor med 10000 element.

```
W = rand(10000, 15);
x = rand(10000, 1);
xs = x;
tic
  for k = 1:10
    x = W * W' * x;
  end
toc
tog 21s
```

```
x = xs;
tic
  for k = 1:10
    x = W * (W' * x);
  end
toc

tog 0.006s
```

MATLAB har en utmärkt profilerare (eng. profiler) med vars hjälp man kan analysera sina program vad avser tidsåtgång. Profileraren har också en rad, M-Lint, som ger kodningstips.
I kodexemplet, `A(:, k) = x;`, får vi rådet:

```
'A' might be growing inside a loop.
Consider preallocating for speed.'
```

Man får också detta meddelande i MATLABs editor (placera musen på eventuella korta horisontella streck i högra marginalen).

16 Rekursion

Rekursion är en kraftfull och elegant problemlösningsmetod (för vissa problem). Metoden kan dock vara resurskrävande (tid och minne). Den används inte speciellt ofta i numeriska beräkningar, men är desto vanligare inom datalogi.
Grundidéen är att formulera ett problem i termer av ett likartat, men mindre, enklare, problem.

Vi kan lösa problemet om vi kan lösa ett enklare problem.
Vi kan lösa detta enklare problem om vi kan lösa ett ännu enklare problem.
Man fortsätter på detta vis och får till slut ett problem som är trivialt att lösa.
Exempel. Låt oss börja med standardexemplet, beräkning av $n!$ (där man i verkligheten aldrig skulle använda rekursion).

$$n! = \begin{cases} n \cdot (n-1)!, & n > 0 \\ 1, & n = 0 \end{cases}$$

Så

```
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1!
1! = 1 * 0!
0! = 1
```

När man har beräknat det sista värdet, 0!, går man nedifrån och upp och får slutligen 4!. Så här ser motsvarande MATLAB-program ut. Vi förutsätter att funktionen anropas med ickenegativa tal.

```
function nf = fakultet(n)
if n > 0
  nf = n * fakultet(n - 1);
else
  nf = 1;
end
```



```
>> fakultet(4)
ans = 24
```

■

Några exempel till:

Rekursiv summation. I praktiken använder man givetvis en for-loop eller `sum`.

```
>> type recu_sum

function s = recu_sum(vec)
if length(vec) == 1
  s = vec;
else
  s = vec(1) + recu_sum(vec(2:end));
end

>> v = rand(10, 1);
>> recu_sum(v) - sum(v)
ans = 0
```

Palindromexemplet ytterligare en gång:

```
>> type ispalin

function palin = ispalin(string)
if length(string) <= 1
  palin = true;
else
  palin = string(1) == string(end) & ...
          ispalin(string(2:end-1));
end

>> ispalin(lower('CigarTossitinacanitissotragic'))
ans = 1

>> ispalin('a')
ans = 1
```

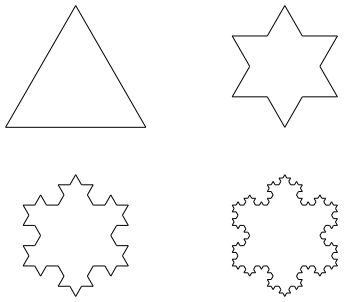


```
>> ispalin('aa')
ans = 1

>> ispalin('')    % tomma strängen, empty string
ans = 1
```

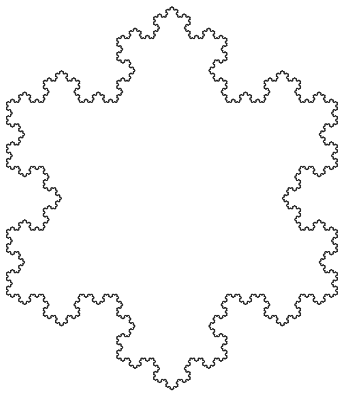
Man kan ha två funktioner som anropar varandra (eng. mutual recursion). Här ett fånigt exempel, som visar ett komplicerat sätt att summera elementen i vektorn `v`.

```
function mutual(v)

func1(v)

function s = func1(v)
if length(v) == 1
  s = v;
else
  s = v(end) + func2(v(1:end-1)); % anropar func2
end

function s = func2(v)
if length(v) == 1
  s = v;
else
  s = v(1) + func1(v(2:end));     % anropar func1
end
```

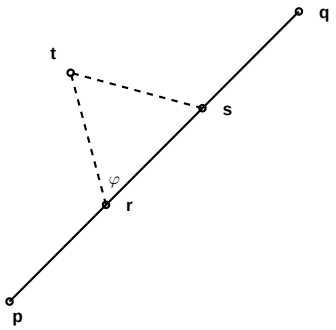
Exempel. Koch-kurvan, efter svenske matematikern Helge von Koch, 1870-1924, "Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire", 1904. Kontinuerlig men ej deriverbar någonstans ("sans tangente").



Om ursprungstriangeln är en liksidig triangel med sidlängd ett, så är omkretsen 3, och nästa figur har omkretsen $4/3 \cdot 3$ och ändå nästa har omkretsen $(4/3)^2 \cdot 3$ osv. Om vi upprepar förfiningen ett oändligt antal gånger så går längden mot oändligheten, men den inskrivna arean är begränsad. Här en bild mer några flera steg:



Vi vill nu använda rekursion för att rita bilden. Den grundläggande operationen är att dela upp ett linjesegment i fyra delar, enligt följande bild:



Givet punkterna p och q får vi r och s genom:

$$r = p + \frac{1}{3}(q-p) \quad s = q - \frac{1}{3}(q-p)$$

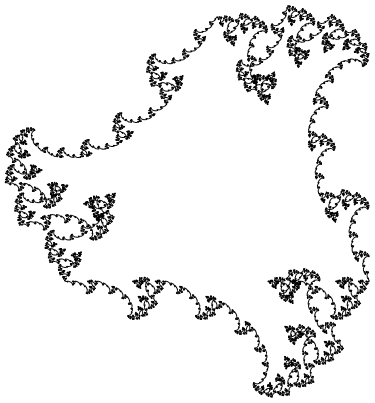
($q - p$ är ju vektorn från p till q). Punkten t kan beräkna på flera olika sätt. Ett är att rotera vektorn $r - p$, en vinkel $\varphi = \pi/3$, och addera den till punkten r. Så,

$$t = r + \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} (r-p) = r + \frac{1}{2}\begin{bmatrix} 1 & -\sqrt{3} \\ \sqrt{3} & 1 \end{bmatrix} (r-p)$$

Låt R vara rotationsmatrisen och sätt $d = (q-p)/3$. Vi får då:

$$r = p + d, \quad s = q - d, \quad t = r + Rd$$

Här är ett program. Vi avslutar rekursionen när sidlängden, d, är mindre än ett givet värde. Notera att vi har kastat om ordningen på ändpunkterna för den horisontella linjen, så att linjen roteras åt rätt håll.
Om man ändrar φ kan man enkelt skapa andra figurer. I följande bild är $\varphi = \pi/1.3$.



Här följer koden:

```
function koch1
% Draw part of Koch curve

global R
phi = pi / 3;
R   = [cos(phi) -sin(phi); sin(phi) cos(phi)];

h = figure;    set(h, 'Color', 'w')
hold on
axis equal
axis off

draw([1; 0], [0; 0])            % reverse order
draw([0; 0], [0.5; sqrt(3)/2])
draw([0.5; sqrt(3)/2], [1; 0])

function draw(p, q)
global R

d = (q - p) / 3;
if norm(d) <= 0.003
  plot([p(1) q(1)], [p(2) q(2)], 'k-')
else
  r = p + d;   s = q - d;   t = r + R * d;
  draw(p, r); draw(r, t); draw(t, s); draw(s, q);
end
```

Om man får (väldigt) många nivåer kan MATLAB ge felmeddelandet:

```
??? Maximum recursion limit of 500 reached.
Use set(0,'RecursionLimit',N) to change the limit.
Be aware that exceeding your available stack
space can crash MATLAB and/or your computer.
```

Exempel. Man kan missbruka rekursion. Här är en deluppgift från första laborationen i numerisk analys för D2 (2000).

> Rekursionsformler är vanliga (bland annat i datalogi).
> Antag att vi vill beräkna $x_k,\ k = 1, \ldots, 60$, då
> $x_{k+1} = 2.25x_k - 0.5x_{k-1}$ med $x_1 = 1/3$, $x_2 = 1/12$.
> Man kan visa att den exakta lösningen ges av $x_k = 4^{1-k}/3$.
> Plotta den beräknade och den exakta lösningen i MATLAB.
> Kommentar? Rita kurvorna i samma diagram.
> Använd logaritmisk skala i y-led.

Så här skall man *inte* göra (rätt många prövade faktiskt)!

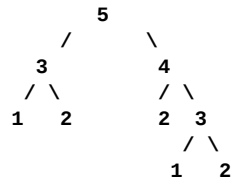
```
function xk = recu(k)

if k == 1
  xk = 1 / 3;
elseif k == 2
  xk = 1 / 12;
else
  xk = 2.25 * recu(k) - 0.5 * recu(k - 1);
end
```

Man skriver givetvis en loop och använder vektorer:

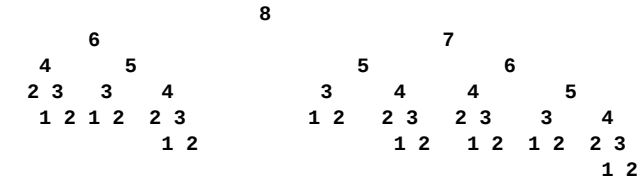
```
x = [1/3; 1/12];
for k = 2:60
  x(k+1) = 2.25 * x(k) - 0.5 * x(k - 1);
end
```

Den senare lösningen tar knappt mätbar tid, 0.001s kanske. Anropet **recu(40)** tar ungefär 44 minuter. Varför? Här kan det vara belysande att studera sekvensen av anrop. Vi använder ett träd (roten överst) för att visa hur anropssekvensen blir.

