UTGIVEN AV THELIN FÖRLAG

Thelin Förlag, Lidköping
Tel. 0510-66100, *www.thelinforlag.se*
Beställningsnummer J200 4940
Tryckeri: JustNu
ISBN: 978-91-7379-390-2
Foto: Mikael Carlsson

Förord

Hösten 2018 kom programmering in som en del i kurserna Ma1c, Ma2c och Ma3c. Att använda programmering i matematiken kan vara till stor hjälp. Med hjälp av programmering kan vi visualisera sådant som annars ofta upplevs som abstrakt och svårt att få grepp om. Det är också möjligt att göra många beräkningar (och göra dem om och om igen med olika värden) som skulle vara mer eller mindre omöjliga med bara papper och penna.

Samtidigt är det många som har ganska begränsade (eller inga) erfarenheter av programmering när de möter det i matematiken. Syftet med den här boken är främst att du ska få öva dig i att räkna med programmering som hjälp, men vi går också igenom de viktigaste grunderna i programmering, så att du kan göra vilka matematiska beräkningar som helst.

Programmering är också roligt! Vi hoppas att den här boken ska vara till stor hjälp under din mattekurs och att du, med hjälp av att använda programmering i matematiken, kommer att få ett nytt förhållningssätt till räknande. Ett sätt som kan vara både spännande, utmanande och givande.

Vi vill tacka Mikael och Rebecka som har varit till stor hjälp vid framtagandet av denna bok. Vi vill också ge ett stort tack till vår käre förläggare Jan-Eric Thelin på Thelin Förlag.

Krister Trangius och Emil Hall, april 2018.

A
Om denna bok

I detta kapitel kommer vi dels att titta på hur detta läromedel är tänkt att användas, dels kommer en kapitelöversikt där vi går igenom bokens innehåll.

A.1 Att använda detta läromedel

Detta läromedel är tänkt som ett komplement till redan befintliga matteböcker för gymnasiet som saknar programmering. Boken riktar sig främst till kurserna Ma1c, Ma2c och Ma3c men går bra att använda som komplement i övriga mattekurser också.

Boken är indelad i två delar. Del I lär ut de viktigaste grunderna i programmering och Matlab/Octave. Den innehåller också en del övningar. Del II innehåller bara övningar som är riktade till specifika mattekurser. De övningarna är ofta mer komplexa än de i del I.

A.1.1 Till lärare

Skolverkets tanke med programmering i matematiken är att det ska vara ett hjälpsamt verktyg som *stödjer* matematikundervisningen. Det finns många intressanta matematiska problem som lämpar sig väldigt väl att lösa med hjälp av programmering medan andra problem lämpar sig bättre att lösa med ”traditionella” metoder (läs: papper och penna) eller andra digitala verktyg.

En svårighet idag är att många elever inte behärskar programmering när de behöver arbeta med det i matematiken. Därför har vi i denna bok valt att ha en del som ändå ger en introduktion till programmering. Den delen kan med fördel behandlas i eller tillsammans med Programmering 1-kursen. Delen kan också vara till hjälp, även om eleverna redan är bekanta med programmering men tidigare inte stött på just Matlab/Octave. Vi författare har försökt hålla denna introduktion till ett absolut minimum - ett minimum som ändå gör att man kan arbeta med programmering för att lösa meningsfulla uppgifter i matematiken.

A.1.2 Konventioner som används i boken

Låt oss nu titta på vissa konventioner som används i boken och vad de betyder.

Pronomen i boken (vi och du)

Genomgående i denna bok används orden *vi* och *oss* - det som syftas då är vi (författarna) och du läsaren. När det står *du* eller *dig*, åsyftas dig, läsaren.

Nya termer och namn

När nya termer och namn på olika saker dyker upp för första gången är de oftast *kursiverade*. Då det inte är självklart vad som är en ”term” så gäller detta inte alltid. Ibland har vi författare också funnit det lämpligt att kursivera en term mer än bara första gången.

Källkod

Stora delar av boken innehåller källkod (programmeringskod) som exempel. Det anges på följande sätt:

Exempel A.1: Exempel på källkod

```
temperature = input('Ange temperatur: ');
if temperature == 100
    disp('Nu kokar vattnet!');
else
    disp('Vattnet är inte exakt 100 grader...');
end
```

Ett problem med att ange källkod i en bok, är att källkoden inte alltid ryms på bredden. Detta har vi författare försökt lösa genom att formatera koden för att passa boksidan så gott det går. Ibland bryts dock en kodrad upp på två rader, vilket tydliggörs då den andra raden har ett indrag och saknar eget radnummer:

```
1 hist(random_numbers,
     min(random_numbers):max(random_numbers));
```

Källkod i löptext

När källkod skrivs som en del av löptext, kan det se ut på följande sätt: Anropa metoden `disp('något skrivs ut')`.

A.1.3 Övningar

I boken finns det övningar som är indelade i tre olika svårighetsgrader. Svårighetsgrad anges med färg och bokstav (E)nkel, (M)edel och (S)vår:

Övning (E), A.1: Enkel övning

Detta är ett exempel på hur en enkel övning ser ut.

Övning (M), A.2: Medelsvår övning

Detta är ett exempel på hur en medelsvår övning ser ut.

Övning (S), A.3: Svår övning

Detta är ett exempel på hur en svår övning ser ut.

Tekniska detaljer

På olika ställen i boken finns tekniska detaljer i en ruta. Dessa detaljer kan vara intressanta för dem som vill, men är inget man måste kunna eller förstå för att hålla på med programmering i matematiken.

Länkar

Det finns mängder av information på nätet. I denna bok har vi författare därför valt att ha länkar till platser där läsaren kan fördjupa sig i vissa aspekter av det som tas upp. Länkar anges i en sån här ruta.

Och här är ett exempel på en länk: *http://www.trangius.se*

A.2 Del- och kapitelöversikt

Här kommer en kapitelöversikt:

Kapitel B - Matlab och Octave

Vi går igenom Matlab, Octave och Octave Online - olika programvaror som du kan använda för att jobba med denna bok.

Del I - Grunderna i programmering

I den här delen lär vi oss det viktigaste om programmering för att kunna arbeta med det i matematiken.

Kapitel 1 - Datorn som miniräknare

Vi går igenom hur man använder Matlab/Octave som en miniräknare. Detta är en viktig grund innan vi går vidare till grafritande räknare och programmering.

Kapitel 2 - Variabler

Vi går igenom hur variabler funkar i programmering.

Kapitel 3 - Listor

Vi går igenom hur listor funkar i programmering.

Kapitel 4 - Grafer och diagram

Vi går igenom hur man ritar grafer och histogram.

Kapitel 5 - Selektion (med if)

Vi går igenom hur man kan göra val mellan olika alternativ, beroende på sådant som värden på olika tal och variabler.

Kapitel 6 - Iteration (med while)

Vi går igenom hur man kan köra vissa kodstycken om och om igen, utifrån önskade villkor.

Kapitel 7 - Problemlösning

Vi tar två exempel på hur programmerare tänker när de bygger upp ett längre program, från början till slut.

Del II - Övningar och facit

I denna del finns övningar med facit för kurserna Ma1c, Ma2c och Ma3c. Många av övningarna lämpar sig för flera av de olika mattekurserna. Här kommer en lista över vilka underrubriker i *kapitel 8 - Övningar* som är relaterade till det centrala innehållet i de olika kursplanerna:

Ma1c

- [...]begreppen primtal och delbarhet.
  - ▷ 8.1 - Primtal, delbarhet och faktorisering
- Algebraiska och grafiska metoder för att lösa linjära ekvationer och olikheter samt potensekvationer[...]
  - ▷ 8.3 - Numerisk lösning av linjära ekvationer
- [...M]etoder för beräkning av sannolikheter vid slumpförsök i flera steg med exempel från spel[...].
  - ▷ 8.2 - Sannolikhet och statistik
- Begreppen förändringsfaktor och index. Metoder för beräkning av räntor och amorteringar för olika typer av lån [...]
  - ▷ 8.5 - Potensfunktioner och exponentialfunktioner
- egenskaper hos [...] exponentialfunktioner.
  - ▷ 8.5 - Potensfunktioner och exponentialfunktioner

Ma2c

- Statistiska metoder för rapportering av observationer och mätdata från undersökningar.
  - ▷ 8.2 - Sannolikhet och statistik
- Metoder för beräkning av olika lägesmått och spridningsmått inklusive standardavvikelse, med digitala verktyg.
  - ▷ 8.2 - Sannolikhet och statistik
- Konstruktion av grafer till funktioner [...] med digitala verktyg.
  - ▷ 8.2 - Sannolikhet och statistik
  - ▷ 8.5 - Potensfunktioner och exponentialfunktioner
  - ▷ 8.4 - Andragradsekvationer

- Egenskaper hos andragradsekvationer
  - ▷ 8.3 - Numerisk lösning av linjära ekvationer [som förkunskapskrav till andragradsekvationer].
  - ▷ 8.4 - Andragradsekvationer
- Algebraiska och grafiska metoder för att lösa exponential-, andragrads- och rotekvationer [...]
  - ▷ 8.4 - Andragradsekvationer
  - ▷ 8.5 - Potensfunktioner och exponentialfunktioner

Ma3c

- Algebraiska och grafiska metoder för lösning av extremvärdesproblem.
  - ▷ 8.4 - Andragradsekvationer
- Orientering när det gäller kontinuerlig och diskret funktion samt begreppet gränsvärde.
  - ▷ 8.6 - Derivata
- Begreppen [...] ändringskvot och derivata för en funktion.
  - ▷ 8.6 - Derivata
- Algebraiska och grafiska metoder för bestämning av derivatans värde för en funktion.
  - ▷ 8.6 - Derivata
- Introduktion av talet e och dess egenskaper.
  - ▷ 8.3 - Numerisk lösning av linjära ekvationer [som förkunskapskrav till talet e].
  - ▷ 8.6 - Derivata

B

Matlab och Octave

I det här kapitlet ska vi översiktligt gå igenom verktygen Matlab och Octave och hur man använder dem. Vilket av dessa verktyg du använder dig av ska inte spela någon roll i den här boken, då samtliga kodexempel är skrivna och testade med båda.

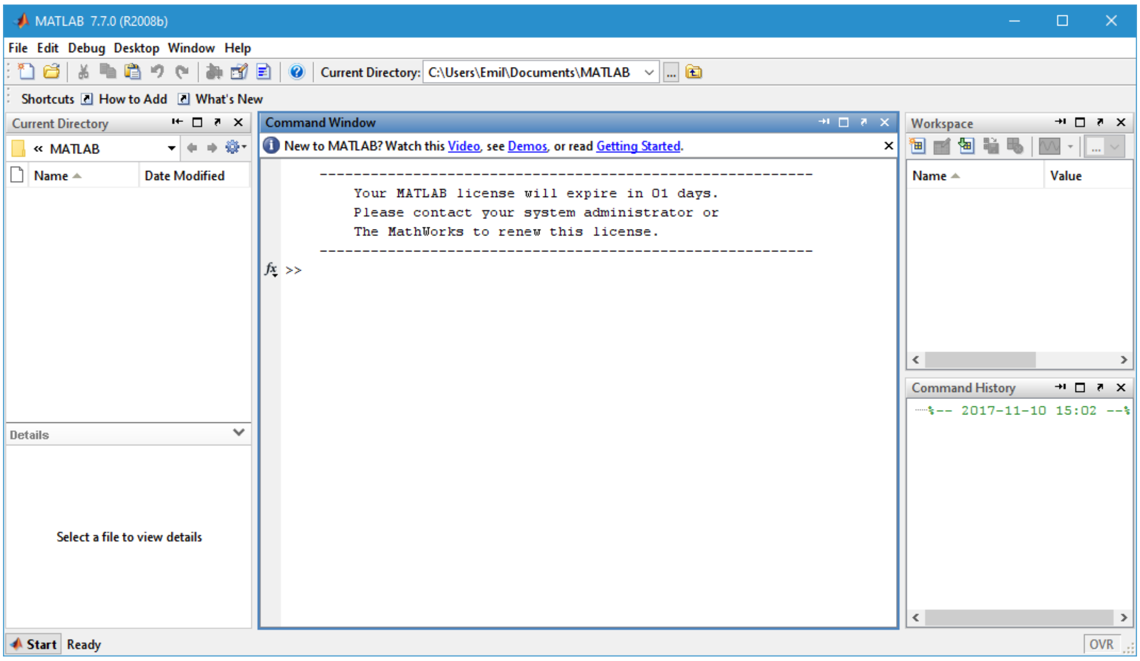
B.1 Matlab: introduktion

För att använda Matlab krävs det en licens som kostar pengar. Om du ska använda Matlab, så har förhoppningsvis din skola en licens eller så har du köpt en själv. Om du inte har det, så kan vi rekommendera Octave istället som inte kostar pengar (se delkapitel B.2). Matlab funkar i Microsoft Windows, Mac OS X och Linux.

B.1.1 Matlab: kommandofönstret

När du först startar Matlab så ser det ut ungefär såhär:

Figur B.1: Start-utseendet på Matlab



Som du ser finns det ett antal olika rutor. Den som kommer vara mest intressant för oss i början av boken är rutan ”Command window”. I den finns två högerpilar:

Exempel B.1: Tom kommando-rad

```
>>
```

Efter högerpilarna finns din blinkande markör, så du kan skriva text där. Testa att skriva in texten `1+1`, så att det ser ut såhär:

Exempel B.2: Skrivit in lite matte

```
>> 1+1
```

och tryck sedan på Enter-tangenten. Vad tror du kommer hända?

Exempel B.3: Hurra, datorn kan räkna!

```
>> 1+1
ans = 2
>>
```

Ordet `ans` är en förkortning av ”answer”.

Notera att `1+1` även hamnar i rutan nere till höger som heter ”Command history”. Det är precis som det låter en lista med alla uträkningar du skrivit in tidigare, sorterade i den ordning du skrev in dem. Prova själv att skriva in flera enkla matteberäkningar i ”Command window” och se hur de dyker upp i ”Command history”. Om du sedan dubbelklickar på en rad i ”Command history” så körs denna beräkning igen. Det kanske inte är så viktigt än så länge, men kommer att bli mer användbart längre fram när du vill slippa skriva in en jättelång uträkning två gånger.

B.1.2 Filer i Matlab

I början av boken kommer vi bara behöva använda ”Command window” men i kapitel 5 behöver vi börja arbeta med filer, för att kunna skriva längre kodstycken.

För att skapa en ny fil, tryck i menyn: *File* -> *New* -> *Blank M-File*.