Låt **k** beteckna anropet **recu(k)**. **recu(5)** ger följande trädstruktur:



recu(8) ger följande trädstruktur (utan grenar):



Trädet är inte djupt (eng. deep tree), men det har stor bredd (eng. wide tree) vilket ger många anrop av **recu**. Låt antalet anrop vara a_k, där $a_1 = a_2 = 1$, då blir $a_k = a_{k-1} + a_{k-2} + 1, k = 3, \ldots$. Man kan uttrycka a_k explicit i k (man får lösa en differensekvation) och får då:

$$a_k = \frac{2}{\sqrt{5}}\left[r^k - \frac{1}{(-r)^k}\right] - 1,\ r = \frac{1+\sqrt{5}}{2} \approx 1.618$$

För stora k är

$$a_k \approx 0.89\ r^k$$

T.ex. gäller att $a_{40} \approx 2 \cdot 10^8$ och $a_{60} \approx 3 \cdot 10^{12}$. ■

Exempel. Skriv ut alla permutationer av tecknen i en sträng. Så givet strängen **'hej'** vill vi få utskriften:

```
hej
hje
ehj
ejh
jhe
jeh
```

Här är en lösning: Idé: säg att vi har strängen $s = [s_1, s_2, s_3, s_4]$, där s_1, s_2, s_3, s_4 är tecken. Vi kan då generera alla permutationer genom

$$[s_1, perm([s_2, s_3, s_4])]$$
$$[s_2, perm([s_1, s_3, s_4])]$$
$$[s_3, perm([s_2, s_1, s_4])]$$
$$[s_4, perm([s_2, s_3, s_1])]$$

```
function perm(s, p)
if p == length(s)
  disp(s)
else
  for j = p:length(s)
    s([p j]) = s([j p]);  % Same idea as above, but
    perm(s, p + 1);       % s is updated in each
  end                     % iteration
end
```

```
>> perm('hej', 1)
hej
hje
ehj
ejh
jhe
jeh
```

■

Exempel. Vi kan använda rutinen ovan för att generera alla magiska kvadrater av ordning tre.

```
function magic3
perm(1:9, 1)

function perm(s, p)

if p == length(s) && is_magic(s)  % note two &
  disp(reshape(s, 3, 3))
  disp('␣␣␣␣␣------------')
else
  for j = p:length(s)
    s([p j]) = s([j p]);
    perm(s, p + 1);
  end
end

function ret = is_magic(s)
%          [ s(1) s(4) s(7) ]
% matrix = [ s(2) s(5) s(8) ]
%          [ s(3) s(6) s(9) ]

ret = ...
 sum(s(1:3))     == 15 & sum(s(4:6))     == 15 & ...
 sum(s(7:9))     == 15 & sum(s([1 4 7])) == 15 & ...
 sum(s([2 5 8])) == 15 & sum(s([3 6 9])) == 15 & ...
 sum(s([1 5 9])) == 15 & sum(s([3 5 7])) == 15;
```

```
>> magic3
     2     9     4
     7     5     3
     6     1     8
     ------------
     2     7     6
     9     5     1
     4     3     8
     ------------
```

```
     4     9     2
     3     5     7
     8     1     6
     ------------
     4     3     8
     9     5     1
     2     7     6
     ------------
     6     7     2
     1     5     9
     8     3     4
     ------------
     6     1     8
     7     5     3
     2     9     4
     ------------
     8     3     4
     1     5     9
     6     7     2
     ------------
     8     1     6
     3     5     7
     4     9     2
     ------------
```

Så alla ser väsentligen likadana ut (omkastning av rader, kolonner etc). Att gör motsvarande för ordning fyra skulle ta tiotals år. Antag att vi kan kontrollera 10^4 fall per sekund. Tiden i år är då

$$\frac{16!}{10^4 \cdot 60 \cdot 60 \cdot 24 \cdot 365} \approx 66.3$$

■

Exempel. Lösning av linjära ekvationssystem, $\boldsymbol{Ax} = \boldsymbol{b}$, med rekursion. Detta är ingen bra metod. Den är ineffektiv, kräver mycket minne och kan ge stora fel. Den är rätt kort och elegant dock. Låt oss partitionera $\boldsymbol{A}$, $\boldsymbol{x}$ och $\boldsymbol{b}$, på följande sätt:

$$\boldsymbol{A} = \begin{bmatrix} \tilde{\boldsymbol{A}} & \boldsymbol{a} \\ \boldsymbol{c}^T & \boldsymbol{\alpha} \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} \tilde{\boldsymbol{x}} \\ \boldsymbol{\xi} \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} \tilde{\boldsymbol{b}} \\ \boldsymbol{\beta} \end{bmatrix}$$

Givet att $\boldsymbol{A}$ är en $\boldsymbol{n} \times \boldsymbol{n}$-matris så är $\tilde{\boldsymbol{A}}$ en $(\boldsymbol{n}-1) \times (\boldsymbol{n}-1)$-matris. $\boldsymbol{a}$ och $\boldsymbol{c}$ är kolonnvektorer med $\boldsymbol{n}-1$ element. $\boldsymbol{\alpha}$, $\boldsymbol{\xi}$ och $\boldsymbol{\beta}$ är skalärer. $\boldsymbol{Ax} = \boldsymbol{b}$ kan, med de nya beteckningarna, skrivas:

$$\begin{cases} \tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} + \boldsymbol{a}\boldsymbol{\xi} = \tilde{\boldsymbol{b}} \\ \boldsymbol{c}^T\tilde{\boldsymbol{x}} + \boldsymbol{\alpha}\boldsymbol{\xi} = \boldsymbol{\beta} \end{cases}$$

Antag att $\boldsymbol{\alpha} \neq 0$. Den andra ekvationen ger då

$$\boldsymbol{\xi} = \frac{\boldsymbol{\beta} - \boldsymbol{c}^T\tilde{\boldsymbol{x}}}{\boldsymbol{\alpha}}$$

vilket i den första ger:

$$\tilde{\boldsymbol{A}}\tilde{\boldsymbol{x}} + \boldsymbol{a}\frac{\boldsymbol{\beta} - \boldsymbol{c}^T\tilde{\boldsymbol{x}}}{\boldsymbol{\alpha}} = \tilde{\boldsymbol{b}} \Leftrightarrow \left(\tilde{\boldsymbol{A}} - \frac{\boldsymbol{a}\boldsymbol{c}^T}{\boldsymbol{\alpha}}\right)\tilde{\boldsymbol{x}} = \tilde{\boldsymbol{b}} - \frac{\boldsymbol{\beta}}{\boldsymbol{\alpha}}\,\boldsymbol{a}$$

vilket är ett nytt, mindre, ekvationssystem. Givet $\tilde{\boldsymbol{x}}$ kan vi så bilda

$$\boldsymbol{x} = \begin{bmatrix} \tilde{\boldsymbol{x}} \\ \boldsymbol{\xi} \end{bmatrix} = \begin{bmatrix} \tilde{\boldsymbol{x}} \\ \frac{\boldsymbol{\beta} - \boldsymbol{c}^T\tilde{\boldsymbol{x}}}{\boldsymbol{\alpha}} \end{bmatrix}$$

Eftersom vi kan uttrycka vårt problem i termer av ett mindre problem är det möjligt att tillämpa rekursion. Vi avslutar rekursionen när vi har ett linjärt ekvationssystem med en ekvation. Lösningen är då $\boldsymbol{x} = \boldsymbol{b}/\boldsymbol{A}$ (alla ingående storheter är ju då tal).
Här kommer MATLAB-koden, första versionen:

```
function x = solve1(A, b)
% Solve a linear system using recursion.

n = length(A);
if n == 1
  x = b / A;              % one equation
else
```



```
  n1    = n - 1;          % temporary quantity
  At    = A(1:n1, 1:n1);  % At = A-tilde, etc.
  a     = A(1:n1, n);
  c     = A(n, 1:n1)';    % note transpose
  alpha = A(n, n);
  bt    = b(1:n1);
  beta  = b(n);

  xt    = solve1(At - a * c' / alpha, ...
                 bt - (beta / alpha) * a);
  x     = [xt; (beta - c' * xt) / alpha];
end
```

Här ett körningsexempel:

```
>> A = rand(4)
A =
   6.4732e-01   2.4498e-01   1.3710e-02   2.4782e-01
   9.7540e-01   2.1001e-01   8.5819e-01   8.7729e-01
   3.8765e-01   4.2133e-01   1.9769e-01   9.5714e-01
   6.8748e-01   2.6403e-01   3.1185e-01   8.9106e-01

>> b = rand(4, 1)
b =
   1.9808e-01
   4.3550e-01
   2.0933e-01
   5.9412e-01

>> x = solve1(A,b)
x =
   7.2898e-01
  -2.1117e+00
  -8.5738e-01
   1.0301e+00

>> x - A \ b   % bättre metod
ans =
```



```
   1.1102e-16
  -8.8818e-16
  -5.5511e-16
   2.2204e-16
```

Vi behöver inte bilda variablerna **At** etc. Här är en kortare variant:

```
function x = solve(A, b)

n = length(A);
if n == 1
  x = b / A;
else
  n1 = n - 1;
  xt = solve( ...
    A(1:n1, 1:n1)-A(1:n1, n)*A(n, 1:n1)/A(n, n), ...
           b(1:n1)-(b(n)/A(n, n))*A(1:n1, n) );
  x = [xt; (b(n) - A(n, 1:n1) * xt) / A(n, n)];
end
```

Låt oss lösa ett större problem:

```
>> A = rand(500);
>> b = rand(500, 1);
>> tic; x = solve(A, b); toc
??? Maximum recursion limit of 300 reached.
Use set(0,'RecursionLimit',N) to change the
limit. Be aware that exceeding your available
stack space can crash MATLAB and/or your computer.

Error in ==> solve at 8
  xt = solve( ...

>> set(0, 'RecursionLimit', 500)
>> tic; x = solve(A, b); toc
Elapsed time is 2.189567 seconds.

>> tic; y = A \ b; toc
Elapsed time is 0.075785 seconds.
```

```
>> norm(x - y)
ans = 6.1780e-11

>> 2.189567 / 0.075785
ans = 2.8892e+01
```

Metoden behöver inte fungera:

```
>> A = [0 1; 1 0], b = [1; 2]
A =
     0     1
     1     0
b =
     1
     2
>> x = solve(A,b)
Warning: Divide by zero.
> In solve at 8
Warning: Divide by zero.
> In solve at 8
Warning: Divide by zero.
> In solve at 11
x =
   NaN
   NaN
>> A \ b
ans =
     2
     1
```

■

17 Något mer om datastrukturer

Den vanligaste datastrukturen i MATLAB är matrisen (vektorn). Den har flera fördelar:

- Användbar i många sammanhang.
- Kompakt lagring.
- Direkt och snabb åtkomst av alla element.
- Räcker med tämligen enkla algoritmer för att söka i en vektor, eller för att sortera elementen.

Det finns dock nackdelar, t.ex.

- Dyrt att lägga till element i vektorn.
- Vanliga vektorer kräver element av samma typ

Cellvektorer löser det sista problemet, men kostar mer i t.ex. minne:

```
>> c = {};
>> for k = 1:100, c{k} = rand(10, 1); end

>> M = [];
>> for k = 1:100, M(:, k) = rand(10, 1); end

>> whos
  Name        Size              Bytes  Class     Attributes

  M         10x100              8000  double
  c          1x100             14000  cell
```

Valet av datastruktur beror på vad som är viktigast: liten minnesåtgång, snabb åtkomst, enkelt att söka, billigt att utöka, bekvämlighet...

Här följer en kort presentation av några vanliga datastrukturer.

17.1 Stack (eng. stack)

En stack är en datastruktur där man bygger en trave av data, ungefär som en stapel av mynt eller tallrikar i ett sådant tallriksställ som man kan finna hos en lunchrestaurang. Det är enkelt att lägga till data på stacken, man lägger bara en ny tallrik överst (eng. push).
En stackpekarvariabel håller reda på var stackens översta element finns.

Det är också enkelt att ta översta tallriken från stacken (eng. pop). Det går dock inte att komma åt data längre ned i stacken, utan att data ovanför har avlägsnats. Man talar ibland om LIFO-stack (Last In First Out), till skillnad från en FIFO-stack (First In First Out).

Stackar används t.ex. vid parameteröverföring (parametrarnas adresser läggs på en stack).
Följande använder jag varje dag (**tcsh** eller **bash**):

```
% pushd ~/Diverse/Prog   pusha aktuell katalog
% arbeta ...             på katalogstacken
                         gå till ~/Diverse/Prog
% popd                   popa, återvänd
```

17.2 Länkad lista (eng. linked list)

En länkad lista kan liknas vid en utbyggbar vektor. En hyfsad analogi är ett godståg. Man har en variabel (pekare, handtag) som pekar på loket. Loket pekar på första godsvagnen som i sin tur pekar på nästa. Den sista vagnens pekare är en sk nollpekare (pekar inte på något).

```
head -> lok -> vagn -> vagn -> ... -> vagn -|
```

Vi kan tänka oss att pekaren innehåller en minnesadress. Så, lok och vagnar (kallas noder) innehåller dels pekaren, men även data.

Man har inte direktåtkomst (eng. direct access) till noderna, utan för att komma till en viss vagn måste man passera alla vagnar mellan lok och den aktuella vagnen (åtminstone om vi bara har tillgång till **head**).

Bilden visar en enkel-länkad lista, vi kan bara gå från vänster till höger. Om man dubblerar utrymmet för pekare, kan man göra en dubbel-länkad lista. Vi kan då gå åt båda hållen:

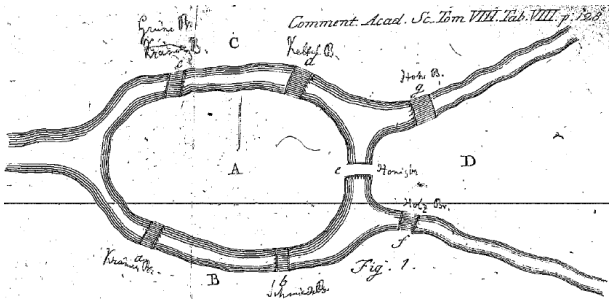
```
head <-> lok <-> vagn <-> vagn <-> ... <-> vagn -|
```

Det är enkelt att lägga till eller ta bort en nod från listan.

17.3 Graf

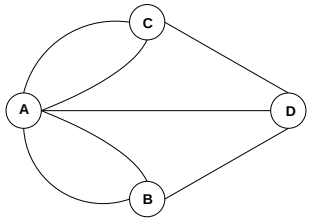
Säg att vi vill bygga upp en struktur för att hålla reda på städer och vägar mellan dessa. En naturlig datastruktur är då en graf. Städerna kallas noder (eng. nodes, vertices) och förbindelserna kanter (eng. edges).

Ett klassiskt problem som lämpar sig för grafteori, är Königsbergs broar (nu Kaliningrad ∈ Ryssland).



Problemet är att passera de sju broarna exakt en gång och komma tillbaks till utgångspunkten. Euler visade, 1736, att problemet inte

kan lösas. Bilden är hämtad ur Eulers artikel "Solutio problematis ad geometriam situs pertinentis". En lämplig graf kan se ut som:



Noderna svarar mot stränderna och kanterna mot broarna.
Ett annat problem, som inte Euler studerade, vore att införa en broavgift och bestämma den billigaste rundturen (man måste då passera en eller flera broar mer än en gång). En lämplig graf har då priser på kanterna, man talar om en *viktad graf*. Man kan tänka sig att det är olika broavgifter beroende på vilken riktning man passerar bron. Man inför då riktningar på kanterna, *riktad graf*. Man kunde också använda vikter för att beteckna avstånd mellan städer.

Grafer är också intressanta när man studerar glesa matriser (matriser med många nollor). Man kan också använda glesa matriser för att representera en graf. Om vi har n noder svarar grafen mot en matris G av ordning n, där $g_{j,k} \neq 0$ när det finns en kant mellan j och k. Vi kan använda elementen för ange vikter (skilda från noll). En osymmetrisk matris kan representera en riktad graf, $g_{j,k} \neq 0$ men $g_{k,j} = 0$ till exempel.

Grafer ger upphov till en stor mängd programmeringsproblem.

- Finns det en väg (eng. path) mellan två noder.
- Hitta den kortaste vägen.
- Sökning efter data i noder.
- Är grafen sammanhängande eller består den av flera delgrafer.

En cykel i en graf, är en väg (längd > 0) genom grafen, där man kan komma tillbaks till den nod man startade i. En graf utan cykler, en acyklisk graf, kallas träd.

Ofta har man en nod som kallas rot. Träd, i datavetenskap, brukar ritas med roten överst och "grenarna" nedåt. Noder längre ned i trädet kallas dotternoder eller liknande (eng. child) och på nivån ovanför finns föräldrar (eng. parent). I ett binärt träd har en förälder två dotternoder. En nod på lägsta nivån, en som inte har någon dotter, kalla löv eller terminal nod.

Träd är användbara när man hanterar hierarkiska strukturer. För att söka i träd används normalt rekursiva program (funktioner som anropar sig själva).
Filsystemet i unix kan representeras som ett träd. Roten markeras med `/`. En vanlig användning av träd är när man analyserar uttryck (eng. parse), att parsa på svengelska.

I MATLAB lagras plottar i en trädstruktur. Roten är en abstraktion, men bildskärmen är en lämplig analogi. Figurfönstren är dotternoder till roten. I ett fönster finns ett eller flera axelobjekt (ser ut som poster), dessa är döttrar till figurfönstret. Ett axelobjekt kan ha flera döttrar, t.ex. **`title`** och **`xlabel`** objekt. Dessutom är plotdata, x- och y-koordinater, linjestil, plotsymboler osv, samlade i ett dotterobjekt. Detta objekt är ett löv i trädet.

De olika nivåerna i trädet knyts ihop av handtag (eng. handle graphics) som vi kan se som ett slags pekare. Varje nod (post) innehåller två datamedlemmar, **`Children`** och **`Parent`** som pekar ned till eventuella dotternoder och upp till föräldranoden.

En egenhet hos handtagen är att de kan tolkas som 64-bitars flyttal. Det beror på att man använder 64 bitar för handtagen. Man skall inte använda talen utan handtagen måste lagras i variabler. Roten har dock handtag noll och figurfönstren ett, två, ...

Många funktioner returnerar handtag till det objekt som kommandot skapas.

```
>> hf = figure
hf = 1

>> x = linspace(0, pi);
>> hp = plot(x, sin(x))
hp = 1.581832275390625e+02

>> hx = xlabel('x')
hx = 1.591827392578125e+02

>> hy = ylabel('y')
hy = 1.591827392578125e+02

>> ht = title('En sinuskurva')
ht = 1.601827392578125e+02
```

`hf` är ett handtag till fönstret och **`hp`** är ett handtag till det löv där plotdata lagras etc. Om man skriver **`plot(x, sin(x))`** returneras inte handtaget. Data lagras alltså i en postliknande struktur. där postens datamedlemmar har värden. Namnen på datamedlemmarna kallas i detta sammanhang "property names" och värdena kallas "property values". Med kommandot **`get`** kan vi hämta fram namn och värden, och med kommandot **`set`** kan vi ändra på värden. Här kommer några hårt bantade exempel:

```
>> get(hf)
        Color = [0.8 0.8 0.8]
        Position = [803 469 560 420]
        Units = pixels
        WindowButtonDownFcn =
        WindowButtonMotionFcn =
        WindowButtonUpFcn =
        Children = [157.183]
        Parent = [0]
        Visible = on
```



```
>> child = get(hf, 'Children')
child = 1.571827392578125e+02   % axlarna

>> get(child)
        Box = on
        FontName = Helvetica
        FontSize = [10]
        FontUnits = points
        FontWeight = normal
        XLabel = [159.183]
        XAxisLocation = bottom
        XLim = [0 3.5]
        XScale = linear
        XTick = [ (1 by 8) double array]
        XTickLabel =
                0
                0.5
                1
                1.5
                2
                2.5
                3
                3.5
        Children = [158.183]

>> set(gca, 'XAxisLocation') % gca = get current axis
[ top | {bottom} ]

% flyttar axelgraderingen till överkanten
>> set(gca, 'XAxisLocation', 'top')

>> get(hp)
                Color: [0 0 1]
           EraseMode: 'normal'
           LineStyle: '-'
           LineWidth: 5.000000000000000e-01
```

```
              Marker: 'none'
          MarkerSize: 6
      MarkerEdgeColor: 'auto'
      MarkerFaceColor: 'none'
               XData: [1x100 double]
               YData: [1x100 double]
               ZData: [1x0 double]
        BeingDeleted: 'off'
      ButtonDownFcn: []
            Children: [0x1 double]
```

Man kan ändra på ett datavärde genom att ändra **XData** och **YData**, utan att rita om hela plotten. Detta utnyttjar man vid animering för att få snabbare bildväxlingar.

18 GUIs

GUI betyder Graphical User Interface. En användare använder datormusen och tangentbordet för att, via olika slags knappar, menyer och rullister, kommunicera med ett program. Det är mycket enkelt att skapa dessa grafiska objekt i MATLAB, åtminstone jämfört med att göra motsvarande i X11 (det underliggande fönstersystemet på unix-maskinerna).

Det är dock inte så enkelt att utforma ett bra GUI eller att skriva den underliggande koden. En svårighet är ju att användare skiljer sig åt; vad en användare tycker är naturligt är onaturligt för en annan användare.

Programmeringsstilen skiljer sig också från de program vi sett i kursen. Programmeraren vet ju inte i vilken ordning som knappar och andra objekt kommer att användas. Standardtekniken är att binda en funktion, en sk *callback* till varje knapp. När (om) en användare klickar på knappen så anropas callback-rutinen.
Om användaren aldrig trycker på knappen anropas inte rutinen (om den inte anropas av någon annan anledning).

I denna kurs introducerar jag bara dessa begrepp. Om du vill lära dig mer har jag en masterskurs om vetenskaplig visualisering som tar upp detaljerna. MATLAB har också en omfattande manual.
Skall man skapa många eller komplicerade GUIs lönar det sig nog att lära sig hantera MATLABs GUI-byggare, **guide**, ett program för att konstruera GUIs.

Här ett första exempel. Följande program skapar en röd knapp i en fönster. När vi trycker på knappen växlar knappens färg mellan röd och cyan. Jag har lagt båda funktionerna i samma fil, för att inte behöva hålla reda på så många filer.


```
function first_ex
hf = figure; % create a plot window; hf is handle
uicontrol(hf,                ...  % the handle
 'Style', 'pushbutton',      ...  % type of object
 'Units', 'centimeter',      ...
 'Position', [1 1 2 2],      ...  % x, y, width, height
 'BackgroundColor', [1 0 0], ...  % a red button
 'Callback', @button_cb);         % called when clicked

%
% This callback routine is called when we click on the
% button. Matlab will supply the arguments,
% handle and event_str.
%
function button_cb(handle, event_str)
red = [1 0 0];

if all(get(handle, 'BackgroundColor') == red)
  color = ~red; % switch to [0 1 1], cyan
else
  color = red;  % switch to red
end
set(handle, 'BackgroundColor', color)
```

hf är ett handtag till ett nytt fönster. Vi skapar en knapp i detta fönster. Knappen skapas av rutinen **uicontrol**, vilken som första parameter tar handtaget. **uicontrol** returnerar ett handtag till knappen, men jag väljer att inte spara detta handtag.
En knapp har flera egenskaper, en är **Style**, typen av knapp. **Position** ger knappens position relativt fönstrets nedre vänstra hörn. Först x- och y-koordinaterna, 1 cm (**Units**), för knappens nedre vänstra hörn, sedan bredd och höjd.

Därefter kommer knappens färg, en vektor med röd, grön, blå komponenter, en RGB-trippel. Min är noll och max ett. Slutligen ger vi ett funktionshantag till callack-rutinen.