Nu har du en tom fil där du kan skriva in samma typ av kommandon som vi tidigare har skrivit in i ”Command window”. Skillnaden är att här körs inte koden direkt efter att du skrivit en rad och tryckt Enter, utan du kan skriva en massa rader och sen köra alltihop på en gång. Bara för att testa detta, skriv in:

Exempel B.4: Skrivit in lite matte

```
1+1
```

Hitta sedan rätt knapp överst i Editor-rutan. Antingen en grafisk knapp med en grön *Play*-symbol som pekar åt höger, eller i menyn *Debug* -> *Save file and run*, eller genom att trycka på *F5*-tangenten.

B.2 GNU Octave: introduktion

Om din du eller din skola inte har en licens för Matlab så kan du använda en gratis opensource-klon som heter GNU Octave och funkar ungefär likadant. Octave funkar i Microsoft Windows, Mac OS X, Linux, BSD och en del andra system.

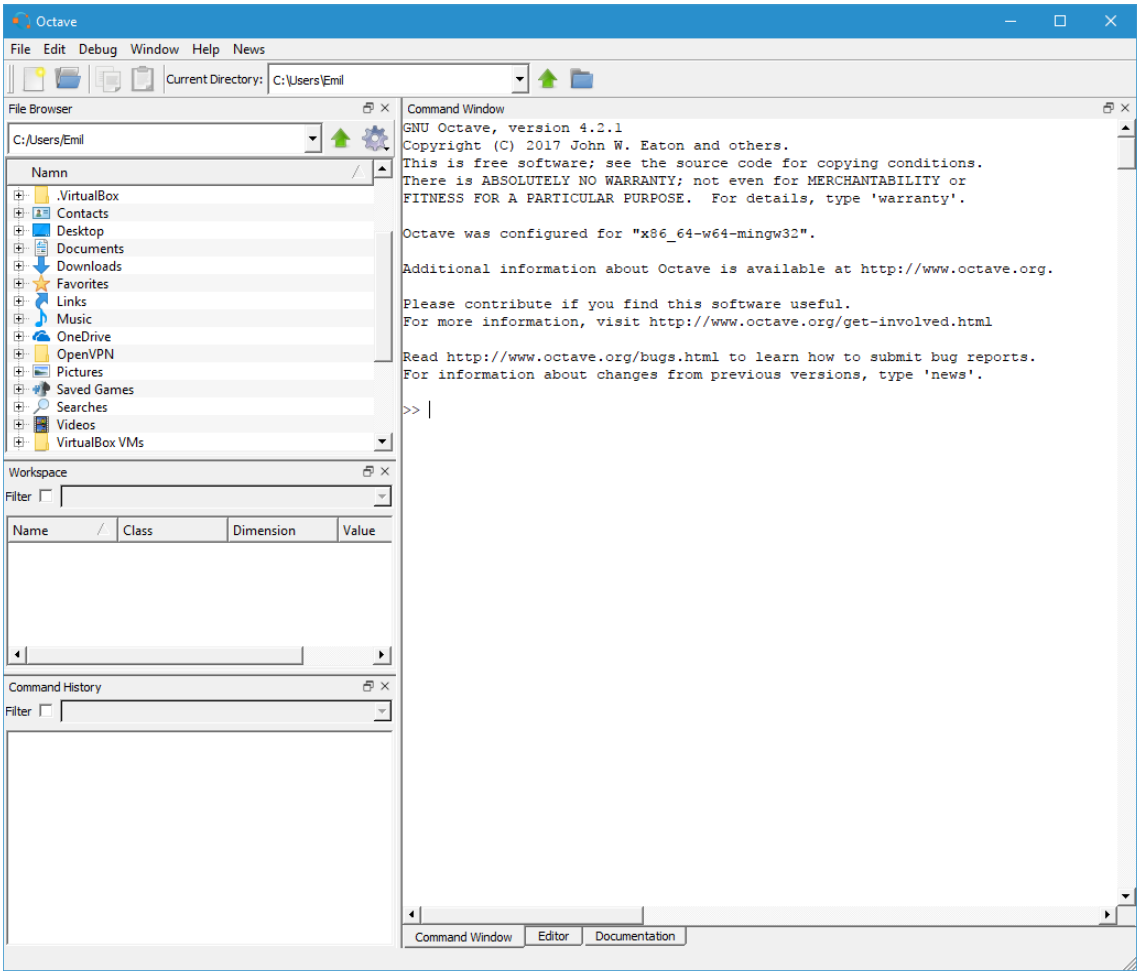
För att kunna programmera i Octave så måste det först finnas nerladdat och installerat på din dator.

> Octave finns att ladda ner på: *https://www.gnu.org/software/octave/*

B.2.1 GNU Octave: kommandofönstret

När du först startar Octave så ser det ut ungefär såhär:

Figur B.2: Start-utseendet på GNU Octave



Som du ser finns det ett antal olika rutor. Det som kommer vara mest intressant för oss i början av boken är rutan ”Command window”. I ”Command window” finns två högerpilar.

Exempel B.5: Tom kommando-rad

```
>>
```

Efter högerpilarna finns din blinkande markör, så du kan skriva text där. Skriv in texten `1+1`, så att det ser ut såhär:

Exempel B.6: Skrivit in lite matte

```
>> 1+1
```

och tryck sedan på Enter-tangenten. Vad tror du kommer hända?

Exempel B.7: Hurra, datorn kan räkna!

```
>> 1+1
ans = 2
```

`ans` är en förkortning av ”answer”.

Notera att `1+1` även hamnar i rutan nere till vänster som heter ”Command history”. Det är precis som det låter en lista med alla uträkningar du skrivit in tidigare, sorterade i den ordning du skrev in dem. Prova själv att skriva in flera enkla matteberäkningar i ”Command window” och se hur de dyker upp i ”Command history”. Om du sedan dubbelklickar på en rad i ”Command history” så körs denna beräkning igen. Det kanske inte är så viktigt än så länge, men kommer att bli mer användbart längre fram när du vill slippa skriva in en jättelång uträkning två gånger.

B.2.2 Filer i Octave

I början av boken kommer vi bara behöva använda ”Command window” men i kapitel 6 behöver vi börja arbeta med filer, för att kunna skriva längre kodstycken.

Längst ner, bredvid ”Command window” så finns fliken ”Editor” - klicka på den! I editor-rutan klickar du sedan på File -> New Script.

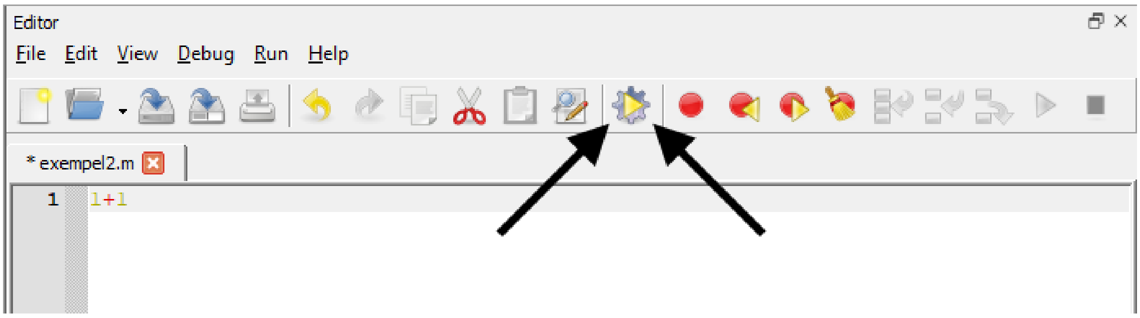
Nu har du en tom fil där du kan skriva in samma typ av kommandon som vi tidigare har skrivit in i ”Command window”. Skillnaden är att här körs inte koden direkt efter att du skrivit en rad och tryckt Enter, utan du kan skriva en massa rader och sen köra alltihop på en gång. Bara för att testa detta, skriv in:

Exempel B.8: Skrivit in lite matte

```
1+1
```

Hitta sedan rätt knapp överst i Editor-rutan. Antingen en grafisk knapp med ett kugghjul och en *Play*-symbol som pekar åt höger, eller i menyn *Run -> Save file and run*, eller genom att trycka på *F5*-tangenten.

Figur B.3: Knapp för att köra en fil med kod



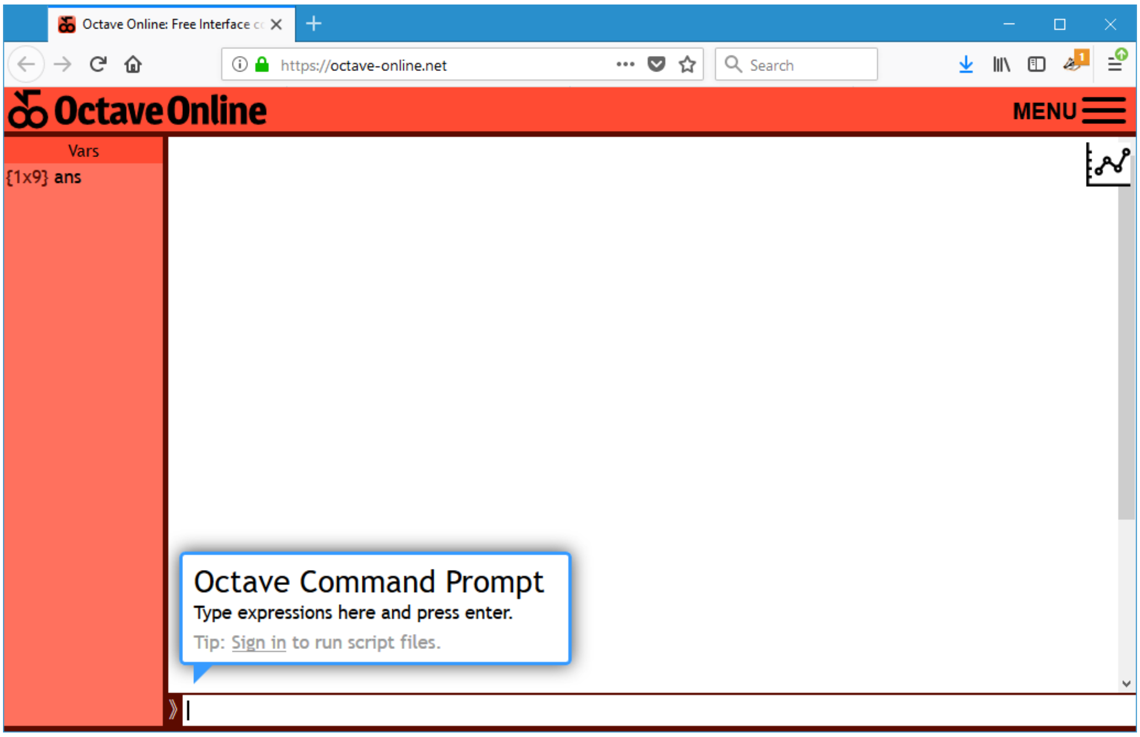
B.3 Octave Online: introduktion

Om du inte vill/kan ladda ner och installera program så kan du använda en gratisversion i webbläsaren istället.

> Octave Online finns på: *https://octave-online.net/*

När du först går in på Octave Online så ser det ut ungefär såhär:

Figur B.4: Start-utseendet på Octave Online



Som du ser finns det ett antal olika rutor. De som kommer vara mest intressant för oss i början av boken är de vita rutorna. I det smala vita fältet längst ner (som kallas ”Command Prompt”) finns två högerpilar. Klicka i det fältet!

Efter högerpilarna finns din blinkande markör, så du kan skriva text där. Skriv in texten `1+1` och tryck sedan på Enter-tangenten. Vad tror du kommer hända?

Exempel B.9: Hurra, datorn kan räkna!

```
ans = 2
```

`ans` är en förkortning av ”answer”.

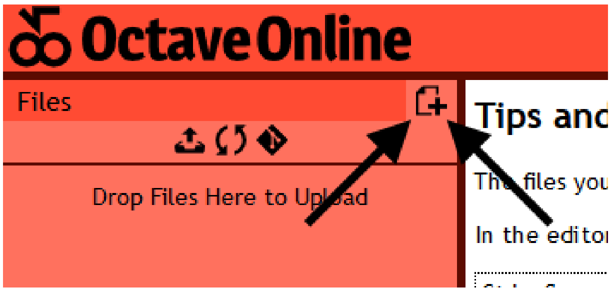
B.3.1 Filer i Octave Online

I början av boken kommer vi bara behöva använda ”Command window” men i kapitel 6 behöver vi börja arbeta med filer, för att kunna skriva längre kodstycken.

För att kunna skapa filer i Octave Online så behöver du ett konto. Det går att skapa ett nytt konto på sidan. Det är också möjligt att logga in med sitt Google-konto.

För att skapa en ny fil, tryck på ”create empty file” (en ikon) uppe till vänster:

Figur B.5: Knapp för att skapa en ny fil



Välj vad filen ska heta. När du har klickat på OK dyker filen upp i listan till vänster. Klicka på den nya filen.

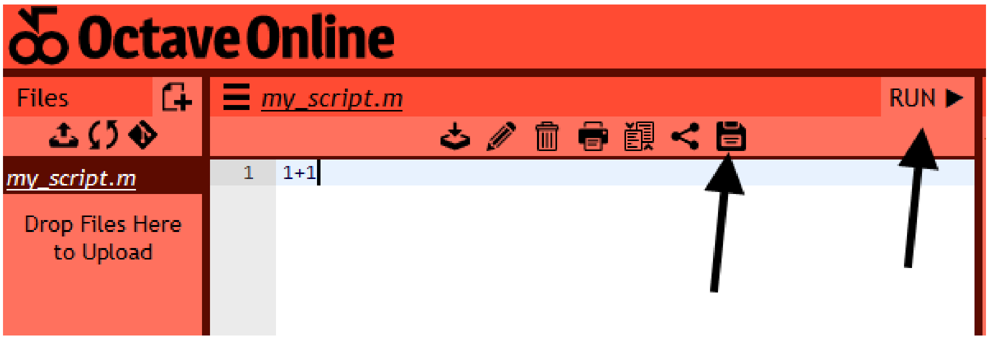
Nu har du en tom fil där du kan skriva in samma typ av kommandon som vi tidigare har skrivit in i ”Command Prompt”. Skillnaden är att här körs inte koden direkt efter att du skrivit en rad och tryckt Enter, utan du kan skriva en massa rader och sen köra alltihop på en gång. Bara för att testa detta, skriv in:

Exempel B.10: Skrivit in lite matte

```
1+1
```

För att kunna köra ditt program, tryck först på spara-knappen, därefter på *run*.

Figur B.6: Spara filen och kör



I

Grunderna i programmering

1

Datorn som miniräknare

I det här kapitlet kommer vi att lära oss att använda Matlab/Octave som en miniräknare. Detta är en viktig grund innan vi går vidare till att använda Matlab/Octave för att rita grafer och programmera.

1.1 Kommentarer

När vi skriver in tal och matteberäkningar i Matlab/Octave så utför datorn dessa beräkningar. Men ibland kan det vara användbart för vår egen skull att skriva in text som datorn *inte* bryr sig om. Som minnesanteckningar till oss själva. Om vi skriver ett procenttecken `%` så kommer datorn att strunta i resten av raden, vad det än står där. Detta kallas inom programmering för *kommentarer*:

Exempel 1.1: Vår första kommentar

```
>> 1+1 % allt efter procenttecknet är bara en kommentar
ans = 2
```

Våra kodexempel i boken kommer hädanefter ofta att innehålla små kommentarer som förklarar detaljer i koden, när vi inte skriver förklaringarna i brödtexten ovanför eller nedanför kodexemplet.

1.2 Aritmetik: de fyra räknesätten

I föregående kapitel så testade vi kommandofönstret i Matlab/Octave genom att skriva `1+1`. Låt oss testa de fyra räknesätten. Vi börjar med bara heltal (så tar vi decimaltal senare):

Exempel 1.2: De fyra räknesätten

```
>> 1-1
ans = 0
>> 1+5-3
ans = 3
>> 122-300
ans = -178
>> 3*5
ans = 15
>> 20+2*7
ans = 34
>> (20+2)*7
ans = 154
>> 21/7
ans = 3
>> 10+6/2
ans = 13
>> (10+6)/2
ans = 8
```

Som du kanske ser ovan, så gäller de vanliga reglerna för de fyra räknesätten även i Matlab/Octave; Multiplikation och division sker före addition och subtraktion och det är, i vanlig ordning, möjligt att styra detta med parenteser. Som du också ser är multiplikationstecknet en asterisk `*`.

1.2.1 Decimaltal

Vad händer om vi gör en division som inte går jämnt upp?

Exempel 1.3: Decimaltal

```
>> 17 / 2
ans = 8.500
```

Jo, vi får ett decimaltal med en punkt mellan heltalsdelen och decimalerna. Svensk standard är att skriva kommatecken där, men Matlab/Octave följer engelsk standard där man använder punkt istället. Om vi själva vill skriva in ett decimaltal så bör vi också använda punkt:

Exempel 1.4: Decimaltal skrivs med punkt

```
>> 8.5 * 2
ans = 17
```

Om vi skriver in kommatecken så kan det eventuellt också fungera i vissa situationer, men eftersom kommatecken även har andra betydelser inom programmering så finns det stor risk att Matlab/Octave missförstår och ger oss ett helt annat resultat än vi ville:

Exempel 1.5: Varning för kommatecken

```
>> 8,5 * 2
ans = 8
ans = 10
```

Övning (E), 1.1: De fyra räknesätten

Testa de fyra räknesätten i Matlab/Octave med olika siffror, både heltal och decimaltal.

1.3 Operatorer

Inom programmering talar man om något som kallas *operatorer*. Vi har redan använt några operatorer, nämligen de fyra räknesätten (`+`, `-`, `*`, `/`) men det finns fler som du redan känner till från matematiken (och några som du kanske inte känner igen).

I den här boken kommer vi att arbeta med lite olika operatorer och du kommer att lära dig dem allt eftersom. Några operatorer som vi kan titta på redan nu, är de så kallade *jämförelseoperatorerna*:

En jämförelseoperator används för att jämföra två tal. Vi kan se det som att vi frågar datorn om jämförelsen stämmer (t.ex. ifall ett tal är mindre än ett annat) och datorn svarar ja eller nej. Låt oss testa:

Exempel 1.6: Mindre än-operatorn

```
>> 3 < 17 % är 3 mindre än 17?
ans = 1
```

Det stämmer ju att 3 är mindre än 17. Som du ser, så svarar datorn 1. Det är datorns sätt att säga ”ja”. Om det inte hade stämt, hade datorn svarat 0:

Exempel 1.7: Mindre än-operatorn igen

```
>> 17 < 3 % är 17 mindre än 3?
ans = 0
```


Här kan du se de jämförelseoperatorer som finns i Matlab/Octave:

Tabell 1.1: Jämförelseoperatorer

Tecken	*Betydelse*
==	lika med
<	mindre än
>	mer än
<=	mindre än eller lika med
>=	mer än eller lika med
~=	inte lika med

Notera att jämförelseoperatorn == består av två lika med-tecken på rad. Det kanske verkar märkligt, men blir begripligt senare, i delkapitel 2.2 där vi talar om tilldelningsoperatorn som skrivs med endast ett lika med-tecken.

Låt oss testa några av dessa:

Exempel 1.8: Test av jämförelseoperatorer

```
>> 3 > 17 % är 3 mer än 17?
ans = 0
>> 5 >= 5 % är 5 mer än eller lika med 5?
ans = 1
>> 8 == 8 % är 8 lika med 8?
ans = 1
```

Du kanske undrar vad det här ska vara bra för. Är det inte självklart att 3 är mindre än 17? Varför ska vi fråga datorn om det? Jämförelseoperatorer används oftast tillsammans med selektion (som vi går igenom i kapitel 5) och iteration (som vi går igenom i kapitel 6).

Det finns också fler operatorer i Matlab/Octave än de vi går igenom i den här boken. Se här för en lista: *https://se.mathworks.com/help/matlab/operators-and-elementary-operations.html*

Övning (E), 1.2: Jämförelseoperatorer

Testa själv med samtliga jämförelseoperatorer och lite olika tal till höger och vänster!

1.4 Kvadratrot

Vi antar att du redan känner till begreppet ”roten ur”, eller *kvadratrot* som det också kallas. Med papper och penna brukar vi skriva till exempel $\sqrt{9} = 3$. Men det finns ingen tangent på datorns tangentbord för att skriva ett sådant rot-tecken. I Matlab/Octave räknar vi istället ut kvadratrötter med ordet `sqrt` - förkortning av engelskans *square root*. Sen direkt efter ordet `sqrt` ska vårt tal stå inom parentes:

Exempel 1.9: Kvadratrot, roten ur 9

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

1.5 Funktioner i programmering

Begreppet *funktion* är kanske något du känner igen från din vanliga mattebok? T.ex. kanske du har hört att y är en funktion av x, alltså: $y = f(x)$. Här händer något med x inne i funktionen f och y har värdet av $f(x)$. Inom programmering används det begreppet lite annorlunda - varje funktion har ett visst namn. Vi kan se det som att namnet är en förkortning som representerar ett längre stycke kod.

Det finns en massa inbyggda funktioner i Matlab/Octave. Vi har redan lärt oss en av dem, nämligen `sqrt`, och vi ska strax lära oss några till. Det går även att skapa egna funktioner, men det kommer vi inte lära oss i denna bok. Att skapa egna funktioner gör programmerare nämligen mest för att strukturera stora program, och vi kommer inte att skapa så stora program.

Om du inte riktigt förstår allt detta så är det ingen fara - du kommer kunna använda funktioner ändå.

För att använda en funktion skriver vi alltid dess namn, sedan en inledande parentes, sedan så kallade *argument*, och sist en avslutande parentes. Till exempel:

Exempel 1.10: Repetition av funktionsanvändning

```
>> sqrt(81)
ans = 9
```


Vissa funktioner har fler än ett argument. Här är exempel på funktioner som tar två argument. Argumenten står inom paranteserna och separeras med kommatecken:

Exempel 1.11: Funktioner med två argument

```
>> min(3.5, 17) % ger det minsta av två tal
ans = 3.5000
>> max(3.5, 17) % ger det största av två tal
ans = 17
>> mod(26, 10) % ger rest efter heltalsdivision
ans = 6
```

Nu förstår du kanske varför det inte går så bra att använda kommatecken för decimaltal?

Notera att ett argument i sin tur kan vara ett resultat av en uträkning, t.ex:

Exempel 1.12: Resultat av uträkning som argument

```
>> min(sqrt(81), 8+2)
ans = 9
```

Och det går förstås att göra hur långa kedjor som helst:

Exempel 1.13: Människor från yttre rymden

```
>> min(sqrt(sqrt(81)*3*3), sqrt((8+2)^2))
ans = 9
```

Övning (E), 1.3: Testa funktioner

Testa att använda alla de ovanstående funktionerna med några olika argument. Testa även att använda funktioner som argument till andra funktioner.

1.6 Mer matte

Det finns förstås många fler funktioner inbyggda i Matlab/Octave. Vi kommer nu att lista andra operatorer och funktioner som brukar användas i mattekurserna Ma1c, Ma2c och Ma3c. Vi kommer också titta på ett par konstanter.

1.6.1 Potenser

I den rena matematiken brukar vi skriva potenser (”upphöjt till”) med små siffror, till exempel 3^2. Åter igen finns det inget sätt på datorns tangentbord att skriva sådana små siffror. Istället använder vi tecknet som ser ut som en uppåtpil, ett spetsigt hustak (se nästa sida):

Exempel 1.14: Potenser

```
>> 3^2
ans = 9
```

1.6.2 Trigonometri

Det speciella talet pi finns inbyggt i Matlab/Octave. Det finns inget specialtecken π utan vi skriver helt enkelt:

Exempel 1.15: pi

```
>> pi
ans = 3.1416
```

De trigonometriska funktionerna sinus, cosinus, tangens och deras släktingar, finns också inbyggda. Låt oss testa sinus-funktionen:

Exempel 1.16: Trigonometri

```
>> sind(45)
ans = 0.70711
```

I följande tabell kan bokstaven `v` mellan parenteserna ersättas med valfritt tal:

Tabell 1.2: Trigonometriska funktioner

Matteformel	*Funktion i Matlab/Octave*
$\sin v$	`sind(v)`
$\cos v$	`cosd(v)`
$\tan v$	`tand(v)`
$\sin^{-1} v$	`asind(v)`
$\cos^{-1} v$	`acosd(v)`
$\tan^{-1} v$	`atand(v)`

Bokstaven `d` i slutet av funktionernas namn står för *degrees* på engelska, alltså grader på svenska. I den här boken räknar vi bara med grader. Det finns också ett annat sätt att räkna vinklar inom trigonometrin, nämligen *radianer*. Om du skriver `sin` istället för `sind` så blir det radianer istället.

Övning (E), 1.4: Trigonometriska funktioner

Testa själv med samtliga trigonometriska funktioner och lite olika tal mellan parenteserna!

1.6.3 Logaritmer

Exempel 1.17: Logaritmer

```
>> e
ans = 2.7183
>> log10(10^3)
ans = 3
>> log(e^3)
ans = 3
```

1.6.4 Avrundning

Som vanligt ska vårt tal stå inom parentes.

Exempel 1.18: Avrundning

```
>> round(3.6) % avrundar till närmaste heltal
ans = 4
>> floor(3.6) % avrundar nedåt
ans = 3
>> floor(-3.6) % nedåt även för negativa tal. ej mot noll
ans = -4
>> ceil(3.2) % avrundar uppåt
ans = 4
```

1.6.5 Slumptal

Det är möjligt att slumpa fram tal i Matlab/Octave. Vi kan be om att få ett slumptal från och med 1 till och med ett valfritt tal, t.ex. 6 om vi vill simulera ett tärningsslag:

Exempel 1.19: Slumptal

```
>> randi(6)
ans = 2
>> randi(6)
ans = 1
>> randi(6)
ans = 6
```

Om du vill ha ett slumptal inom ett intervall som *inte* börjar på 1, så behöver du arbeta med listor, vilket vi går igenom i delkapitel 3.8

1.6.6 Förvandla negativa tal till positiva (abs)

En annan funktion som kan vara användbar är `abs`, som förvandlar negativa tal till positiva:

Exempel 1.20: abs

```
>> abs(-26) % förvandlar negativa tal till positiva
ans = 26
```

1.6.7 Mer då?

I det här kapitlet gick vi igenom sånt som gör Matlab/Octave till en vanlig miniräknare. Det finns såklart sådant som gör att vi kan använda datorn som en grafräknare också. Det kommer vi att gå igenom i kapitel 4 men först ska vi lära oss några viktiga grunder i programmering.

2

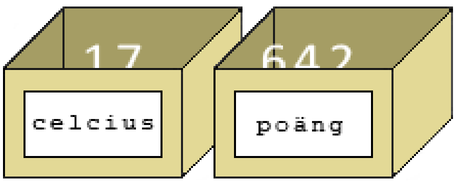
VARIABLER

I det här kapitlet ska vi gå igenom variabler och hur de funkar i programmering. Vi kommer också lära oss hur man själv kan styra utskrift på ett tydligare sätt.

Du känner kanske redan till begreppet variabler? Traditionellt inom matematiken talar man ofta om okända variabler som x och y eller a och b. Variabler i Matlab/Octave är något liknande, men de används lite annorlunda.

En variabel i programmering ses kanske enklast som en låda, med en etikett på. På etiketten står det ett namn och i lådan ligger det ett tal.

Figur 2.1: Variabler kan ses som lådor med etikett och innehåll



I matematikböcker är variablers namn oftast bara en bokstav långa, men i programmering brukar namnet vara ett helt ord, vilket hjälper oss att hålla reda på dem då vi har många.

En annan skillnad är att variabler i programmering alltid har ett visst värde. I traditionell matematik kan man t.ex. säga att $a^x * a^y = a^{x+y}$ som en generell regel som gäller för alla värden. I Matlab/Octave så arbetar man inte med generella värden på variablerna, utan de "innehåller" alltid ett visst tal.

2.1 Att skapa en variabel

För att skapa en variabel hittar man först på ett namn till den, sen skriver man namnet, likamed-tecken, och värdet. Till exempel:

Exempel 2.1: Skapa variabeln celsius

```
>> celsius=17
```

Det är även okej att ha mellanslag om man tycker att det blir mer lättläst när det är mindre tätt:

Exempel 2.2: Skapa variabeln celsius

```
>> celsius = 17
```

Namnet kan inte innehålla åäö, mellanslag eller andra konstiga specialtecken. Om vi vill att namnet ska bestå av flera ord så kan vi som sagt inte använda mellanslag, utan istället brukar vi separera orden med hjälp av understreck; `min_egen_variabel`.

2.2 Tilldelningsoperatorn =

Lika med-tecknet = som vi använde ovan kallas för *tilldelningsoperatorn*. I delkapitel 1.3 tittade vi på jämförelseoperatorn ==. Inom programmering skiljer man på *jämförelse* och *tilldelning*. Tilldelningsoperatorn skrivs med bara *ett* lika-med tecken och används, som vi såg ovan, när man vill ge en variabel ett värde. Det är mycket viktigt att man inte blandar ihop tilldelningsoperatorn = och jämförelseoperatorn ==.