Om en användare klickar på knappen, kommer MATLAB att anropa callack-rutinen och dessutom tillhandahålla två argument, **handle** samt **event_str** (event structure). Vi måste skriva vår funktionsdeklaration med dessa variabler, även om de inte används i koden (vi kan givetvis ha andra namn).

handle är handtaget till det objekt som gav upphov till anropet av rutinen (i detta fall är **handle** handtaget till knappen). **event_str** innehåller inget i detta exempel. Efter if-satsen sätter vi den nya färgen med **set**.

Ett exempel där **event_str** inte är tom, får vi om vi definierar en **KeyPressFcn** till ett fönster. Trycker på en tangent, med musen i fönstret, anropas callback-rutinen och **event_str** innehåller då information om vilken knapp vi tryckt på och om någon modifierar-knapp (shift, alt etc.) varit nedtryckt.

19 Lite mer flervariabelgrafik

Exempel. Säg att vi, i Matlab, vill rita:

$$f(x,y) = e^{x^2\sqrt{1-x^2-y^2}}$$

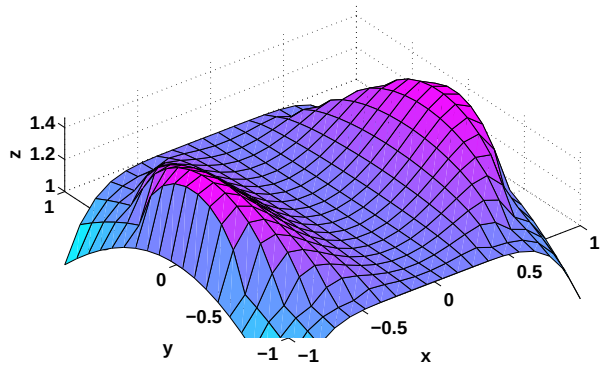
Detta leder till problem eftersom man i Matlab arbetar med rektangulära rutnät. Den största definitionsmängden, D, för f utgörs dock inte av en rektangel, utan av:

$$D = \{(x,y) : 1 - x^2 - y^2 \geq 0\} = \{(x,y) : x^2 + y^2 \leq 1\}$$

(Den största, eftersom vi kan tänka oss en mindre mängd.)

Följande bild visar det uppenbara sättet att försöka rita ytan, ungefär så här:

```
[X, Y] = meshgrid(linspace(-1, 1, 20));
Z = exp(X.^2 .* sqrt(1 - X.^2 - Y.^2));
surf(X, Y, real(Z)) % real, annars klagar Matlab
axis equal
```



Detta ger dock en felaktig bild, eftersom funktionen beräknas för (x,y) utanför D. Eftersom Matlab kan räkna med komplexa tal blir därför en del z-värdena komplexa. Matlab klagar om vi försöker rita ut dessa värden.

Vi får en bättre bild genom att nollställa (eller avlägsna) alla z-värden vars imaginärdelar är skilda från noll. Bilden är nu inte felaktig, men definitionsmängden är mindre än D.

För att få en snygg och korrekt bild gör vi ett koordinatbyte till polära koordinater.

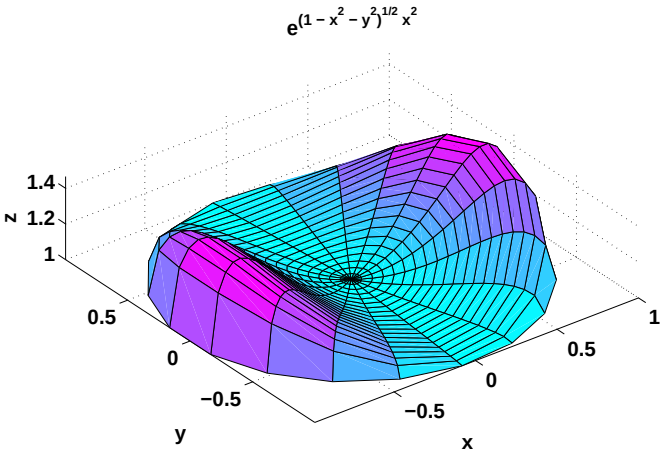
$$\begin{cases} x = r\cos\varphi \\ y = r\sin\varphi \end{cases}$$

Med r och φ som nya variabler kan ytan skrivas

$$z = e^{r^2\cos^2\varphi\sqrt{1-r^2\cos^2\varphi-r^2\sin^2\varphi}} = e^{r^2\cos^2\varphi\sqrt{1-r^2}}$$

Det är inte svårt att beskriva D med dessa nya variabler, $0 \leq r \leq 1$, $0 \leq \varphi \leq 2\pi$. Här följer lite Matlabkod och bilden:

```
[R, F] = meshgrid(linspace(0, 1, 20), ...
                  linspace(0, 2*pi, 20));
X = R .* cos(F);
Y = R .* sin(F);
Z = exp(X.^2 .* sqrt(1 - R.^2));
surf(X, Y, Z)
```

Vi får ett rutnät i r-φ-planet som svarar mot ett deformerat nät i x-y-planet.

När jag plottade ritade jag vissa punkter två gånger. $r = 1, \varphi = 0$ och $r = 1, \varphi = 2\pi$ ger båda $(x,y) = (1,0)$. Analogt ger $r = 0$, för godtyckligt φ, samma x-y-värde, $(x,y) = (0,0)$. Detta gör nu ingen skada för grafiken, utan tvärtom vill vi sluta ytan, så att vi inte får några springor.

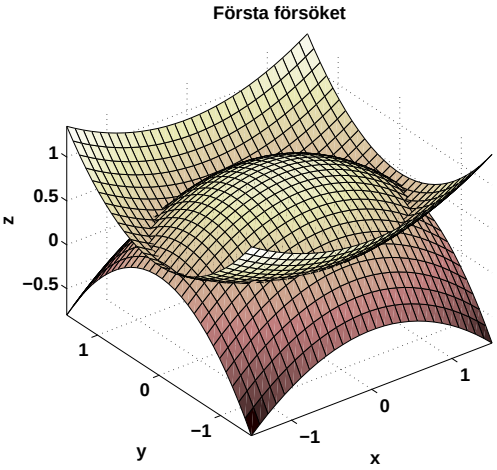
Det utgör dock ett problem om vi vill att den inversa avbildningen skall existera. ■

Exempel. Säg att vi vill visualisera kroppen som innesluts av ytorna

$$z = 0.2x^2 + 0.4y^2 \quad \text{och} \quad z = 1 - (0.3x^2 + 0.5y^2)$$

Att räkna ut volymen av kroppen vore en standarduppgift i en flervariabelkurs (träning på dubbelintegraler). Här är första försöket.

```
[X, Y] = meshgrid(linspace(-1.5, 1.5, 30));
Z1 = 0.2 * X.^2 + 0.4 * Y.^2;
Z2 = 1 - (0.3 * X.^2 + 0.5 * Y.^2);

figure(1)
hold off
surf(X, Y, Z1)
hold on
surf(X, Y, Z2)
axis equal
colormap pink  % byte av färgtabell
```



Om vi bara vill rita kroppen kan vi göra så här. Tag fram ekvationen för skärningskurvan:

$$0.2x^2 + 0.4y^2 = 1 - (0.3x^2 + 0.5y^2) \Leftrightarrow 0.5x^2 + 0.9y^2 = 1$$

Inför elliptisk-polära koordinater:

$$\begin{cases} x = a\,r\cos\varphi \\ y = b\,r\sin\varphi \end{cases}$$

där vi bestämmer a och b så att

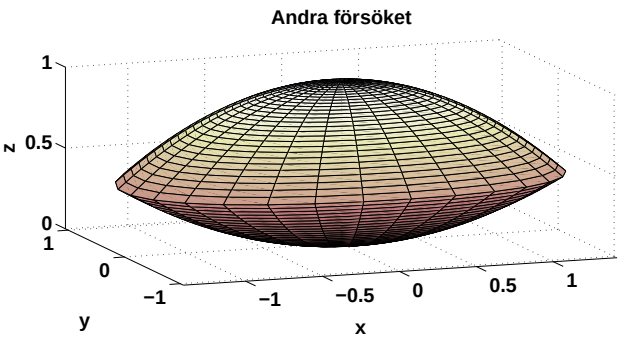
$$0.5(a\,r\cos\varphi)^2 + 0.9(b\,r\sin\varphi)^2 = 1$$

för konstant $r = 1$, så $a = 1/\sqrt{0.5}$ och $b = 1/\sqrt{0.9}$ går bra. När $r \in (0,1)$ och $\varphi \in (0, 2\pi)$ så bildar (x, y) punkter i ellips-skivan.

Så här ser MATLAB-koden ut:

```
[R, PHI] = meshgrid(linspace(0, 1, 30), ...
                    linspace(0, 2 * pi, 30));

X  = (1 / sqrt(0.5)) * R .* cos(PHI);
Y  = (1 / sqrt(0.9)) * R .* sin(PHI);
Z1 = 0.2 * X.^2 + 0.4 * Y.^2;
Z2 = 1 - (0.3 * X.^2 + 0.5 * Y.^2);

figure(2)
hold off
surf(X, Y, Z1)
hold on
surf(X, Y, Z2)
axis equal
colormap pink
view(-20, 10)   % nytt kommando
```

och här är bilden:



20 Filhantering, I/O

Säg att vi vill skriva en snygg tabell där varje kolonn tar ett visst antal positioner, där decimalpunkter står i kolonner etc.
Vi får då ange ett format när vi skriver ut. Här ett exempel:

```
>> x = logspace(1, 10, 10)';
>> [(1:10)', x, log(x)]
ans =
   1.0000e+00   1.0000e+01   2.3026e+00
   2.0000e+00   1.0000e+02   4.6052e+00
   3.0000e+00   1.0000e+03   6.9078e+00
   4.0000e+00   1.0000e+04   9.2103e+00
   5.0000e+00   1.0000e+05   1.1513e+01
   6.0000e+00   1.0000e+06   1.3816e+01
   7.0000e+00   1.0000e+07   1.6118e+01
   8.0000e+00   1.0000e+08   1.8421e+01
   9.0000e+00   1.0000e+09   2.0723e+01
   1.0000e+01   1.0000e+10   2.3026e+01
```

Det ser ju inte så snyggt ut. I första kolonnen vill vi bara ha heltal, I den tredje vill vi kanske inte ha någon exponentdel angiven. Så här kan man göra:

```
>> fprintf('%2d %15.4e %12.4f\n', ...
               [(1:10)', x, log(x)]')
 1      1.0000e+01       2.3026
 2      1.0000e+02       4.6052
 3      1.0000e+03       6.9078
 4      1.0000e+04       9.2103
 5      1.0000e+05      11.5129
 6      1.0000e+06      13.8155
 7      1.0000e+07      16.1181
 8      1.0000e+08      18.4207
 9      1.0000e+09      20.7233
10      1.0000e+10      23.0259
```


Formatet utgörs av strängen **'%2d %15.4e %12.4f\n'** och det beskriver hur en rad i tabellen skall utformas. Det är samma syntax som i C.

% talar att en formatbeskriving börjar, så **%2d** säger att här skall ett heltal (**d**) skrivas ut och två positioner skall användas. Man hade kunnat skriva **%d** och hade då fått en standardvidd.

%15.4e anger 15 positioner för ett flyttal på exponentform, 4:an anger antalet decimaler. Blanktecknet mellan **d** och **%** ger en blank i tabellen. **%12.4f** anger 12 positioner för ett flyttal utan exponent, 4:an anger antalet decimaler. **\n** står för newline (ny rad).