Traditionellt inom matematiken har du kanske lärt dig att det inte spelar någon roll på vilken sida ”lika med”-tecknet olika tal står. När man programmerar i Matlab/Octave är det dock annorlunda. Variabeln, som ska få ett värde, står till vänster om tilldelningsoperatorn. Värdet som variabeln ska få, står till höger. Detta är alltså inte okej, men testa gärna själv och se vad som händer:

Exempel 2.3: Man får inte sätta variabelnamnet på fel sida

```
>> 17 = celsius % ej ok!
```

Det som ligger på högersidan om tilldelningsoperatorn händer först. Det innebär att vi kan göra beräkningar på högersidan. När det väl har räknats ut, tilldelas variabeln värdet. Vi kan t.ex. plussa ihop en massa siffror och sedan lägga det i variabeln. Här får variabeln `nr` värdet 110:

Exempel 2.4: Beräkningar görs på högersidan om tilldelningsoperatorn

```
>> nr = 100 + 3 + 7
```

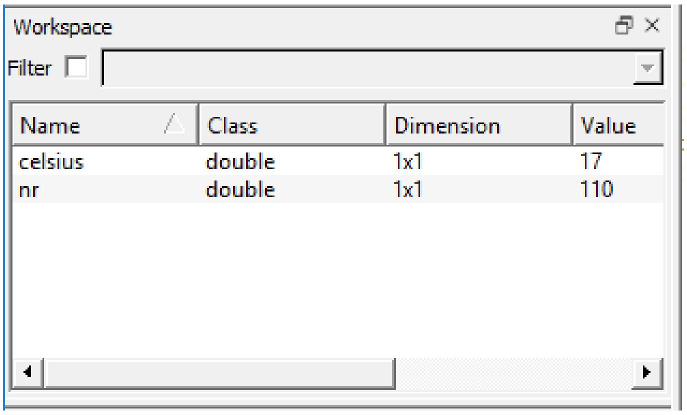
Med andra ord sker det i följande två steg:

1. Talen 100, 3 och 7 summeras till 110.
2. Variabeln `nr` skapas och tilldelas värdet 110.

2.3 Rutan ”Workspace” i Matlab/Octave

Dags att prata om en till av rutorna i Matlab/Octave. Om du skrivit in ovanstående kodexempel i ”Command window” så har du skapat de två variablerna `celsius` och `nr`. Dessa går nu att se i den lilla rutan ”Workspace”.

Figur 2.2: Rutan ”Workspace” efter att vi skapat två variabler



Än så länge behöver du bara bry dig om kolumnen ”Name” och kolumnen ”Value”. Octave visar även några andra kolumner som du inte behöver tänka på än.

Det finns en motsvarande ruta i Octave Online som heter ”Vars”. Där kan man klicka på variablernas namn för att se deras värde.

För att rensa alla variabler som du har skapat kan du skriva kommandot `clear`. Då ser du också att rutan workspace blir tom.

2.4 Att läsa en variabels innehåll

Efter att en variabel har skapats så kan man läsa innehållet i den och göra beräkningar med den. Man kan till exempel addera två variabler med varandra. Här tilldelar vi variabeln `summa` det sammanlagda värdet av `nr1` och `nr2`, alltså 655 (se kod på nästa sida):

Exempel 2.5: Addera två variabler

```
1 >> nr1 = 100
2 nr1 =  100
3 >> nr2 = 555
4 nr2 =  555
5 >> total = nr1 + nr2
6 total =  655
```

Men, kanske du undrar nu, sa vi inte nyss att man inte får sätta variabelnamn på högersidan? Nja, det får man visst, om de variablerna redan finns sedan tidigare och därmed faktiskt har ett värde (100 och 555 i exemplet ovan). Vad vi egentligen menade var att den variabel som vi vill tilldela ett värde måste stå på vänstersidan.

Viktigt att notera är att koden körs uppifrån och ned. Så först skapar vi två variabler (på rad 1 och rad 3). Därefter adderar vi dem (på rad 5). Det hade inte gått att göra tvärtom, att addera två variabler innan de skapats.

Efter att man har skapat och använt variabeln, kan man fortsätta att använda den. Man kan t.ex. ändra variabelns värde. I följande kodstycke ändrar vi en variabel från att ha värdet 100 till att ha värdet 555:

Exempel 2.6: Ändra värdet på en variabel

```
>> nr = 100
nr =  100
>> nr = 555
nr =  555
```

> Faktum är att ordet *variabel* kommer från latinets *variare*, vilket betyder ”ändra”. Jämför svenskans variera!

Man kan öka en variabels värde. Det gör man genom att lägga på variabelns (tidigare) värde till sig själv och addera ett nytt värde. Detta kanske är lite förvirrande om du fortfarande tänker på tilldelningsoperatorn som ett lika med-tecken. Men det finns inget som hindrar att vi använder en variabel i en beräkning på högersidan, och skriver samma variabel på vänstersidan. På första raden i följande kodstycke får variabeln `nr` värdet 100, på andra raden får den värdet 150:

Exempel 2.7: Öka värdet på variabeln

```
1 >> nr = 100
2 nr =  100
3 >> nr = nr + 50
4 nr =  150
```


Om detta vore en matematisk ekvation så vore det förstås omöjligt. Det finns ju inget tal som är lika med sig självt plus 50.

2.5 Styra utskrifterna

Hittills har vi låtit datorn automatiskt skriva ut resultaten av alla våra beräkningar. Men vi måste inte ha det så alltid.

2.5.1 Semikolon

Om vi skriver ett semikolon (`;`) på slutet av raden (innan eventuell kommentar) så får vi inte någon utskrift av resultatet:

Exempel 2.8: Hejda utskrift med semikolon

```
>> nr1 = 100; % skrivs inte ut...
>> nr2 = 555; % skrivs inte ut...
>> summa = nr1 + nr2 % men denna skrivs ut!
summa =  655
```

Detta kan vara användbart när vi skriver längre program framöver och inte vill bli distraherade av en massa onödiga utskrifter.

2.5.2 Skriva ut med disp

Ordet `disp` är en förkortning av engelskans *display*. Med `disp` kan vi be datorn skriva ut saker:

Exempel 2.9: Skriv ut på kommando

```
>> nr1 = 100;
>> nr2 = 555;
% vi kan skriva ut text med apostrofer,
% så kallade enkelfnuttar:
>> disp('nu ska vi räkna');
nu ska vi räkna
>> disp(nr1); % vi kan skriva ut en variabels värde,
100
>> disp(nr1+nr2+12); % ...och resultatet av en uträkning
667
```


2.5.3 Övningar för variabler

Övning (E), 2.1: Höger eller vänster?

Om du kör dessa tre rader kod:

```
a = 1;
b = 2;
a = b
```

Vad är nu värdet på `a` och `b`? Fundera själv innan du testar och läser facit.

Övning (E), 2.2: Kopplas variabler ihop för all framtid?

Om du kör dessa tre rader kod:

```
a = 1;
b = a;
a = 2
```

Vad är nu värdet på `b`? Fundera själv innan du testar och läser facit.

Övning (E), 2.3: Summan och medelvärdet av tre tal

Låt säga att du redan har dessa tre variabler inmatade i Matlab/Octave:

```
a = 23;
b = 45;
c = 67;
```

Skriv en rad kod som beräknar summan av dessa variabler och skriver ut summan. Skriv därefter en till rad kod som skriver ut medelvärdet. Kom ihåg att det är datorn som ska göra uträkningen, inte du!

Övning (E), 2.4: Decimaltal till heltal

Låt säga att du redan har denna variabel inmatad i Matlab/Octave:

```
a = 11.534;
```

Skriv en rad kod som tar denna variabel, omvandlar decimaltalet till närmsta heltal, och skriver ut det.

3 Listor

I det här kapitlet ska vi gå igenom listor och se hur de funkar i programmering. Man kan se listor som en speciell typ av variabler (som vi lärde oss i föregående kapitel). En lista kan, till skillnad från en vanlig variabel, innehålla flera värden som ligger på rad. Listor är användbara då man vill hålla reda på många saker samtidigt.

> Den formella beteckningen för en lista är *array* eller *vektor*. Termen vektor kan vara lite förvirrande, då den används på olika sätt i olika sammanhang. Ibland syftar man på en riktning eller position i en rymd (då har den x-, y- och kanske z-koordinater). Termen array används bara på engelska men inte på svenska. I denna bok kommer vi att använda termen lista.

Ett konkret exempel på en lista kan vara temperaturer olika dagar. Den första dagen var det 15 grader, den andra 13, den tredje 16 osv. Varje temperatur ligger var för sig lagrad i något som kallas *element*. Vi kan komma åt varje element i en lista med något som kallas för *index*.

Här är en illustration av en lista som innehåller fem olika element. Vi kan se varje elements index:

Figur 3.1: En lista

index	*1*	*2*	*3*	*4*	*5*
	17	-65	-20	9	42

Om vi ska skapa denna lista i Matlab/Octave kan vi skriva så här:

Exempel 3.1: Skapa lista

```
% Skapa listan:
temperature = [];

% Tilldela listans element olika värde genom index:
temperature(1) = 17;
temperature(2) = -65;
temperature(3) = -20;
temperature(4) = 9;
temperature(5) = 42;
```

För att sedan använda den kan vi t.ex. göra så här:

Exempel 3.2: Använda lista

```
% Lägg samman värdet på de olika elementen i listan och
% skriv ut medelvärdet:
summa = temperature(1) + temperature(2) + temperature(3)
   + temperature(4) + temperature (5);
disp(summa / 5);
```

Då får vi följande utskrift:

```
-3.400
```

3.1 Indexering av listor

I exempel 3.1 tilldelar vi listan fem olika heltal med hjälp av deras index. Genom en siffra indexeras ett specifikt element i listan. Index är det som står inne i parentesen efter variabelnamnet. Observera att indexeringen börjar på 1. Vi kan använda index både för att skriva och läsa ett specifikt element.

När man arbetar med listor och indexering av dem gäller det dock att se upp! Det är lätt hänt att man försöker att läsa ett element som inte finns. Det kommer bli fel då man kör programmet. Här är ett exempel:

Exempel 3.3: Felaktig indexering av en lista

```
temperature = [];

temperature(1) = 17;
temperature(2) = -65;
temperature(3) = -20;
temperature(4) = 9;
temperature(5) = 42;

% Skriv ut elementen med index 1 och 6
disp('First element: ');
```



```
disp(temperature(1));
disp('Sixth element: ');
disp(temperature(6)); % fel!
```

Då får vi följande utskrift:

```
First element:
 17
Sixth element:
error: temperature(6): out of bound 5
```

Först skrivs värdet på det första elementet, `temperature(1)` ut. Därefter får vi som väntat ett felmeddelande. Felet är *”temperature(6): out of bound 5”*. Vi försöker helt enkelt indexera ett element som ligger utanför vår lista.

Det går alltså inte att läsa ett index som inte finns, men däremot går det jättebra att skriva till ett index som inte finns. Då kommer det helt enkelt att skapas. Och om vi skriver till ett index som redan finns så kommer det gamla värdet att ersättas, precis som det funkar för vanliga variabler.

3.2 Listans längd

En lista har ett antal element. Antalet kallas även listans längd. När vi skriver till ett element med ett nytt index så växer listan - den blir längre. Vi kan mäta listans längd med hjälp av funktionen `size`. Den används såhär:

Exempel 3.4: Listans längd

```
temperature = [];

% Stoppa in tre element i listan,
% och skriv ut listans längd efter varje steg:
temperature(1) = 17;
disp(size(temperature, 2));
temperature(2) = -65;
disp(size(temperature, 2));
temperature(3) = -20;
disp(size(temperature, 2));
```

Vi får resultatet:

```
 1
 2
 3
```

> Du kanske undrar varför det står `size(temperature, 2)` och inte bara `size(temperature)`, det hade ju känts enklare. Anledningen har med matriser att göra men är mer avancerad än vad vi tar upp i denna bok.

Vad händer om vi inte lägger till element i direkt nummerordning? Då skapas nya element upp till och med det index som vi anger. Alla tidigare element får värdet noll:

Exempel 3.5: Listan fylls automatiskt på med tomma element

```
temperature = [];
temperature(3) = -20;
disp(temperature);
```

Vi får resultatet:

```
   0     0   -20
```

3.3 Listans sista element

Vår listas första element är alltid `temperature(1)`, men vad är dess sista element? Det sista elementets index varierar ju. Vi kan alltid komma åt det sista elementet genom att mäta längden, såhär:

```
temperature(size(temperature, 2))
```

Men eftersom det är långt och tjatigt att skriva så finns det ett kortare sätt:

```
temperature(end)
```

Och om vi vill lägga till nya element i slutet av listan, så att den växer, så kan vi skriva såhär:

```
temperature(end+1) = 13;
temperature(end+1) = 20;
% nu har vi lagt till två nya element
```


3.4 Förenklat sätt att skapa listor

Det finns också ett enklare sätt att skapa listor på:

Exempel 3.6: Förenklat sätt att skapa listor

```
temperature = [17 -65 -20 9 42];
```

Eller om vi vill skapa en lista med ett visst antal element, och alla element ska ha samma värde:

Exempel 3.7: Massproduktion

```
temperature = zeros(1, 4); % samma som [0 0 0 0]
temperature = ones(1, 4); % samma som [1 1 1 1]
```

Detta kan vara praktiskt om man vill skapa väldigt många element och slippa skriva in dem ett i taget. Också praktiskt om det önskade antalet element ligger i en variabel.

> Du kanske undrar varför det står `zeros(1, 5)` och inte bara `zeros(5)`, det hade ju känts enklare. Anledningen har åter igen med matriser att göra men är mer avancerad än vad vi tar upp i denna bok.

3.5 Fylla en lista med nummer i ordning

Ofta kommer vi vilja fylla en lista med tal på rad, t.ex. `[-2 -1 0 1 2]`. Då finns det ett koncisare sätt att skriva. Vi kan skriva det lägsta talet, sedan ett kolon `:` och sedan det högsta talet:

Exempel 3.8: Skapa en lista som innehåller alla heltal från -2 till 2

```
x = [-2 : 2];
```

Med hjälp av ett till kolon så kan vi dessutom ta kortare ”steg” mellan elementen:

Exempel 3.9: Anpassa steglängd mellan element

```
x = [-2 : 0.5 : 2]
```

Vi får resultatet:

```
  -2.00000   -1.50000   -1.00000   -0.50000    0.00000
   0.50000    1.00000    1.50000    2.00000
```

Med andra ord är alltså regeln `start:steglängd:slut`, eller bara `start:slut` och då blir steglängden automatiskt 1.

3.6 Söka i en lista

Vi har lärt oss att läsa elementet med ett visst index. Ibland kan vi istället vilja göra samma sak ”baklänges” och få svar på frågan: På vilket index ligger ett visst element? T.ex. vilken dag var temperaturen 9 grader? Då kan vi använda funktionen `find`:

```
temperature = [17 -65 -20 9 42];
find(temperature == 9) % ger resultatet 4
```

3.7 Matematiska operationer på varje element i listan

Vi kan enkelt göra samma matematiska operation på varje element i en lista, t.ex. multiplicera alla element med 2, eller med sig själva. Vi använder de vanliga operatorerna `+ - * /` men med en punkt framför:

Exempel 3.10: Matematiska operationer på varje element i listan

```
x = [1 2 3 4];
y = x .+ 1;
z = x .* 2;
w = x .* x;
disp(y);
disp(z);
disp(w);
```

Vi får resultatet:

```
   2  3  4  5
   2  4  6  8
   1  4  9  16
```

> Varför behöver vi skriva punkten framför multiplikationstecknet? Anledningen är mer avancerad än vad vi tar upp i denna bok, men har som vanligt med matriser att göra. Om vi glömmer punkten när vi skriver `z = x .* 2` så funkar det ändå, men om vi glömmer punkten när vi skriver `w = x .* x` så får vi felmeddelandet: `operator *: nonconformant arguments (op1 is 1x4, op2 is 1x4)`

Det är också möjligt att köra funktioner på varje element i en lista.

Exempel 3.11: Köra funktioner på varje element i en lista

```
x = [4 9 16 25];
disp(sqrt(x));
```

Vi får resultatet:

```
2 3 4 5
```

3.8 Slumptal med hjälp av listor

Om vi vill slumpa ett tal inom ett valfritt intervall, t.ex. 7 och 42, så gör vi såhär:

```
randi([7 42]); % ger ett slumptal fr.o.m 7 t.o.m 42
```

3.9 Övningar för listor

Övning (E), 3.1: Olika sätt att skapa samma lista

Skapa en lista som innehåller 8 element. Det första elementet ska ha värdet 20, det andra ska ha värdet 30, det tredje 40, och så vidare upp till och med 90. Hur många olika sätt kan du komma på för att skapa en sådan lista? Vilket tycker du känns lämpligast?

Övning (E), 3.2: `disp` med lista

Tänk dig följande kod:

```
temperature = [];
temperature(3) = -20;
temperature(6) = -13;
disp(temperature(1));
disp(size(temperature, 2));
```

Utan att skriva in koden själv i Matlab/Octave, vad tror du att vi får för utskrift?

4
Grafer och diagram

En stor anledning till att vi programmerar i Matlab/Octave är att det där är väldigt lätt att rita olika sorters grafer och diagram. I det här kapitlet kommer vi gå igenom en funktion som heter `plot` - den använder vi för att rita grafer. Vi kommer också lära oss funktionen `hist` som används för att rita ut histogram.

4.1 Plot

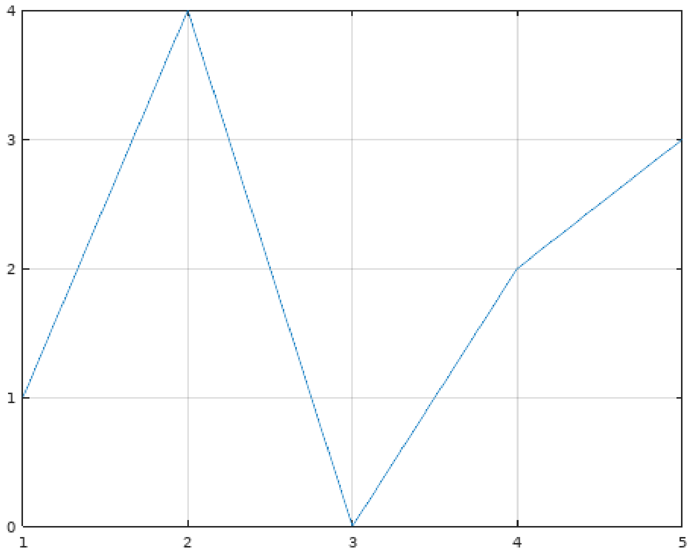
Tänk dig att vi med en termometer har mätt temperaturen en gång per dag, fr.o.m. måndag t.o.m. fredag, så att vi har 5 mätvärden. Om vi har våra mätvärden i en lista så kan vi ge listan till Matlab/Octave med kommandot `plot` och då kommer vi få upp en graf på skärmen:

Exempel 4.1: Vår första graf

```
temperaturPerDag = [1 4 0 2 3];
plot(temperaturPerDag);
```

Du ser resultatet här nedanför. Notera att linjen börjar på 1 längst till vänster, sen går den upp till 4, ner till 0, och så vidare, precis i samma ordning som vår lista. Notera också att den horisontella axeln går från 1 till 5, vilket motsvarar antalet element i vår lista. Den vertikala axeln går från 0 till 4, vilket beror på att den lägsta temperaturen i vår lista är 0 och den högsta är 4.

Figur 4.1: Vår första graf



4.1.1 Rita punkter

För att göra grafen lite tydligare så kan vi be Matlab/Octave att rita små cirklar vid varje mätpunkt. Det gör vi genom att lägga till `, '-o'` innan slutparentesen efter `plot`:

Exempel 4.2: Vår andra graf

```
temperaturPerDag = [1 4 0 2 3];
plot(temperaturPerDag, '-o');
```

> Det finns många fler möjligheter att ställa in och anpassa hur grafen ser ut. T.ex. färger, storlek, var de vertikala och horisontella axlarna ska börja och sluta, med mera. Du kan läsa mer om inställningsmöjligheterna på *https://se.mathworks.com/help/matlab/ref/plot.html* och *https://se.mathworks.com/help/matlab/creating_plots/change-axis-limits-of-graph.html*

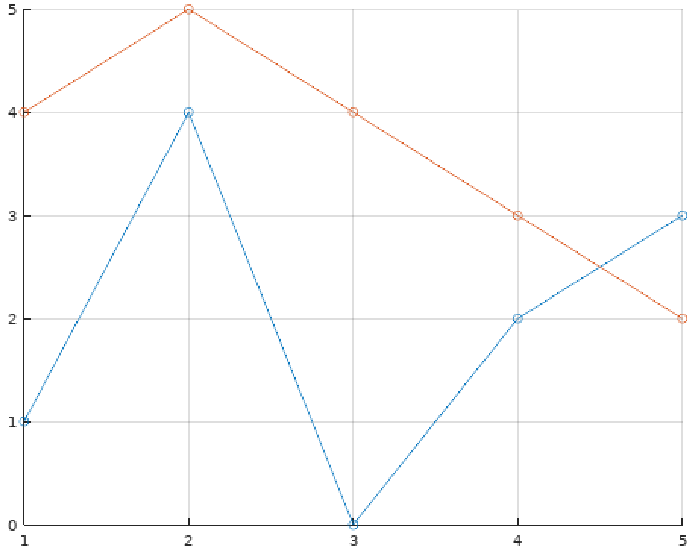
4.1.2 Flera grafer på samma gång

Vad händer om vi använder `plot` flera gånger på rad, fast med olika listor? Jo, varje `plot` rensar skärmen från allt som syntes där tidigare, så vi ser bara resultatet av den sista `plot`:en. Men om vi skriver `hold on;` överst i vårt program så går det att se flera resultat i samma graf. Varje `plot`-linje får då en egen färg, för att vi lättare ska kunna se skillnad på dem:

Exempel 4.3: Två grafer i en

```
hold on;
plot([1 4 0 2 3], '-o');
plot([4 5 4 3 2], '-o');
```

Figur 4.2: Två grafer i en



4.1.3 Plotta med två listor, i två dimensioner

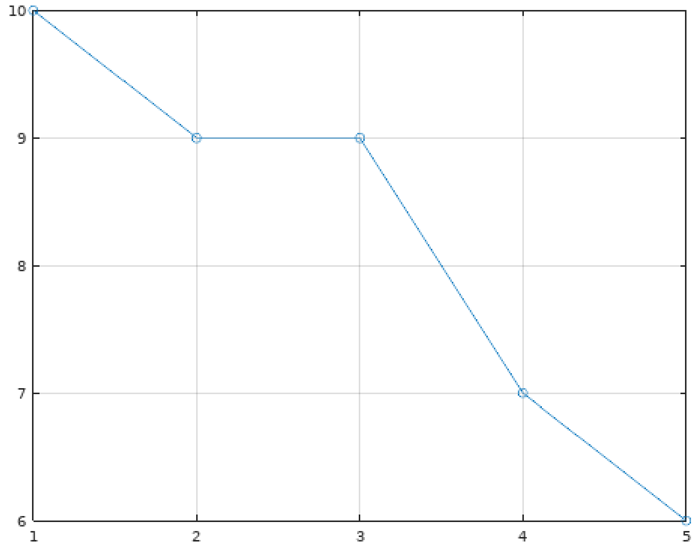
Tänk om vi hade tänkt mäta temperaturen varje dag i en vecka, men vi glömde mäta på torsdag och fredag. Hur vill vi att vår graf ska se ut då? Det vore bra om det syntes i grafen att två dagar saknas i mitten. Vi kan förstås plotta som vanligt...

Exempel 4.4: Syns inte att torsdag och fredag saknas

```
temperaturPerDag = [10 9 9 7 6];
plot(temperaturPerDag, '-o');
```

(Se graf på nästa sida...)

Figur 4.3: Syns inte att torsdag och fredag saknas

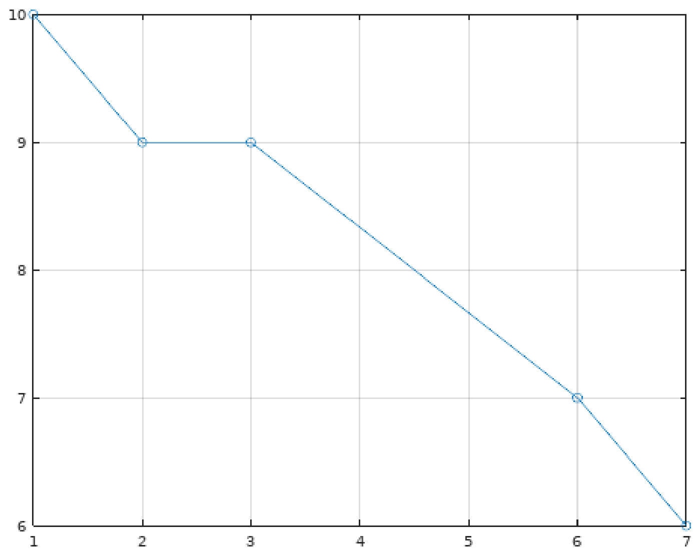


... men då ser det ut som att vi mätte fem dagar på rad. Det vore bättre om vi fick ett ”glapp” i grafen, så att det inte finns någon punkt på $x = 4$ och $x = 5$, men att det sedan finns punkter igen på $x = 6$ och $x = 7$. Kan vi åstadkomma detta i Matlab/Octave? Såklart vi kan! Men då måste vi använda plot på ett nytt sätt. Istället för att bara skicka in en lista så skickar vi in två listor:

Exempel 4.5: Syns att torsdag och fredag saknas

```
dagar = [1 2 3 6 7]; % vi hoppar över 4 och 5
temperaturPerDag = [10 9 9 7 6];
plot(dagar, temperaturPerDag, '-o');
```

Figur 4.4: Syns att torsdag och fredag saknas



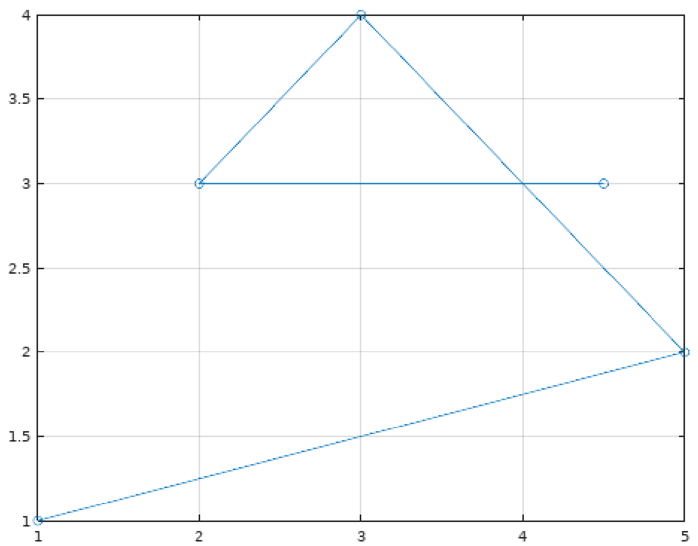
Nu fick vi det glapp som vi ville ha, hurra!

Notera att vi på detta sätt kan rita vilka former som helst, även linjer som vänder tillbaka, korsar sig själv, osv:

Exempel 4.6: Plotta vilken form som helst

```
plot([1 5 3 2 4.5], [1 2 4 3 3], '-o');
```

Figur 4.5: Kors och tvärs



Om vi enkelt vill rita en rät linje, så kan vi använda ovanstående metod och bara ange linjens startpunkt och slutpunkt, såhär: `plot([xstart xend], [ystart yend])`. Exempel:

Exempel 4.7: Rät linje

```
plot([0 10], [7 5]);
```

4.1.4 Plotta en funktion av x

Nu har vi lärt oss allt vi behöver för att kunna rita en graf av en funktion av x. Som vi såg i delkapitel 3.7 så är det möjligt att köra en funktion på varje element i en lista. Låt oss testa att rita ut en graf av funktionen `sind` några varv:

Exempel 4.8: Plotta en sinusvåg

```
x = [0 : 10 : 1080]; % fyll lista med x-värden att visa
y = sind(x); % kör funktionen sind på varje element
plot(x, y); % rita ut grafen!
```

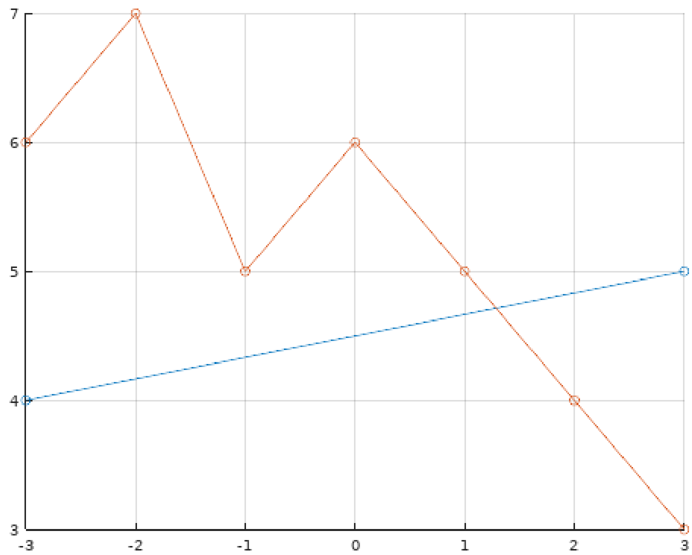
Notera att vi också använder det vi lärde oss i delkapitel 3.5 där vi fyllde en lista med alla nummer mellan -2 och 2.