Data skrivs ut kolonnvis, det är därför jag har transponerat matrisen. **disp** gör att vi slipper få med variabelnamnet, **ans**.
help sprintf för fler formatkoder.
man -s3 printf, i ett terminalfönster, för C-varianten.

20.1 Inläsning

Inläsning från tangentbordet.

```
>> n = input('How many nodes? ')
How many nodes? 10
n = 10

>> s = input('Give a string: ', 's')
Give a string: a string
s = a string
```

Vi vill också kunna läsa från och skriva på fil. Det finns många rutiner för detta och jag tar inte upp alla varianter. **help iofun** ger en *lång* lista på rutiner.

Enklaste sättet att spara data på fil är att skriva

```
% spara alla variabler på matlab.mat
save
```

```
% spara alla variabler på filnamn.mat
save filnamn

% spara var1, var2 på filnamn.mat
save filnamn var1 var2
```

Filen är en sk binärfil, den är maskinläsbar (eng. machine-readable) och inte människoläsbar (eng. human-readable), med andra ord. "Människoläsbar" or en ovanlig term men maskinläsbar är lite vanligare. På engelska är båda orden vanliga.

Google ger 41 träffar för människoläsbar, 1310 för maskinläsbar, 2.2e6 för human-readable (med eller utan -) och 1.59e6 för machine-readable. Ibland talar man om "plain text files" eller "ascii files" när människoläsbara filer avses.

Här ett exempel:

```
>> clear
>> A = magic(3)
A = 8     1     6
    3     5     7
    4     9     2
>> b = rand(4, 1)
b = 4.7786e-01
    7.6375e-01
    8.6388e-01
    2.4192e-02

>> save testfil A b   % SPARA

>> clear
>> load testfil       % LÄS IN
>> A
A = 8     1     6
    3     5     7
    4     9     2
>> b
```



```
b = 4.7786e-01
    7.6375e-01
    8.6388e-01
    2.4192e-02
```

Så variabelnamnen ligger lagrade i filen, i detta fall.

Man kan använda **load/save** för att hantera människoläsbara filer, i viss utsträckning, som i detta exempel:

```
>> A = magic(3)
A = 8     1     6
    3     5     7
    4     9     2
>> b = rand(4, 1)
b = 8.0555e-01
    5.5425e-01
    6.4956e-01
    1.9222e-01

>> save testfil_1 A b -ascii
>> save testfil_2 A b -ascii -double
```

I unix:

```
% cat testfil_1
   8.0000000e+00   1.0000000e+00   6.0000000e+00
   3.0000000e+00   5.0000000e+00   7.0000000e+00
   4.0000000e+00   9.0000000e+00   2.0000000e+00
   8.0554743e-01
   5.5425212e-01
   6.4955918e-01
   1.9221567e-01

% cat testfil_2
   8.0000000000000000e+00   1.0000000000000000e+00   6.0(
   3.0000000000000000e+00   5.0000000000000000e+00   7.0(
   4.0000000000000000e+00   9.0000000000000000e+00   2.0(
   8.0554742811082347e-01
   5.5425211650678285e-01
```



```
   6.4955918344994601e-01
   1.9221566584977601e-01
```

Detta ger problem i Matlab:

```
>> load testfil_1
??? Error using ==> load
Number of columns on line 3 of ASCII file
testfil_1 must be the same as previous lines.
```

Observera att variabelnamnen försvinner i detta fall.

Här ett exempel som visar på lite allmännare filhantering (men det finns många färdiga MATLAB-funktioner som man kanske kan använda i stället).

```
fid = fopen('data_file','w');  % öppna fil,
                               % fid = file identifier
% samma data som tidigare
x = logspace(1, 10, 10)';

% skriv på filen
fprintf(fid, '%2d %15.4e %12.4f\n', ...
             [(1:10)', x, log(x)]');
fclose(fid);                        % stäng filen

>> type data_file                   % lista filen

 1      1.0000e+01       2.3026
 2      1.0000e+02       4.6052
 3      1.0000e+03       6.9078
 4      1.0000e+04       9.2103
 5      1.0000e+05      11.5129
 6      1.0000e+06      13.8155
 7      1.0000e+07      16.1181
 8      1.0000e+08      18.4207
 9      1.0000e+09      20.7233
10      1.0000e+10      23.0259
```



```
>> fid = fopen('data_file','r'); % öppna fil, r = read
>> A = fscanf(fid, '%d %e %f', [3, 10])
A =
  Columns 1 through 7
   1.0000e+00   2.0000e+00   3.0000e+00   4.0000e+00   5.
   1.0000e+01   1.0000e+02   1.0000e+03   1.0000e+04   1.
   2.3026e+00   4.6052e+00   6.9078e+00   9.2103e+00   1.
  Columns 8 through 10
   8.0000e+00   9.0000e+00   1.0000e+01
   1.0000e+08   1.0000e+09   1.0000e+10
   1.8421e+01   2.0723e+01   2.3026e+01

>> fclose(fid)
ans = 0

>> fid = fopen('data_file','r');  % öppna fil, r = read
>> A = fscanf(fid, '%d %e %f', [3, 10])'  % OBS '
A =
   1.0000e+00   1.0000e+01   2.3026e+00
   2.0000e+00   1.0000e+02   4.6052e+00
   3.0000e+00   1.0000e+03   6.9078e+00
   4.0000e+00   1.0000e+04   9.2103e+00
   5.0000e+00   1.0000e+05   1.1513e+01
   6.0000e+00   1.0000e+06   1.3816e+01
   7.0000e+00   1.0000e+07   1.6118e+01
   8.0000e+00   1.0000e+08   1.8421e+01
   9.0000e+00   1.0000e+09   2.0723e+01
   1.0000e+01   1.0000e+10   2.3026e+01

>> fclose(fid)
ans = 0

>> fid = fopen('finns_ej','r')
fid = -1   % får positivt heltal annars
```

Exempel. Här följer ett tentamensproblem (2008-08-20):
Vi har en uppsättning textfiler vars namn alla slutar på **.txt**. Skriv en rutin, **[file, n] = find_str(str)**, som letar efter strängen, som ges av variabeln **str**, i alla filer. Funktionen skall, i variabeln **file**, returnera namnet på den fil som har flest förekomster av strängen. Variabeln **n** skall innehålla antalet förekomster. Om det finns flera filer med samma antal, skall funktionen returnera namnet på den första filen (utifrån den ordning som ges av **dir**-kommandot). Om **str** inte finns i någon fil så skall **n** sättas till noll och **file** till en tom vektor **[]**. En (kort) fil kan se ut som:

```
I think that we might venture a little farther than
this. Look at it in this light. On what occasion
would it be most probable that such a presentation
would be made? When would his friends unite to give
him a pledge of their good will? Obviously at the
```

Om **str = 'the'** finns tre förekomster (**farther**, **their** och **the**). Observera att t.ex. **The** inte räknas.
Ledning: lite lätt modifierad **help dir**:

```
D = DIR('*.txt') returns the results in an M-by-1
 structure with the fields:
    name    -- filename
    date    -- modification date
    bytes   -- number of bytes allocated to the file
    isdir   -- 1 if name is a directory and 0 if not
    datenum -- modification date as a MATLAB serial
               date number
```

Mera ledning: **str = fgetl(fid)** läser in en rad från filen med "file identifier" **fid**. Raden lagras i teckenvektorn **str**. När filen är slut innehåller **str** talet (inte strängen) **-1**.
Ännu mera ledning: **k = strfind(str, pattern)** searches the string **str** for occurrences of a shorter string, **pattern**, and returns the starting index of each such occurrence in the double array **k**. If **pattern** is not found in **str**, or if **pattern** is longer than **str**, then **strfind** returns the empty array **[]**. (4p)


```
function [file, n] = find_str(str)

n    = 0;               % max number of hits
file = [];              % in case str is not found
files = dir('*.txt');   % all files ending in .txt
for k = 1:length(files) % for each such file
  hits = 0;             % hits in this file
  fid = fopen(files(k).name);

  lin = fgetl(fid);     % read the first line
  while ischar(lin)
%   Update no. of hits in the file:
    hits = hits + length(strfind(lin, str));
    lin = fgetl(fid);   % read a line
  end  % while
  fclose(fid);

% Update n and file
  if hits > n
    n = hits;
    file = files(k).name;
  end

end
```

```
>> [f, n] = find_str('the')
f = test_file.txt
n = 3

>> [f, n] = find_str('hej')
f = []
n = 0
```


21 C++, mer om datatyper

Det kommer nu en introduktion till C++. Den viktigaste anledningen är att man måste kunna lite mer om typer och deklarationstvång. Det är också bra att ha sett ett annat språk och känna till lite om kompilatorer.
Vi börjar med exemplet från introduktionen, där vi beräknade en approximation av $1 + 1/2 + \cdots + 1/1000$.

```
#include <iostream>   // header- or include files
using namespace std;

int main()            // integer function main
{                     // start of function body
  double sum;         // sum is double precision
  int      k;         // k is integer

  sum = 0.0;          // important
  // for k = 1, ..., 1000
  //   sum = sum + 1.0 / k

  for(k = 1; k <= 1000; k++)  // k++ or k = k + 1
    sum += 1.0 / k;           // sum = sum + 1.0 / k

  // print. endl = end-of-line
  cout << "1 + 1/2 + ... + 1/1000 = " << sum << endl;

  return 0;  // return status to the shell
}            // end of function body
```

Så här översätter man programmet till "maskinkod" och exekverar denna (mer detaljer om ett tag):

```
% g++ sum.cc                kompilera
% a.out                     exekvera
1 + 1/2 + ... + 1/1000 = 7.48547
```

Om du inte har **.** i din unix-path får du skrivas **./a.out** .

Några detaljer i programmet.