4.1.5 Övningar för plot

Övning (E), 4.1: Återskapa kod efter bild

Skriv koden för att rita följande bild:

Figur 4.6



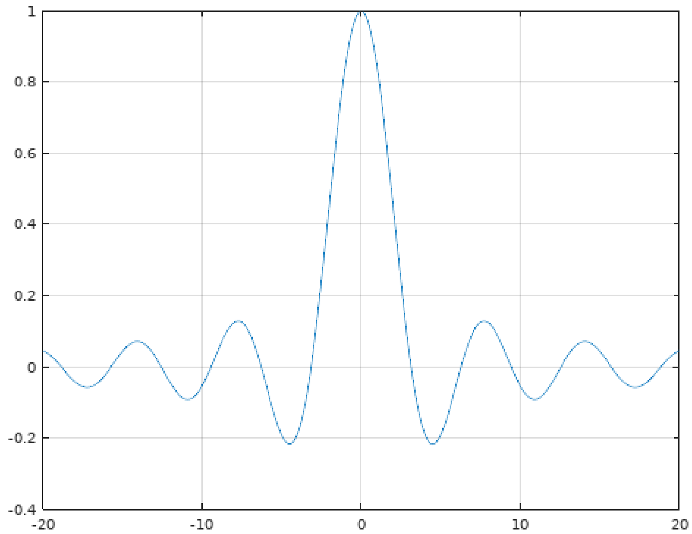
Övning (E), 4.2: Plotta given funktion

Här är några ofullständiga rader kod:

```
x = % här ska det stå något
y = sin(x) ./ x;
% här ska det stå kod för att rita ut grafen
```

Ersätt kommentarerna med kod för att rita följande bild:

Figur 4.7



4.2 Histogram

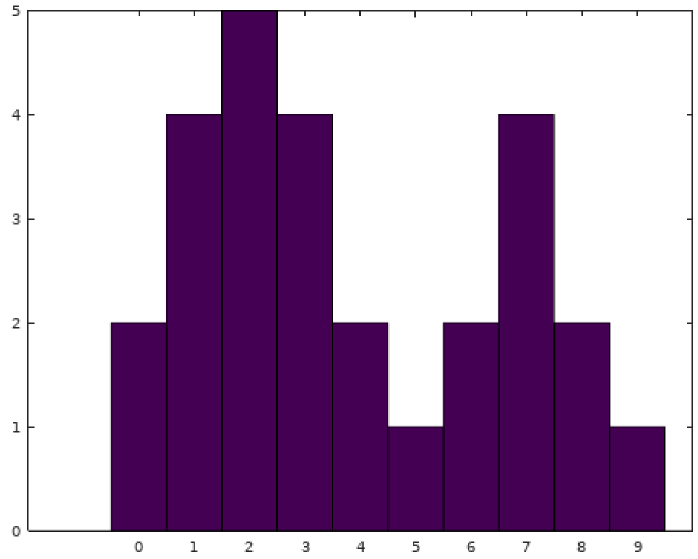
Om vi istället har temperaturen från många dagar så kanske vi hellre vill se en slags sammanfattning - hur många dagar var det 3 grader varmt? Hur många dagar var det 4 grader? Då passar det bra med ett så kallat histogram, även känt som stapeldiagram eller stolpdiagram.

Exempel 4.9: Vårt första histogram

```
temperaturPerDag = [1 4 0 2 3 4 3 6 7 8 9 8 7 7 6 5 7 3 2
   3 2 2 1 1 2 1 0];
hist(temperaturPerDag, 0:9);
```

(Se histogram på nästa sida...)

Figur 4.8: Vårt första histogram



I histogrammet kan vi lätt se att det var 3 grader varmt fyra dagar och 4 grader varmt i två dagar. Funktionen `hist` tar alltså två argument. Det första är en lista av tal att sammanfatta, som kan vara hur lång som helst. Det andra argumentet berättar hur vi vill sammanfatta listan, närmare bestämt vilka staplar vi vill ha. `0:9` betyder att den första stapeln ska vara 0 och den sista stapeln ska vara 9. Prova vad som händer om du ändrar till t.ex. `0:5` eller `-5:15`! (Ser du likheten med delkapitel 3.5 där vi fyllde en vektor med alla nummer mellan -2 och 2?)

När vi vill visa en sån här lista, där vi vet att den lägsta temperaturen är 0 och den högsta är 9, så passar det förstås bäst att ha just de staplarna i diagrammet. Mer generellt, när vi har en lista som bara innehåller heltal (inga decimaltal), och har ett ganska litet antal olika heltal, så passar det bäst att ha en stapel för varje heltal. Vi kan automatisera det såhär:

Exempel 4.10: Välja rätt antal staplar automatiskt

```
temperaturPerDag = [1 4 0 2 3 4 3 6 7 8 9 8 7 7 6 5 7 3 2
   3 2 2 1 1 2 1 0];
hist(temperaturPerDag,
   min(temperaturPerDag):max(temperaturPerDag));
```

Med ovanstående kod kan vi lätt lägga till nya temperaturer utan att behöva komma ihåg att ändra på `hist`-raden.

Men om vi istället har en lista som innehåller väldigt många olika heltal, eller decimaltal, så passar det inte bäst med något särskilt antal staplar utan är mer av en smaksak.

5 Selektion (med if)

Ett program behöver ofta göra olika val beroende på olika värden på saker och ting (t.ex. olika variablers värden). Då använder man något som kallas för *selektion*. I det här kapitlet kommer vi att gå igenom `if`-satsen. Den utför selektion men brukar i sig kallas för *villkorssats*.

Hittills har vi bara skrivit in vår kod i ”Command window” men nu kommer våra program att växa och då blir det mycket enklare om vi använder filer. Om du inte redan är bekant med det, se kapitel B för hur du ska göra med just ditt program (om du använder Matlab, Octave eller Octave Online).

5.1 Läsa in variabler

Innan vi går vidare med selektion så ska vi ta ett kort sidospår och lära oss hur man kan läsa in variabler medan en kodsnutt körs. Det är smidigt om man vill köra samma kodsnutt flera gånger, men testa olika värden på variablerna. Hittills när vi har skrivit kod i ”Command window” har vi ju bara gjort det för oss själva. Men när vi nu börjar arbeta med filer, så har vi ju möjlighet att låta andra personer köra vår kod.

Vi skapar en ny fil och skriver följande kod:

Exempel 5.1: Läsa in variabler

```
a = input('Ange variabeln a: ');
b = input('Ange variabeln b: ');
disp(a+b);
```

Funktionen `input` skriver alltså ut en text på skärmen, och låter användaren (den som kör vår kod) skriva in något. När användaren tryckt på Enter-tangenten, så fortsätter vår kod köras, och tilldelar det som användaren har skrivit in till variablerna.

Övning (E), 5.1: input

Testa funktionen `input`. Prova lite olika utskrifter och olika namn på variabler.

Testa att köra exemplet ovan, men istället för att skriva in en siffra så skriver du in ditt namn. Vad får du för felmeddelande? Vad tror du det beror på?

5.2 if-satsen

En `if`-sats jämför alltså två värden med varandra. En `if`-sats på svenska blir alltså en om-sats:

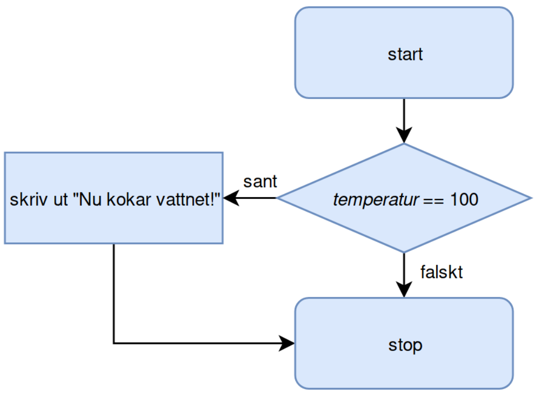
```
OM något SÅ
   gör detta.
SLUT OM
```

T.ex. så kan man kontrollera hur varmt vatten är:

```
OM temperatur är 100 SÅ
   skriv ut "Nu kokar vattnet!" på skärmen.
SLUT OM
```

Detta kan illustreras visuellt som ett flödesschema som ser ut så här:

Figur 5.1: If-sats som flödesschema



Låt oss prova detta i Matlab/Octave. Vi lägger till lite kod för att användaren ska få mata in värdet på variabeln `temperature`:

Exempel 5.2: Vår första if-sats

```
temperature = input('Ange temperatur: ');
if temperature == 100
    disp('Nu kokar vattnet!');
end
```

Du minns väl att det är skillnad på jämförelseoperatorn och tilldelningsoperatorn? Om du inte minns, se delkapitel 1.3 och delkapitel 2.2. När vi arbetar med `if`-satser använder vi jämförelseoperatorn just för att jämföra två olika tal (eller i exemplet ovan, värdet av variabeln `temperature` och talet `100`).

Lägg märke till att det inte är något semikolon `;` efter `if`-satsen. I kodblocket, alltså det som ligger efter `if temperature == 100` och innan `end`, ligger den kod som vi vill utföra, ifall villkorssatsen visar sig stämma. I exemplet ovan har vi bara en rad kod att utföra i kodblocket.

Om vi anger att vattnet är 100 grader, får vi alltså följande resultat:

```
Ange temperatur: 100
Nu kokar vattnet!
```

5.2.1 else

Att använda en `if`-sats utan något mer, gör att vi kör ett stycke kod om villkorssatsen visar sig stämma. Annars gör vi ingenting speciellt utan programmet fortsätter bara att köra. Låt oss fortsätta med temperaturer. Om vi i körningen av exempel 5.2 angav något annat än 100, slutar programmet abrupt:

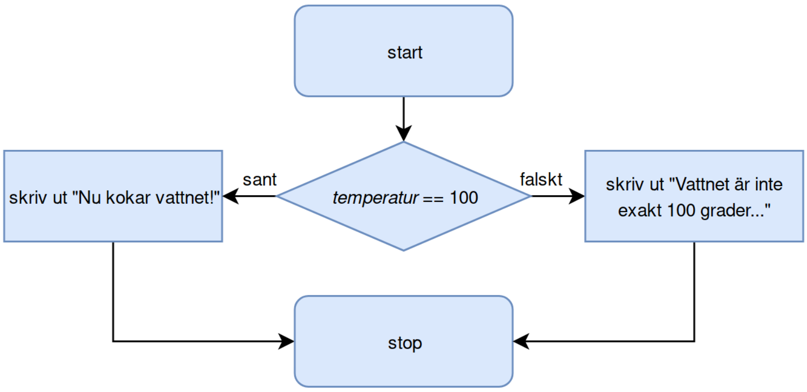
```
Ange temperatur: 89
```

Men ofta vill man ju faktiskt göra något annat, om det visar sig att villkorssatsen *inte* stämmer. Låt oss fortsätta med det kokande vattnet, först på svenska:

```
OM temperatur är 100 SÅ
   Skriv ut "Nu kokar vattnet!" på skärmen.
ANNARS
   Skriv ut "Vattnet är inte exakt 100 grader..."
SLUT OM
```

Detta kan illustreras som flödesschema så här:

Figur 5.2: if och else som flödesschema



Låt oss programmera detta i Matlab/Octave:

Exempel 5.3: Vår första else-sats

```
temperature = input('Ange temperatur: ');
if temperature == 100
    disp('Nu kokar vattnet!');
else
    disp('Vattnet är inte exakt 100 grader...');
end
```

Nu får vi i alla fall ett meddelande, om vi anger att temperaturen är annat än 100 grader:

```
Ange temperatur: 89
Vattnet är inte exakt 100 grader...
```

5.2.2 if-satser med mindre än-operatorn <

Låt oss testa `if`-satser med en annan jämförelseoperator. Vi tar mindre än-operatorn <.

Vi tar det först på svenska:

```
OM temperatur är mindre än 100 SÅ
    Skriv ut "Vattnet är inte tillräckligt varmt än..." på
   skärmen.
ANNARS
    Skriv ut "Vattnet kokar!"
```

Kodat i Matlab/Octave blir det:

Exempel 5.4: Mindre än-operatorn

```
temperature = input('Ange temperatur: ');
if temperature < 100
    disp('Vattnet är inte tillräckligt varmt än...');
else
    disp('Vattnet kokar!');
end
```

På samma sätt som med operatorerna == och <, kan du använda `if`-satser med de övriga jämförelseoperatorerna som finns listade i delkapitel 1.3.

5.2.3 Input med bokstäver

Ibland känns det lite tråkigt att vi bara kan prata siffror med datorn. Det vore roligare att kunna säga små ord till den, i alla fall ”j” och ”n” för att symbolisera ja/nej. Vi skapar ett program som ställer frågan ”Är det fint väder?”. Om användaren svarar ”j” skriver programmet ut ”Vi går på picknick!”. Annars händer ingenting. Men hur ska datorn kunna förstå svaret ”j”? Vi kan använda följande trick:

Exempel 5.5: Kontrollera vädret

```
j = 1; % det här är tricket
svaret = input('Är det fint väder? ');
if svaret == j
    disp('Vi går på picknick!');
end
```

5.3 Övningar

Övning (E), 5.2: Kontrollera vädret (fortsättning)

Arbeta vidare på exempel 5.5 men lägg till att användaren kan svara ”n”. Då skriver programmet ut ”Vi stannar inne och läser en bok”. Är det klurigt? Fundera på värdet på variabeln `n`.

Övning (E), 5.3: Var är det kallast?

Skapa ett program där man får mata in temperaturen i Östersund och Göteborg. Programmet ska sedan berätta var det är kallast. Men om det är lika kallt i båda städerna så ska programmet berätta detta istället.

Övning (E), 5.4: Felaktig if-sats

Något stämmer inte riktigt med följande if-sats:

```
x = 9;
if x = 10
    disp('den är 10!');
end
```

När vi försöker köra koden så får vi ett felmeddelande - vad är det som inte stämmer? Skriv om koden så att det blir rätt!

6 Iteration (med while)

Iteration är att köra ett kodstycke om och om igen, utan att behöva skriva kodstycket flera gånger. Ofta brukar ett sånt kodstycke kallas loop. Kodstycket upprepas så länge ett visst villkor är sant. Ofta vill man upprepa något ett visst antal gånger. Om man t.ex. vill gå igenom en lista med femtio element (saker i listan) i och göra något med varje element, så loopar man ett kodstycke femtio gånger.

I Matlab/Octave finns det flera typer av loopar med olika syften. Den enda loop vi kommer att arbeta med i denna bok är `while`, eftersom den är enklast att förstå, och går att använda till allt.

6.1 while-loopen

Man kan beskriva `while`-loopen på svenska så här:

```
MEDAN någonting SÅ
   Gör detta
SLUT MEDAN
```

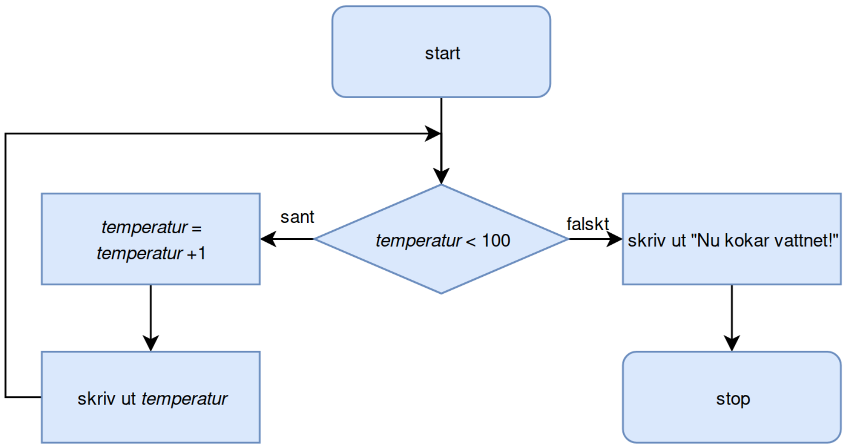
Vi kan t.ex. be någon ange temperatur. Så länge temperaturen är mindre än 100, så kör vi ett varv i loopen. För varje varv i loopen, så ökar vi temperaturen med en grad och skriver ut den. När temperaturen är 100 så går vi vidare i programmet och skriver ut ett meddelande:

(Se nästa sida...)

```
MEDAN temperatur är mindre än 100 SÅ
   Plussa på temperaturen med ett
   Skriv ut temperaturen
SLUT MEDAN
Skriv ut "Nu kokar vattnet!"
```

Detta kan illustreras som flödesschema så här:

Figur 6.1: En while-loop som flödesschema



Precis som med selektion (`if`-satser), så kan vi använda samtliga jämförelse-operatorer (som finns listade i delkapitel 1.3) när vi jobbar med iteration och `while`-loopar. Nu testar vi med mindre än-operatorn <.

I Matlab/Octave blir det:

Exempel 6.1: Vår första while-loop

```
temperature = input('Ange temperatur: ');
while temperature < 100
   temperature = temperature + 1;
   disp('Temperaturen är nu: ');
   disp(temperature);
end
disp('Nu kokar vattnet!');
```

Om vi anger den nuvarande temperaturen som 92 får vi följande resultat:

```
Ange temperatur: 92
Temperaturen är nu:
93
Temperaturen är nu:
94
Temperaturen är nu:
95
Temperaturen är nu:
96
```

```
Temperaturen är nu:
97
Temperaturen är nu:
98
Temperaturen är nu:
99
Temperaturen är nu:
100
Nu kokar vattnet!
```

Om vi istället anger temperaturen 100 (eller mer), kommer kodstycket i loopen aldrig att köras. Programmet hoppar då direkt till ”Nu kokar vattnet!”-meddelandet och vi får följande resultat:

```
Ange temperatur: 100
Nu kokar vattnet!
```

6.2 Oändliga loopar

När vi skriver koden för en loop så gäller det att se upp. Om vår jämförelse aldrig slutar vara sann, så fortsätter loopen att snurra i all oändlighet. Det kan t.ex. hända om vi glömmer att skriva koden för att plussa på temperaturen inuti loopen, såhär:

Exempel 6.2: Oändlig loop, varning!

```
temperature = input('Ange temperatur: ');
while temperature < 100
   disp('Temperaturen är nu: ');
   disp(temperature);
end
```

Om vi kör en oändlig loop så går det inte att köra någon mer kod efteråt. Vi kan då behöva stänga av hela Matlab/Octave för att avbryta vår loop, och sedan starta det igen.

6.3 for-loopen

Som sagt finns det flera typer av loopar. Den så kallade `for`-loopen har fördelen att den inte kan råka bli oändlig. Den är även mer koncis. Här är ett exempel på en `for`-loop:

```
start_temperature = input('Ange temperatur: ');
for t = start_temperature:100
   disp('Temperaturen är nu: ');
   disp(t);
end
```

Det finns ingen situation där man *måste* använda `for`-loopen, men det finns däremot situationer där man *måste* använda `while`-loopen. För att hålla denna bok enkel kommer vi inte att arbeta något mer med `for`-loopen.

6.4 Gissa talet!

Låt oss tillverka ett enkelt litet spel. Spelaren ska få gissa på ett tal mellan 1 och 100, ett tal som vi programmerare redan bestämt innan. Talet är 42. Så länge spelaren gissar fel, ska den få fortsätta att gissa. När spelaren har gissat rätt skriver vi ut ett grattis-meddelande. Vi tar det först på svenska:

```
Skriv ut "Gissa ett tal mellan 1 och 100"
Mata in tal
MEDAN talet inte är 42 SÅ
   Skriv ut "Fel. Gissa igen"
   Mata in tal
SLUT MEDAN
Skriv ut "Grattis! Du gissade rätt!"
```

Kodat i Matlab/Octave blir det:

Exempel 6.3: Gissa talet

```
guess = input('Gissa ett tal mellan 1 och 100:');
while guess ~= 42
    guess = input ('Fel! Gissa igen: ');
end
disp('Grattis! Du gissade rätt!');
```

(Se nästa sida...)

Här har min kompis försökt spela spelet:

```
Gissa ett tal mellan 1 och 100: 50
Fel! Gissa igen: 25
Fel! Gissa igen: 37
Fel! Gissa igen: 44
Fel! Gissa igen: 41
Fel! Gissa igen: 42
Grattis! Du gissade rätt!
```

Det här spelet är inte så roligt att spela mer än en gång. Vi kommer att återkomma till spelet i kapitel 7 där du också kan vidareutveckla det i en klurig övning.

6.5 Övningar

Övning (E), 6.1: Tal mellan 1 och 20

Skapa ett program som använder iteration för att skriva ut alla tal mellan 1 och 20.

Övning (E), 6.2: Tal mellan 1 och 100

Skapa ett program där användaren får mata in valfritt tal upp till 100. Programmet skriver sedan ut alla tal, från talet som användaren matade in upp till och med 100. Om man matar in ett tal som är större än 100 så stängs programmet av direkt.

Exempel:

```
Mata in ett tal: 93
93
94
95
96
97
98
99
100
```

Övning (E), 6.3: Singla slant

Be användaren mata in hur många gånger denne vill singla slant. Programmet ska sedan slumpvis mata ut om det blir krona eller klave, lika många gånger som användaren angett.

För att implementera övningen i Matlab/Octave behöver du använda funktionen `randi` för att slumpa fram nummer.

Övning (E), 6.4: Yatzy

Skapa ett program som fem gånger slumpar fram tärningsslag (tal mellan 1 och 6).

För att implementera övningen i Matlab/Octave behöver du använda `randi` för att slumpa fram nummer.

Övning (E), 6.5: Väderstationen

Skapa en lista som ska innehålla temperaturmätningar från en väderstation. I programmets början ska användaren få ange hur många mätningar som har gjorts. Därefter får användaren mata in olika temperaturer. Programmet ska sedan skriva ut de olika temperaturerna och medeltemperaturen.

Övning (M), 6.6: Multiplikationstabellen

Skapa ett program som skriver ut multiplikationstabellen, dvs skriv ut resultatet av att multiplicera $1 * 1$, sedan $1 * 2$, och så vidare hela vägen upp till $10 * 10$.

Tips: Du behöver använda två loopar - den ena inuti den andra.

7 PROBLEMLÖSNING

När man programmerar är det viktigt att kunna dela upp problemet i mindre delar. Det är faktiskt en av de allra viktigaste färdigheterna som en programmerare har. Ett mindre program kanske kan brytas ner till några få beståndsdelar, medan ett stort och komplext program kan innehålla tusen och åter tusen beståndsdelar. Man får bryta ner problemen i olika delar. Man börjar med det största problemet och bryter ner det i mindre underproblem. Dessa underproblem får sedan egna underproblem osv.

En tumregel för när man brytit ner problemen till lagom stora delar, är då det är möjligt söka efter lösningar med hjälp av en sökmotor på nätet. För att de ska vara sökbara, bör problemen kunna formuleras i några få nyckelord.

I det här kapitlet kommer vi att titta på två problem och hur man kan gå till väga för att lösa dem. Det första problemet är ganska kort och tanken är att du ska få lära dig hur du bryter ner problemet i så små delar att de är sökbara på nätet. Det andra problemet börjar från noll och byggs upp till en färdig lösning och du får se varje steg på vägen.

7.1 Nedbrytning och webbsökning

Låt oss igen ta gissa talet-spelet som exempel (från delkapitel 6.4). Istället för att alltid använda talet 42, så ska spelet slumpvis hitta på ett heltal mellan 1 och 100. Användaren ska sedan gissa talet. Gissar man fel ska programmet nu svara *”Fel. Mitt tal är lägre”* respektive *”Fel. Mitt tal är högre.”*. I slutet skriver programmet ut antalet försök.

Här är ett exempel på hur det kan se ut vid körning:

```
Gissa ett tal mellan 1 och 100: 15
Fel. Mitt tal är högre. Gissa igen: 50
Fel. Mitt tal är lägre. Gissa igen: 40
Fel. Mitt tal är högre. Gissa igen: 45
Fel. Mitt tal är högre. Gissa igen: 48
Rätt! Så här många gissningar behövde du:
5
```

Hur gör vi nu för att lösa detta? Vi kan kanske inte söka efter och hitta hela lösningen på webben. Däremot kan vi dela upp det i mindre bitar och söka efter dem enskilt:

1. Vi kan söka information om hur man slumpar fram ett tal, genom att söka på ”matlab random”.

2. Vidare kan vi söka reda på hur man tar emot ett tal ifrån en användare, genom att söka på ”matlab user input”.

3. För att jämföra två tal med varandra kan vi söka på ”matlab compare numbers”

Nu kanske vi redan vet hur man löser flera av dessa problem utantill. Då behöver vi såklart inte söka efter informationen på webben - men tankesättet är ändå detsamma när man bryter ner problem i mindre delar. På det här sättet har vi kommit närmare en lösning. Nu är det bara att sätta ihop dessa delar och skapa ett litet spel. Prova själv!

När man söker är det viktigt att inte ge upp i första taget. Men generellt gäller att om man inte hittar det man söker på något av de första 10 resultaten, är det oftast bättre att försöka med några nya sökord, än att leta vidare i lägre prioriterade resultat.

Övning (E), 7.1: Gissa talet

Skapa spelet ”gissa talet” (se ovan) där datorn slumpar fram ett tal mellan 1 och 100 och användaren får gissa vilket tal det är.

När du spelar spelet så finns det en strategi som är den optimala för att alltid behöva göra så få gissningar som möjligt. Kan du hitta den strategin?

Övning (M), 7.2: Datorn gissar talet

Låt oss göra ungefär samma övning som ovan, fast denna gång är det användaren som bestämmer sig för ett tal (det räcker om den gör det tyst i huvudet) och datorn som gissar. Användaren får svara `r`, `h` eller `l` för ”rätt”, ”högre” eller ”lägre”. Såhär skulle en exempelkörning kunna se ut:

```
Jag gissar på:
50
Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? h
Jag gissar på:
75
Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? l
Jag gissar på:
62
Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? h
Jag gissar på
68
Är det [r]ätt? Eller är ditt tal [h]ögre eller
   [l]ägre? r
Såhär många gissningar behövde jag:
4
```

Kan du programmera datorn så att den använder samma optimala gissnings-strategi som du kom på när du gissade själv?

7.2 Hur ett program växer fram

Låt oss ta ett exempel där vi grundligt förklarar hur vi tänker medan vi bygger upp programmet och löser problemet. Vi bygger upp programmet gradvis, steg för steg. När du ska lösa egna problem så kan det vara bra att tänka på samma steg-för-steg-sätt.

Vi gör ett program som skriver ut alla primtal mellan 1 och 100. Idén kommer från en grekisk herre vid namn Erastothenes som levde för några tusen år sedan, långt innan det fanns datorer.

Ett sätt att skriva ut alla tal mellan 1 och 10 är med hjälp av en loop:

```
i = 1;
while i <= 10
   disp(i);
   i = i + 1;
end
```

Tänk dig att vi på något magiskt sätt redan vet vilka tal mellan 1 och 10 som är primtal och inte. Hur skriver vi ut bara dem, och inte resten? Vi behöver någon sorts `if`-sats (som vi lärde oss i kapitel 5):

```
i = 1;
while i <= 10
   if should_print(i)
      disp(i);
   end
   i = i + 1;
end
```

Ovanstående kod funkar inte i verkligheten- om du försöker köra den så klagar förstås Matlab/Octave på att `should_print` inte existerar. Men vi kan förstås skriva in den manuellt som en lista (som vi lärde oss i kapitel 3):

```
should_print = [1 1 1 0 1 0 1 0 0 0];
i = 1;
while i <= 10
   if should_print(i)
      disp(i);
   end
   i = i + 1;
end
```

Detta är förstås ingen lösning - vårt mål är ju att datorn ska räkna ut vilka tal som är primtal, inte att vi ska behöva skriva in det manuellt. Men det är en början.

En allmänt bra strategi för att lösa ett svårt problem när vi inte har någon aning om lösningen, är att börja med att lösa ett relaterat men mycket enklare problem. Så istället för att räkna ut alla primtal, låt oss börja med att räkna ut vilka tal som *inte* är delbara med 2, förutom talet 2 självt:

```
% skapa lista med 10 element. Varje element har värdet 1
should_print = ones(1, 10);
% för alla tal som är en multipel av 2,
% sätt should_print till 0
j = 4;
```

```
while j <= 10
   should_print(j) = 0;
   j = j + 2;
end
% skriv ut
i = 1;
while i <= 10
   if should_print(i)
      disp(i);
   end
   i = i + 1;
end
```

Förstår du while-loopen? Först sätter den `should_print(4)=0`, sen sätter den `should_print(6)=0`, och så vidare med 8 och 10.