Headerfilen, **iostream**, läses in i koden. Denna fil är nödvändig eftersom vi använder utskrifter i programmet. Om man vill se vad filen innehåller (vill man normalt inte) så hittar man den i filsystemet, **/usr/include/c++/3.3.4/iostream**.
En sådan fil innehåller typiskt definitioner av konstanter, och sk funktionsprototyper. T.ex. har sinusfunktionen prototypen

```
double sin(double)
```

som säger att funktionen har namnet **sin**. Funktionen tar en parameter av typen **double** och funktionen returnerar ett double-resultat.
iostream innehåller dock definitioner av funktioner som behövs för I/O (inget om sinus, med andra ord). Det finns många header-filer och man måste läsa manualer etc. för att veta vilka man skall inkludera.
namespace är ett sk namnutrymme (ung. en behållare) som ger oss tillgång till definitionerna i headerfilen.
Huvudprogrammet måste heta **main** och det är en funktion. Det kan ta parametrar och returnerar ett statusvärde till shellet, där vi startade programmet.
Vi tar inte emot några parametrar, det står **main()**. Om vi vill komma åt inparametrar blir det lite krångligare, man får skriva:

```
int main(int argc, char *argv[])
```

Vi struntar dock i sådant för tillfället.
När vi ger kommandot **ls -l** (lång fil-listning) så exekveras ett C-program. Argumenten till motsvarande huvudprogram är namnet på kommandot **ls** och flaggan **-l**, samt antalet argument.

```
% ls -l *.cc
-rw-------  1 thomas _math 592 Apr 25 17:17 sum.cc
-rw-------  1 thomas _math 136 Apr 25 21:15 sum_vec.cc
% echo $status
0
```

```
% ls -l hajhgjdsgaygdqgd
ls: hajhgjdsgaygdqgd: No such file or directory
% echo $status
1
```

Som det står i

```
% info coreutils ls
```

"An exit status of zero indicates success, and a nonzero value indicates failure."
{ } avskiljer det man kallar funktionskroppen.
double sum; är en sk typdeklaration. Den reserverar minnesutrymme för en variabel, **sum**, av typen **double**. **double** är namnet på typen. På våra datorer får man ett 64-bitars flyttal. En annan flyttalstyp är **float** som på våra datorer ger ett 32-bitars flyttal. I MATLAB avslutas rader av radslutstecken, men i C++ måste man markera slutet med semikolon, **;**. Man hade kunnat skriva

```
double
sum;
```

En tämligen oläslig variant, utan kommentarer, ser ut som:

```
#include <iostream>
using namespace std;int main(){double sum;int k;
sum=0.0;for(k=1;k<=1000;k++)sum+=1.0/k;cout<<
"1␣+␣1/2␣+␣...␣+␣1/1000␣=␣"<<sum<<endl;return 0;}
```

Ett vanligt nybörjarfel är att glömma semikolon. Detta ger upphov till många, ibland svårtolkade, kompileringsfel.
Tar jag t.ex. bort semikolon efter **double sum** får jag

```
% g++ sum.cc
sum.cc: In function ‘int main()’:
sum.cc:7: error: expected initializer before "int"
sum.cc:9: error: ‘sum’ was not declared in this scope
sum.cc:13: error: ‘k’ was not declared in this scope
```


Typdeklarationen **int k;** ger oss ett heltal, **k** (32 bitar). **int** och **double** lagras på olika sätt internt och de har också olika beräkningsegenskaper. Om t.ex. **j** och **k** är två variabler av typen **int** så har kvoten **j / k** också ett heltalsvärde.
Så blir t.ex. kvoten **3 / 2** exakt **1** alla decimaler stryks,
sk heltalsdivision.

Raden **sum = 0.0;** initierar summationsvariabeln. Man hade kunnat skriva **sum = 0;** . Det finns dock en viss skillnad. **0.0** är en flyttalskonstant av typen **double** och **0** är en heltalskonstant av typen **int**. Dessa två konstanter lagras på olika sätt. När man skriver **sum = 0;** kommer en konvertering från **int** till **double** att utföras.
I uttrycket **sum += 1.0 / k;** har vi också en typkonvertering. Hade jag skrivit **sum += 1 / k;** hade vi fått heltalsdivision, och summan hade blivit ett. För alla **k** större än ett så blir **1 / k** noll. Vi typkonverteringen kommer den "mindre" typen, **int** , att konverteras till **double** . Om man tvekar det minsta, kan man konvertera explicit och inte förlita sig på automatiken.

```
sum += 1.0 / double(k);
```

Man måste vara försiktig när parenteser ingår:

```
2.0 * (1 / 2)        2.0 * 1 / 2
```

Värdet på det första uttrycket är **0.0**, ty **1 / 2** beräknas först till **0** som sedan konverteras till **0.0** som multipliceras med **2.0** och ger resultatet **0.0**.
I det andra uttrycket beräknas först **2.0 * 1** som blir **2.0**, efter att **1** har konverterats till **1.0**. Man har då uttrycket **2.0 / 2** som blir **1.0** efter konvertering av **2**.
for-satsen fungerar på följande vis:

```
for(initiering; testuttryck; uppdateringsuttryck)
  loop-kropp
```


Skrivet med en while-loop

```
initiering;
while ( testuttryck ) {
  loop-kropp
  uppdateringsuttryck;
}
```

Man kan tänka sig flera alternativa **for**-loopar, t.ex.:

```
k = 1;
for(; k <= 1000;) {
  sum += 1.0 / k;
  k++;
}

k = 1;
for(;;) {
  sum += 1.0 / k;
  if ( k == 1000 )
    break;         // Hoppa ur loopen
  k++;
}
```

Notera att **{ }** är nödvändigt för att gruppera ihop satserna i loopkroppen (om man har mer än en). Så betyder

```
k = 1;
for(; k <= 1000;)
  sum += 1.0 / k;
  k++;
```

något helt annat, man får en oändlig loop. Om man är en försiktig programmerare kan man alltid sätta ut **{ }**, t.ex.

```
for(k = 1; k <= 1000; k++) {
  sum += 1.0 / k;
}
```


Man kan föra en lång och hetsig diskussion om var man skall placera **{ }**. Ett alternativ (som jag inte gillar) är:

```
for(k = 1; k <= 1000; k++)
{
  sum += 1.0 / k;
}
```

MATLAB är interaktivt och kan exekvera ett program direkt.
Detta är ej fallet med C++, utan vi använder en kompilator, **g++**, för att översätta C++-programmet till assemblerspråk (maskinens eget språk).
Ett annat program, en assembler, översätter sedan assemblerprogrammet till ettor och nollor (ungefär) och lagrar resultatet på en så kallad objektfil.
Slutligen kombinerar ett tredje program, länkaren, objektfilen med rutiner från olika bibliotek, rutiner för I/O, matematikbibliotek om vi använder sinus t.ex. Resultatet läggs på filen **a.out** (assembler output). Denna fil är exekverbar och man exekverar programmet genom att skriva **a.out** .
För att se alla stegen som utförs av **g++** har jag slagit på **verbose**-flaggan. Här en del av det som skrivs:

```
% g++ -v sum.cc
/usr/libexec/gcc/i386-redhat-linux/3.4.6/cc1plus -quiet
 -dumpbase sum.cc -auxbase sum -Wall -version
 -o /tmp/ccckMdRr.s

as -V -Qy -o /tmp/ccmkFnB6.o /tmp/ccckMdRr.s

/usr/libexec/gcc/i386-redhat-linux/3.4.6/collect2
  --eh-frame-hdr -m elf_i386 -dynamic-linker /lib/ld-linu
 /usr/lib/gcc/i386-redhat-linux/3.4.6/../../../crt1.o
 /usr/lib/gcc/i386-redhat-linux/3.4.6/../../../crti.o
 /usr/lib/gcc/i386-redhat-linux/3.4.6/crtbegin.o -L/usr/
 -L/usr/lib/gcc/i386-redhat-linux/3.4.6 -L/usr/lib/gcc/i
 /tmp/ccmkFnB6.o -lstdc++ -lm -lgcc_s -lgcc -lc -lgcc_s
```

```
 /usr/lib/gcc/i386-redhat-linux/3.4.6/crtend.o
 /usr/lib/gcc/i386-redhat-linux/3.4.6/../../../crtn.o
```

/tmp/ccckMdRr.s är assemblerfilen och **/tmp/ccmkFnB6.o** objektfilen.

Assemblerfilen är lång, mer än 400 rader. Beräkning utförs på några rader och motsvarande assemblerkod ser ut så här.

```
                                    Mina kommentarer
.L11:
        cmpl    $1000, -12(%ebp)    k == 1000?
        jg      .L12                jump if > to .L12
        fildl   -12(%ebp)
        fld1
        fdivp   %st, %st(1)         1.0 / k
        fldl    -8(%ebp)
        faddp   %st, %st(1)         s += 1.0 / k
        fstpl   -8(%ebp)
        leal    -12(%ebp), %eax
        incl    (%eax)              k = k + 1
        jmp     .L11                jump to .L11
.L12:
```

fdivp och **faddp** står för floating point division respektive addition. **p** står för pop (från en stack). **%ebp** är namn på ett register i CPUn. **cmpl** jämför värdet på vår loopvariabel med 1000 och markerar resultatet med en bit i ett stausregister. **jg** tittar på denna bit och om vi är färdiga med loopen, så hoppar programmet till "label" **.L12**. I slutet av loopen ser **jmp** till att vi hoppar till början av loopen. **incl** ökar loopvariabelns värde med ett.

Man kan analysera detta i alla detaljer, vilket vi avstår ifrån för då måste vi gå in på hur register och stacken i flyttalsenheten fungerar.

Kan få ett snabbare (och mindre, i detta fall) program genom att begära att kompilatorn optimerar koden:


```
% g++ -O3 sum.cc
```

C++ har inget direkt stöd för mer avancerade numeriska beräkningar (linjära ekvationssystem, differentialekvationer etc) och det finns inget direkt stöd för grafik eller GUIs. Däremot kan man länka ihop sitt C++-program med externa numeriska bibliotek.

Varför använder man C++ då? Beräkningar utgör en rätt liten del av all datoranvändning och andra typer av program (ordbehandlare, operativsystem, kompilatorer, databassystem, spel) kräver andra slags programspråk.

C++ ger mycket bättre kontroll över minneshantering och har stöd för mer avancerade datastrukturer. Det går att skriva mycket större program i C++ än i MATLAB. Eftersom C++ kompileras kan vissa delar bli snabbare än i MATLAB (men MATLAB består ju av en hel del kompilerad kod).

Här följer nu ytterligare några C++-exempel.

Exempel. Skriv en funktion som summerar talen i en vektor av flyttal.

```
double sum_vec(double vec[], int n)
{
  int k;
  double sum;

  sum = 0.0;
  for(k = 0; k < n; k++)
    sum += vec[k];

  return sum;
}
```

Vektorer fungerar ungefär som i MATLAB, med följande viktiga skillnader:

- första index är noll och inte ett
- C++ har inte stöd för matrisalgebra (men man kan skaffa sig stöd genom att skapa klasser)
- ingen automatisk utvidgning av vektorer vid tilldelning
- C++ kontrollerar inte att index håller sig inom indexgränserna

Här följer ett huvudprogram com anropar **sum_vec**

```
#include <iostream>
using namespace std;

// A function prototype
double sum_vec(double [], int);

int main()
{
  int      n = 1000;
  double   w[n];  // space for w[0], ..., w[999]

  // k is local to the loop
  for(int k = 0; k < n; k++)
    w[k] = 1.0 / double(k + 1);

  cout << "vector sum = " << sum_vec(w, n) << endl;

  return 0;
}
```

```
% g++ sum_vec_main.cc sum_vec.cc   separatkompilering
% a.out
vector sum = 7.48547
```

Ett stort problem i språk som C++ är att man inte får någon varning vid indexfel. Om vi ändrar anropet av **sum_vec** och ljuger om vektorlängden

```
  cout << "vector sum = " << sum_vec(w, n+100) << endl;
```


notera **n+100**, får vi i detta fall svaret

```
% a.out
vector sum = -50.5759
```

Summans värde beror på vad som ligger efter sista elementet. I C++ (men ej i C) finns det mer avancerade vektorer (klasser) som har indexkontroll, men dessa vektorer ger normalt långsammare kod och passar inte ihop med numerisk programvara skriven i Fortran.

21.1 void-funktioner

En void funktion, en funktion utan returvärde, skrivs i C/C++

```
  void funktions_namn(parameterlista)
```

Observera att man inte kan skriva som i MATLAB:

```
  funktions_namn(parameterlista)
```

Det är förbjudet i C++ och i C får man en funktion av typen **int**. I andra språk skiljer man mellan funktioner som lämnar returvärde (kallas funktioner) och void-funktioner (kallas procedurer eller subrutiner).

Det är dålig programmeringsstil att ändra på parametrarna (eller t.ex. globala variabler) i en icke void-funktion, man säger att funktionen har sidoeffekter (eng. side effects).
Här är en anledning:

```
#include <iostream>
using namespace std;

bool side_effects();

int main()
{
```

```
  int choice;

  cout << "choice? ";  // no endl
  cin  >> choice;      // read

  // && = logical and, & = bitwise and
  if ( choice > 0 && side_effects() )
    cout << "then" << endl;
  else
    cout << "else" << endl;

  return 0;
}

// a boolean (logical) function
bool side_effects()
{
  cout << "in side_effects" << endl;

  return true;
}
```

```
% g++ side_effects.cc

% a.out
choice? 2
in side_effects
then

% a.out
choice? -1
else
```

Notera att lat evaluering används. **side_effects** anropas inte om **choice > 0** är **false**.

C++ ger mycket större kontroll över parameteröveföringen mellan funktioner. I exemplet ovan överförs **n** via värdeanrop (eng. call-by-value). Funktionen har en lokal variabel **n** som är en kopia av värdet i det anropande programmet. Om funktionen ändrar på värdet på **n** så ändras inte värdet i **main**. Utrymmet för den lokala variabeln deallokeras när exekveringen av funktionen avslutas (vid återhopp).

Det vore för slösaktigt att överföra fält på detta sätt, så fält överförs med referensanrop (eng. call-by-reference). Adressen till första elementet överförs till funktionen. Ett annat språkbruk är att en pekare till första elementet överförs till funktionen. Det gör att funktionen kommer åt alla elementen i vektorn så om funktionen ändrar i vektorn så finns ändringarna kvar efter återhopp från funktionen.

Man kan använda referensanrop även för skalära variabler, när man vill returnera värden, som i följande exempel:


```
#include <cmath>  // for cos
#include <iostream>
using namespace std;
void min_max(double [], int, double *, double *);

int main()
{
  int      n = 10;
  double   v[n], min_v, max_v;

  for(int k = 0; k < n; k++)
    v[k] = cos(1.0 / double(k + 1));

  min_max(v, n, &min_v, &max_v);     // note &

  cout << "min = " << min_v << endl;
  cout << "max = " << max_v << endl;

  return 0;
}

void min_max( double v[], int n,
  double *adr_min_v, double *adr_max_v )
{
  *adr_min_v = v[0];  // note *
  *adr_max_v = v[0];

  for(int k = 1; k < n; k++)
    if ( v[k] < *adr_min_v )
      *adr_min_v = v[k];
    else if ( *adr_max_v < v[k] )
      *adr_max_v = v[k];
}
```

```
% a.out
min = 0.540302
max = 0.995004
```


#include <cmath> inkluderar prototypen för bland annat cosinusfunktionen. **void** betyder att funktionen inte returnerar något värde i namnet, svarar mot en MATLAB-funktion som saknar utparameter.

&variabel är adressen (i minnet) till **variabel**. Vi skickar denna adress till funktionen. Inuti funktionen kommer vi åt minnescellen (värdet) genom att skriva ***variabel**
(eng. dereferencing, indirection).
Detta förklarar också deklarationen

```
void min_max( double v[], int n,
  double *adr_min_v, double *adr_max_v )
```

adr_min_v är en adress, så ***adr_min_v** är värdet, en **double**. Normalt skriver man *inte* **adr_** framför namnet, jag har gjort det i ett försök att vara pedagogisk. **adr_min_v** är alltså adressen till **min_v** och ***adr_min_v** på det som ligger på denna adress.
Här en bild som visar situationen i **main** och inuti **min_max**:

```
main                              min_max
                 Minne
           |                 |
           -------------------
& min_v -> | värdet på min_v | <-- adr_min_v
           | = *adr_min_v    |
           -------------------
           |                 |
           -------------------
           |                 |
```

Det skall sägas att man kan göra referensanrop snyggare i C++ (men inte i C).

I MATLAB är det enkelt att skapa nya variabler. I C++ måste man deklarera variabler, som i exemplen ovan. Man kan allokera sk automatiska variabler i en funktion:

```
void function(double s, int n)
{
  // vec is a variable-length automatic variable
  double vec[n];
  int temp; // temp is also an automatic variable
  ...
}
```

Automatiska variabler har begränsad livslängd, de deallokeras vid återhopp. Om man behöver minne med längre livslängd brukar man använda **new**-operatorn. Så här kan det se ut:

```
#include <iostream>
using namespace std;

void new_example(bool, int);

int main()
{
  new_example(true,  10);  // first call
  new_example(false, 10);  // last call
  return 0;
}

void new_example(bool first, int n)
{
  static double *vec; // keep vec between calls

  if ( first ) {
    vec = new double[n];  // allocate memory

    for(int k = 0; k < n; k++)
      vec[k] = 1.0 / double(k + 1);
  } else {
    cout << "vec[n-1] = " << vec[n-1] << endl;
    delete [] vec;  // free the memory
  }
}
```


vec är en pekare till minne som allokeras av **new**. För att vi inte skall tappa greppet om minnet (och få en sk minnesläcka) är pekaren **vec** en statisk variabel som behåller sitt minne mellan anropen. **delete**-operatorn frigör minnet när vi har använt det färdigt.
new, **delete** motsvaras av **malloc**, **free** i C.
Slutligen ett exempel med en post:

```
#include <iostream>
using namespace std;

// a structure
struct complex {
  double re, im;
};

complex add(complex, complex);
complex mul(complex, complex);
void print(complex);

int main()
{
  complex z, w;
  complex v[10];  // not used in the program

  cout << "Give two complex numbers: ";

  // read two complex numbers
  cin >> z.re >> z.im >> w.re >> w.im;

  cout << "The sum     = "; print(add(z, w));
  cout << "The product = "; print(mul(z, w));

  return 0;
}
```



```
complex add(complex z, complex w)
{
  complex sum;

  sum.re = z.re + w.re;
  sum.im = z.im + w.im;

  return sum;
}

complex mul(complex z, complex w)
{
  complex prod;

  prod.re = z.re * w.re - z.im * w.im;
  prod.im = z.re * w.im + z.im * w.re;

  return prod;
}

void print(complex z)
{
  cout << "(" << z.re << ", " << z.im << ")" << endl;
}
```

```
Give two complex numbers: 1 2 3 4
The sum     = (4, 6)
The product = (-5, 10)
```

Detta går att göra mycket snyggare med klasser, men det är en helt ny kurs.
Det finns dessutom stöd för komplex tal i standardbiblioteket. Så här kort blir programmet då:


```
#include <iostream>
#include <complex>
using namespace std;

int main()
{
  complex<double> z, w;

  cout << "Give two complex numbers: ";
  cin >> z >> w;

  cout << "The sum     = " <<  z + w  << endl;
  cout << "The product = " <<  z * w  << endl;
  cout << "exp(z)      = " <<  exp(z) << endl;

  return 0;
}
```

```
Give two complex numbers: (1, 2) (3, 4)
The sum     = (4,6)
The product = (-5,10)
exp(z)      = (-1.1312,2.47173)
```

complex är en klass och **complex<double>** säger att vi vill ha komplexa tal (objekt) där real- och imaginärdeler är av typen **double** (och inte **float**). Notera header-filen **<complex>**.

Bakom **«**, **»**, **+**, ***** och **exp** döljer sig överlagrade funktioner som är knutna till klassen **complex**. Notera att andra funktioner används för att beräkna t.ex. **x * y** då **x** och **y** är av typen **double**.

Man kan ofta använda klasser i C++ utan att kunna så mycket om OOP.

Man lever mera farligt i C++ än i MATLAB, eftersom MATLAB har fler kontroller (som kostar lite exekveringstid, men sparar programmerartid).
Här några exempel:

```
#include <iostream>
using namespace std;

void sub(double []);

int main()
{
  double  b[1], a[10];

  b[0] = 1;
  cout << "b[0] before sub: " << b[0] << endl;
  sub(a);
  cout << "b[0] after  sub: " << b[0] << endl;

  return 0;
}

void sub(double a[])
{
  a[11] = 12345;
}
```

```
% a.out
b[0] before sub: 1
b[0] after  sub: 12345
```

Verkar orimligt att **b[0]** är ändrat, eftersom **b** inte är en parameter till **sub**. Vi har ju inte heller globala variabler.

Om vi ändrar **a[100000] = 12345;** så "fungerar det":

```
% a.out
b[0] before sub: 1
b[0] after  sub: 1
```

a[1000000] = 12345; ger

```
% a.out
b[0] before sub: 1
Segmentation fault
```

Detta är ett av det värsta fel man kan få (ett annat av samma typ är **Bus error**). Programmet försöker att läsa från eller skriva till en minnesadress där programmet inte har rättigheter att göra detta. Det kan vara enormt svårt att hitta orsaken till dessa fel. Man har kanske räknat ut ett index fel och detta felaktiga index används sedan på ett helt annat ställe i koden och långt senare. Man kan försöka att använda en avlusare. Den kan kanske hitta det ställe där felet visar sig (om det nu gör det).

```
% g++ -g adr_err1.cc    OBS -g
% a.out
b[0] before sub: 1
Segmentation fault

% gdb a.out              starta "the GNU debugger"
GNU gdb Red Hat Linux (6.3.0.0-1.143.el4rh)
Copyright 2004 Free Software Foundation, Inc.  ...

(gdb) run
Starting program: a.out
b[0] before sub: 1

Program received signal SIGSEGV, Segmentation fault.
0x08048890 in sub (a=0xbffcebe0) at adr_err1.cc:20
20        a[1000000] = 12345;
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```


Skilj mellan tilldelning **=** och kontroll av likhet **==**.

```
#include <iostream>
using namespace std;

int main()
{
  for(int k = 0; k <= 5; k++)
    if ( k = 3 ) {
      // Do something special when k is equal to 3
      cout << "k = " << k << endl;
    }

  return 0;
}
```

```
% a.out
k = 3
k = 3
k = 3
k = 3
k = 3
 ...    ger en oändlig loop, bryt med ^C
```

Det skall vara **if (k == 3) {** osv.
Vad som händer i programmet är följande: I C/C++ så ger en tilldelning, **variabel = uttryck;**, ett värde, nämligen det värde **variabel** får. Man kan t.ex. nollställa flera variabler genom:

```
sum1 = sum2 = sum3 = 0;
```

ty **sum3 = 0;** har värdet **0** som tilldelas **sum2** etc.

I if-satsen får **k** värdet **3**, **k = 3** har alltså värdet **3**. Värdet svarar mot sant, varför **cout**-raden exekveras. Eftersom man dessutom ändrar på loop-variabels värde, kommer alltid villkoret, **k <= 5**, att vara uppfyllt och vi får en oändlig loop.

Nästlade if-satser.

"Trailing else" hör till den innersta if-satsen, så med korrekt indentering ser det ut så här:

```
if ( villkor )
  if ( annat villkor ) {
    satser
  } else {
    satser
  }
```

Om man vill koppla else till den yttre if-satsen får man skriva:

```
if ( villkor ) {
  if ( annat villkor ) {
    satser
  }
} else {
  satser
}
```

Denna tvetydighet finns inte i MATLAB, eftersom man alltid måste sätta ut **end**. Här är MATLAB-koden för uttrycken ovan:

```
if villkor
  satser
  if annat_villkor
    satser
  else
    satser
  end
end
```

```
if villkor
  satser
  if annat_villkor
    satser
  end
else
  satser
end
```

22 Top-down and bottom-up design

När man designar större program krävs en genomtänkt strategi. Det finns mycket skrivet om detta när det gäller objekt-orienterad programmering. Det finns också mycket skrivet om de speciella tekniker som krävs i samband med GUIs, realtidsprogrammering och parallellitet.

Vid stora projekt (som denna kurs inte handlar om) är det dessutom viktigt att ha en projektbeskrivning, en programmeringsstandard (om flera programmerare deltar), versionshantering av kod, kontinuerlig dokumentation och testning etc.

När det gäller traditionell programmering i MATLAB, C och Fortran så används ofta sk "top-down" eller "bottom-up design" eller en kombination av båda. En väsentlig del i en design av ett program är att bestämma vilka datastrukturer som skall använda, så när det talas om kod nedan, så avses kod med vidhängade datastruktur.

Top-down design:

- Skapa en överblick, ett skelett, av hela programmet och bilda alltmer detaljerade beskrivningar av de olika delarna (stegvis förfining, eng. stepwise refinement). I början består vissa delar bara av sk "stubs" ("dummy routines"), väsentligen tomma funktioner, som kanske bara skriver namnet på funktionen och returnerar några standardvärden. Man kan då testa programmet utan att alla delar är färdiga.

  En nackdel är att programmet är begränsat till sin funktion under utvecklingsfasen.

Bottom-up design:

- Börja med detaljerna, och sätt ihop dessa för att skapa större byggstenar. Detta är speciellt lämpligt om man har tillgång till en stor mängd färdig kod, som i MATLAB. En nackdel är att man tidigt kan fatta designbeslut som senare visar sig leda till svårigheter.

Exempel. Här ett enkelt exempel på top-down design. Vi har en uppsättning datafiler som innehåller numeriska data från experiment. Vi vill läsa filerna och beräkna min, max, medelvärde, median och standardavvikelse. Dessa värden skall, tillsammans med numret på experimentet, lagras, i experiment-ordning, på en utdatafil. Datafilerna lagras i en given katalog och filnamnen börjar på **data**. Varje fil innehåller resultatet från ett experiment. Filerna har följande utformning:

```
experiment-nummer (heltal)
kommentar         (sträng)
kommentar         (sträng)
datavärden        (flyttal), ett per rad
```

En del av en top-down design skulle kunna se ut som:

```
tag reda på datafilernas namn
for ( alla datafiler )
  läs en datafil
  beräkna och spara min, max etc.,
    samt experiment-nummer i en matris
end
skriv sparade värden på fil
```

Man skriver ofta med en blandning av naturligt språk och programspråk (eng. "pseudo code").
Vi funderar nu på hur delstegen skall hanteras:

tag reda på datafilernas namn:

```
skaffa en lista med alla filnamn i aktuell katalog
plocka ut filnamn som börjar med 'data'
```

Vi använder en cellvektor för att lagra namnen.

läs en fil:

```
tag reda på namnet på datafilen
öppna filen
läs första raden och spara experiment-nummer
hoppa över två rader
läsa data (lagras i en vektor)
```


Man fortsätter att förfina på detta vis tills man kommer ned på kodnivå. För att kunna börja programmera och testa lämnar man vissa rutiner till senare, t.ex.

```
function params = compute_params(data)
disp('in compute_params')
params = 1:5;
% ------------------------------------------------
function save_params(directory, params)
disp('in save_params')
```

Så här skulle ett färdigt program (utan kommentarer och dokumentation) kunna se ut:

```
function process_data(directory)

[file_list, num_files] = get_file_names(directory);

if num_files == 0
  error('No data files found')
end

% space for exp_number, min, max etc.
params = [];

for file = 1:num_files
  [data, exp_number] = ...
    read_file(directory, file_list{file});
  if length(data) > 0
    params(file, :) = ...
      [exp_number, compute_params(data)];
  else
    disp(['*** skipping file: ', file_list{file}])
  end
end

save_params(directory, params)
```



```
% ------------------------------------------------

function ...
  [file_list, num_files] = get_file_names(directory)

num_files = 0;
file_list = {};

file_info = dir([directory, '/', 'data*']);
for k = 1:length(file_info)
  if ~file_info(k).isdir
    num_files = num_files + 1;
    file_list{num_files} = file_info(k).name;
  end
end

% ------------------------------------------------

function ...
  [data, exp_number] = read_file(directory, file_name)

if directory(end) == '/'
  full_path = [directory, file_name];
else
  full_path = [directory, '/', file_name];
end

fid = fopen(full_path, 'r');
if fid == -1
  warning(['Could not open: ', full_path])
  data = [];
  exp_number = [];
else
  % read one number
  exp_number  = fscanf(fid, '%f', 1);
  if isempty(exp_number)
    warning(['No experiment number in file: ',  ...
```

```
            full_path])
    data = [];
  else
    comment = fgetl(fid);  % skip
    comment = fgetl(fid);  % skip
    % read to EOF
    [data, num_items_read] = fscanf(fid, '%f', inf);

    if num_items_read == 0
      warning(['No data in file: ', full_path])
    end
  end

  if fclose(fid);
    warning(['Cannot close file: ', full_path])
  end

end

% ------------------------------------------------

function params = compute_params(data)
params = [min(data), max(data), mean(data), ...
          median(data), std(data)];

% ------------------------------------------------

function save_params(directory, params)

if directory(end) == '/'
  full_path = [directory, 'params'];
else
  full_path = [directory, '/params'];
end

fid = fopen(full_path, 'w');
```



```
if fid == -1
  error(['Could not open: ', full_path])
end

% sort params
[junk, index] = sort(params(:, 1));
params = params(index, :);

count = fprintf(fid, '%e %e %e %e %e\n', params');
if count == 0
  error(['Could not save params on file: ', full_path])
end

if fclose(fid)
  warning(['Cannot close file: ', full_path])
end
```


22.0.1 Ett bottom-up-exempel

I detta exempel kommer jag att göra en typisk unix-lösning för att göra en tabell över frekvensen av ord i en engelsk text. En viktig idé med unix är att man kan skapa nya kommandon genom att bygga ihop existerande (av vilka finns ett stort antal).
Man skapar typiskt pipelines, en sekvens av kommandon, där nästa kommandos indata är föregående kommandos utdata. T.ex.

```
% wc -l *.m | sort -rn | less
 2678 total
  991 exportfig.m
  236 testa.m
  227 cos_ex_gui.m
  106 rot_trans.m
  100 rot_trans_godt.m
  100 plot_mesh.m
etc.
```

Räkna antalet rader i alla m-filer (i aktuell katalog), sortera numerisk (efter radantal) i omvänd ordning och skicka till en pager (**less**).
Kan kombineras med loopar, if-satser etc. i det skal (eng. shell) man använder. En del kommandon man använder är sk script (oftast skrivna med Bourne-shell-syntax, Stephen Bourne, 1977). På studentmaskinerna används antingen **bash** eller **tcsh**.

```
% file /usr/bin/ps2pdf
/usr/bin/ps2pdf: Bourne shell script text executable

% cat /usr/bin/ps2pdf
#!/bin/sh
# $Id: ps2pdf,v 1.1 2000/03/09 08:40:40 lpd Exp $
# Convert PostScript to PDF.

# Currently, we produce PDF 1.2 by default, but this is r
# not to change in the future.
exec ps2pdf12 "$@"
```


Till skillnad från

```
% file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 2.2.5, dynamically
linked (uses shared libs), stripped
```

Hur många script finns i **/usr/bin**?

```
% cd /usr/bin
% ls | wc -l
3262
```

Här ett litet program i **tcsh**:

```
touch ~/script_files
foreach f (*)
  set a = `file $f`
  if ( $a[2] == Bourne ) then
    echo $f >> ~/script_files
  endif
end
wc -l ~/script_files
```

403 rader i **script_files**.

Här följer ett ordräkningsprogram i denna anda. Jag antar att det är en text på engelska och att bokstäverna a-z, A-Z kommer i följd och utan luckor, vilket är sant för den teckenkodning, UTF-8, som vi använder på studentdatorerna. Det är inte sant om vi lägger till åÅ, äÄ samt öÖ, så man får ändra programmet något för att det skall fungera för en text på svenska.

```
tr -cs A-Za-z '\n' < baskerville | \
tr A-Z a-z | sort | uniq -c | sort -rn > freqs
```

Första **tr** (translate) byter ut allt som inte är en bokstav (**-c** för complement), till **\n**, newline. **-s**, squeeze-repeats, byter ut sekvenser av newline-tecken till ett enda.
\ sist på en rad ger fortsättningsrad.

Så,

```
% tr -c A-Za-z '\n'
  "Interesting, though elementary," said he as he



Interesting

though
elementary


said
he
as
he
```

och

```
% tr -cs A-Za-z '\n'
  "Interesting, though elementary," said he as he
här står en newline
Interesting
though
elementary
said
he
as
he
```

< är en så kallad "redirection". **tr** läser från "stdin" (standard input) och skriver på "stdout" (standard output). **<** styr om indataströmmen från filen **baskerville** till stdin. **|** skickar utdata (ett ord per rad) vidare till nästa **tr**-kommando, som konverterar från versaler till gemener. **\n** markerar fortsättningsrad. **sort** sorterar orden och **uniq -c** tar bort kopior och ger oss antalet kopior


```
% tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq -c
  "Interesting, though elementary," said he as he
      1           en newline
      1 as
      1 elementary
      2 he
      1 interesting
      1 said
      1 though
```

sort -rn sorterar "reverse numeric". **>** skickar stdout ut på filen **freqs**. Här är de tio vanligaste orden:

```
% head -10 freqs
   3328 the
   1628 and
   1592 of
   1498 i
   1446 to
   1306 a
   1144 that
   1009 it
    919 he
    912 in
```

Ordet baskerville är inte fullt lika vanligt:

```
% grep baskerville freqs
    113 baskerville
     12 baskervilles
```

av totalt 59879 ord:

```
% awk '{s += $1} END {print s}' freqs
59879
```

(efter Alfred Aho, Peter Weinberger, och Brian Kernighan) och

```
% wc -l freqs
5586 freqs
```

unika.

Just the other day...

You have $\approx$ 600 files each consisting of $\approx$ 24000 lines (a total of $\approx 14 \cdot 10^6$ lines) essentially built up by:

```
<DOC>
<TEXT>
Many lines of text (containing no DOC or TEXT)
</TEXT>
</DOC>
<DOC>
<TEXT>
Many lines of text
</TEXT>
</DOC>
etc.
```

There is a mismatch between the number of **DOC** and **TEXT**.
Find it!

We can localize the file this way:

```
% foreach f ( * )
foreach? if ( `grep -c "<DOC>" $f` !=   \
              `grep -c "<TEXT>" $f` ) echo $f
foreach? end
```

Not so efficient; we are reading each file twice.
Takes $\approx$ **3.5** minutes.
We used binary search to find the place in the file.

A more fancy program; run the following awk-script for every file:

```
#!awk -f
/^<DOC>/,/^<\/DOC>/ {
                      if ( $1 == "<TEXT>" )
                        found_1++
                      else if ( $1 == "<\/TEXT>" )
                        found_2++
                    }

/^<\/DOC>/    {
                if ( found_1 != 1 || found_2 != 1 )
                  print FILENAME, NR, found_1, found_2
                found_1 = found_2 = 0
              }
```

Takes 10 minutes.