Tänk om vi vill gå högre, alltså öka 10 till 25. Då måste vi ändra på tre olika ställen. Det är jobbigt i längden. Låt oss istället lägga tian i en variabel så att vi lätt kan ändra den:

```
limit = 25;
should_print = ones(1, limit);
j = 4;
while j <= limit
   should_print(j) = 0;
   j = j + 2;
end
i = 1;
while i <= limit
   if should_print(i)
      disp(i);
   end
   i = i + 1;
end
```

På samma sätt kan vi ju skriva ut alla tal som varken är delbara med 2 eller 3:

```
limit = 25;
should_print = ones(1, limit);
% för alla tal som är en multipel av 2,
% sätt should_print till 0
j = 4;
while j <= limit
   should_print(j) = 0;
   j = j + 2;
end

% för alla tal som är en multipel av 3,
% sätt should_print till 0
j = 6;
```

```
while j <= limit
    should_print(j) = 0;
    j = j + 3;
end

% skriv ut
i = 1;
while i <= limit
    if should_print(i)
        disp(i);
    end
    i = i + 1;
end
```

Börjar du se mönstret? Tänk om vi gör detta inte bara för 2 och 3 utan för *alla* tal upp till och med 25. Då blir det ju bara primtalen kvar som skrivs ut. Men vi vill ju inte behöva skriva nästan samma kod 25 gånger på rad, så det är bättre att lägga in den koden i en till, yttre loop:

```
limit = 25;
should_print = ones(1, limit);
i = 2;
while i <= limit
    % för alla tal som är en multipel av i,
    % sätt should_print till 0
    j = i*2;
    while j <= limit
        should_print(j) = 0;
        j = j + i;
    end
    i = i + 1;
end

% skriv ut
i = 1;
while i <= limit
    if should_print(i)
        disp(i);
    end
    i = i + 1;
end
```

Hurra, det funkar! Vi har lyckats skriva ett kort program som skriver ut alla primtal mellan 1 och 25, och det är lätt att öka gränsen till 100. Programmet är dock inte så snabbt som det skulle kunna vara. Det gör en hel del beräkningar i onödan - det sätter `should_print` till `0` flera gånger för samma tal. Vi kan göra tre optimeringar.

För det första går den yttre loopen onödigt långt. Den går hela vägen upp till och med 25, men det behövs inte. Tänk dig att vi står på `i==7` som ju är ett primtal. Då kommer vi att sätta `should_print=0` för $7 * 2 = 14$ och $7 * 3 = 21$. Men vi har ju redan gjort detta då vi stod på `i==2` och `i==3`. Mer generellt så behöver den yttre loopen bara gå upp till och med 5, dvs kvadratroten ur 25. Vi använder funktionen `sqrt()` för att räkna ut kvadratroten.

För det andra. Först besöker vi alla multiplar av 2, dvs 4,6,8,10,12,14,16 osv. Och lite senare besöker vi alla multiplar av 4, dvs 8,12,16 osv, men det är ju onödigt, alla de är ju redan satta. När `i==4` så borde vi inte köra den inre loopen alls, eftersom 4 inte är ett primtal. Mer generellt, när `should_print(i)` är 0 så borde vi inte köra den inre loopen alls.

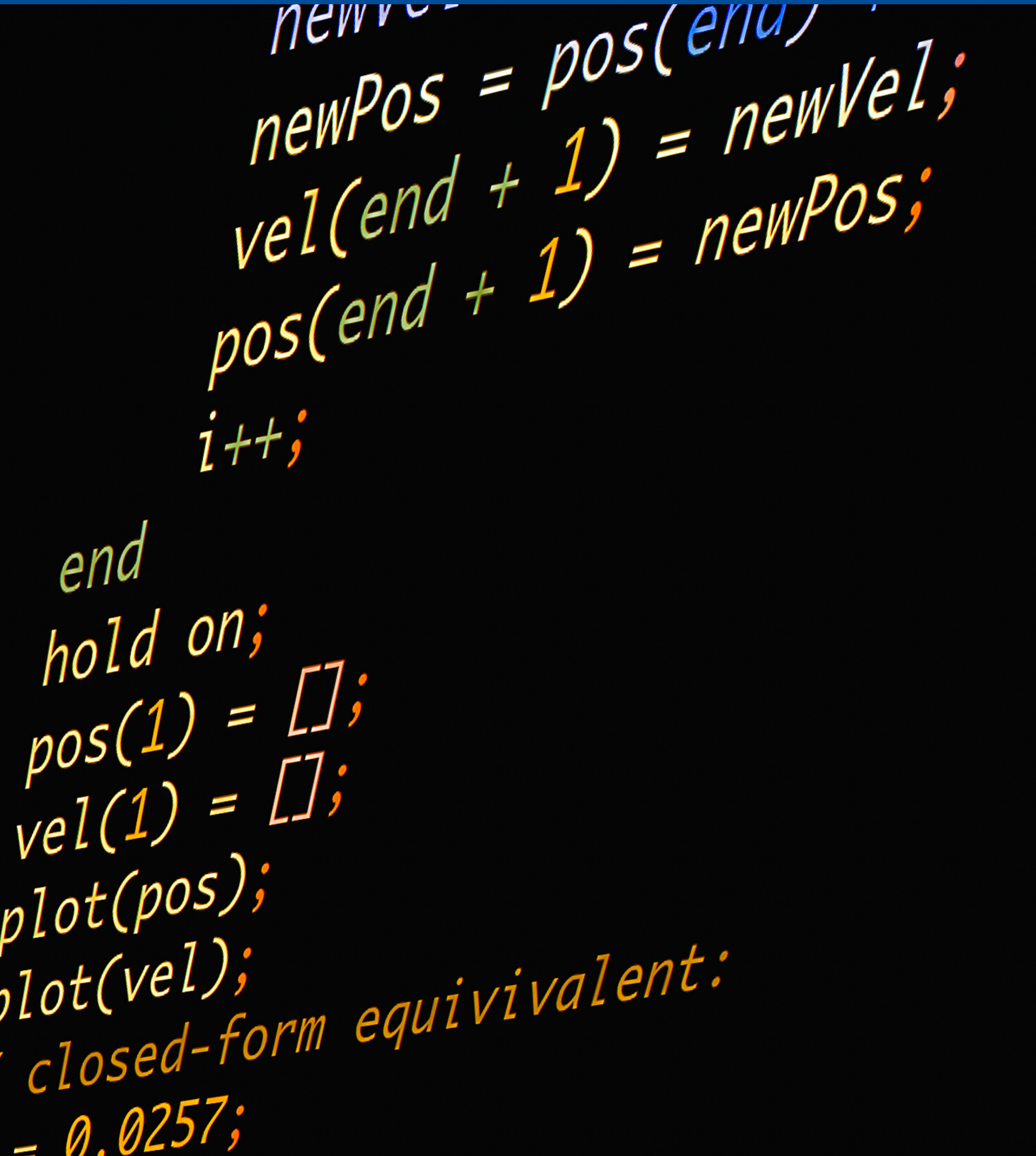
För det tredje så börjar den inre loopen på ett onödigt lågt tal. När vi står på `i==3` så behöver vi inte besöka `j==6` för det har vi redan gjort, alltså kan den inre loopen börja på `j==9`. Och när vi står på `i==5` så behöver vi inte besöka `j==10`, `j==15` eller `j==20`, för det har vi redan gjort, alltså kan den inre loopen då börja på `j==25`. Mer generellt kan den inre loopen börja på `j=i*i`.

Om vi implementerar alla dessa tre optimeringar så får vi den färdiga Sieve of Erastothenes:

```
limit = 25;
should_print = ones(1, limit);
i = 2;
while i <= sqrt(limit) % optimering 1: sluta tidigare
   if should_print(i) % optimering 2
      j = i * i; % optimering 3: börja loopa senare
      while j <= limit
         should_print(j) = 0;
         j = j + i;
      end
   end
   i = i + 1;
end

% skriv ut
i = 1;
while i <= limit
   if should_print(i)
      disp(i);
   end
   i = i + 1;
end
```

II

Övningar och facit



8 Övningar

8.1 Primtal, delbarhet och faktorisering

Följande övningar är lämpliga för Ma1c och är relaterade till följande centrala innehåll:

- ***Taluppfattning, aritmetik och algebra:*** *[...] begreppen primtal och delbarhet. (Ma1c)*

Övning (E), 8.1: Kontrollera faktorer (Ma1c)

Kim hävdar att om vi multiplicerar alla tal i listan `[3 5 7 17 23]` så blir produkten lika med 41055, och ber dig skriva ett program som kontrollräknar. Du föreslår att Kim helt enkelt skriver in `3*5*7*17*23` som ett kommando i Matlab/Octave, men Kim säger att han kommer vilja använda programmet på väldigt långa listor som redan är färdigskrivna, så han orkar inte skriva in så många multiplikationstecken. Skriv ett program åt Kim så att han lätt kan kontrollräkna sånt själv i fortsättningen. Programmets första två rader ska vara:

```
all_factors = [3 5 7 17 23]; % här kan Kim ändra
expected_product = 41055; % här också
```

Programmet ska sedan använda de två variablerna, och skriva ut ”Rätt” eller ”Fel”.

Övning (E), 8.2: Delbart med 3? (Ma1c)

Skriv ett program som låter användaren mata in ett heltal. Programmet ska testa om talet är delbart med 3 eller inte. Skriv ut ”delbart” eller ”ej delbart” beroende på vad det är.

Tips: bläddra tillbaka till exempel 1.11 och läs om funktionen `mod`.

Övning (M), 8.3: Primtal eller ej (Ma1c)

Skriv ett program som låter användaren mata in ett heltal. Kolla om talet är ett primtal. Skriv ut ”primtal” eller ”ej primtal” beroende på vad det är.

Tips: Utgå från förra övningen och bygg vidare på den.

Övning (M), 8.4: Faktorisera ett heltal (Ma1c)

Skriv ett program som låter användaren mata in ett heltal. Skriv ut alla talets primtalsfaktorer.

Tips: Utgå från förra övningen och bygg vidare på den.

Övning (M), 8.5: Testa att programmet stämmer (Ma1c)

Gör om ovanstående faktoriserings-program så att det lägger alla faktorerna i en lista. Använd sedan programmet från Övning 8.1 för att testa att ditt faktoriserings-program verkligen räknade rätt. Om programmen inte håller med varandra så är det fel i antingen det ena eller det andra programmet. Hitta i så fall felet och fixa det.

8.2 Sannolikhet och statistik

Följande övningar är lämpliga för Ma1c och Ma2c och är relaterade till följande centrala innehåll:

- ***Sannolikhet och statistik:*** *[...M]etoder för beräkning av sannolikheter vid slumpförsök i flera steg med exempel från spel[...]. (Ma1c)*
- ***Sannolikhet och statistik:*** *Statistiska metoder för rapportering av observationer och mätdata från undersökningar. (Ma2c)*
- ***Sannolikhet och statistik:*** *Metoder för beräkning av olika lägesmått och spridningsmått inklusive standardavvikelse, med digitala verktyg. (Ma2c)*
- ***Samband och förändring:*** *Konstruktion av grafer till funktioner [...] med digitala verktyg. (Ma2c)*

Programmering kan vara lämpligt för att räkna på och förstå sannolikhet och statistik. Matteuppgifter om sannolikhet har ofta en exakt lösning som vi kan få fram via någon färdig formel. Din vanliga mattebok är garanterat fylld av såna. Men vad gör du om du får en uppgift där du inte har någon färdig formel? Om du klurar tillräckligt länge så kanske du kan komma på en formel själv. Men det är ofta lättare att istället skriva ett program som gör en simulering, t.ex. kastar tärning jättemånga gånger, och sedan samlar ihop statistik om resultatet. Du får inte ett exakt resultat, men ett ungefärligt. Ju fler kast desto mer exakt.

Exempel: Om du kastar två tärningar (vanliga sexsidiga tärningar), vad är sannolikheten att summan blir 2? Att den blir 3? Och så vidare.

Vi skriver ett program som kastar en tärning 100 gånger, och ritar resultatet i ett histogram som vi lärde oss i kapitel 4:

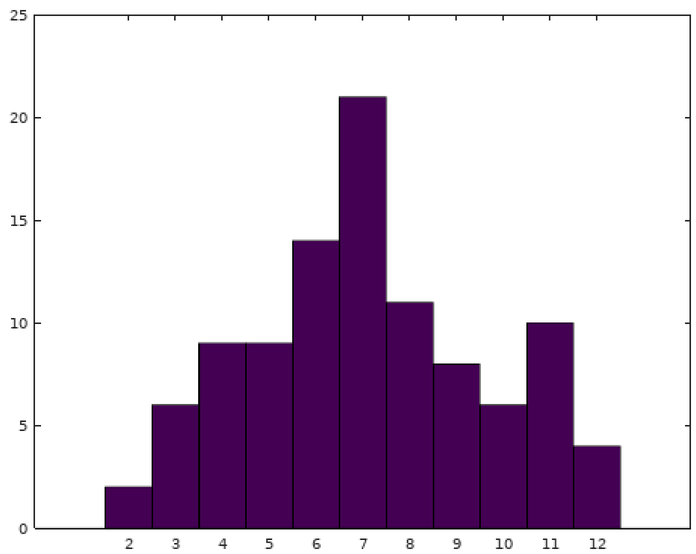
```
random_numbers = [];
number_throws = 100;
throw = 1;
while throw <= number_throws;
   dice1 = randi(6);
   random_numbers(end + 1) = dice1;
   throw = throw + 1;
end
hist(random_numbers,
   min(random_numbers):max(random_numbers));
```

Nu utökar vi programmet så att det kastar två tärningar åt gången, och mäter summan av dem:

```
random_numbers = [];
number_throws = 100; % den här kan man ändra
```

```
throw = 1;
while throw <= number_throws;
   dice1 = randi(6);
   dice2 = randi(6);
   sum = dice1 + dice2;
   random_numbers(end + 1) = sum;
   throw = throw + 1;
end
hist(random_numbers,
   min(random_numbers):max(random_numbers));
```

Figur 8.1: Histogram över 100 summor



Övning (E), 8.6: Funktionen randi (Ma1c, Ma2c)

Eftersom programmet använder slumptalsfunktionen `randi` så kommer resultatet att bli olika varje gång du kör programmet. Prova det några gånger!

Övning (E), 8.7: När blir datorn långsam? (Ma1c, Ma2c)

Datorn är såpass snabb att den kan simulera hundra kast utan problem. Prova att öka antalet kast till 1000, 10000 och så vidare och kolla när det börjar gå långsamt på din dator!

Övning (E), 8.8: Hur förändras formen? (Ma1c, Ma2c)

Hur förändras formen på histogrammet när du ökar antalet kast, från 100 till 1000 och 10000? Beskriv med ord.

Övning (E), 8.9: Skillnad mellan två tärningar (Ma1c, Ma2c)

Gör om programmet så att det inte längre mäter summan av de två tärningarna, utan istället mäter skillnaden mellan dem. Här måste du tänka på att vi inte är intresserade av negativa skillnader. Alltså, om det första tärningskastet ger 2 och det andra tärningskastet ger 5 så vill vi ändå tänka att skillnaden är 3, inte -3.

Tips: bläddra tillbaka till delkapitel 1.6.6 och läs om funktionen `abs`.

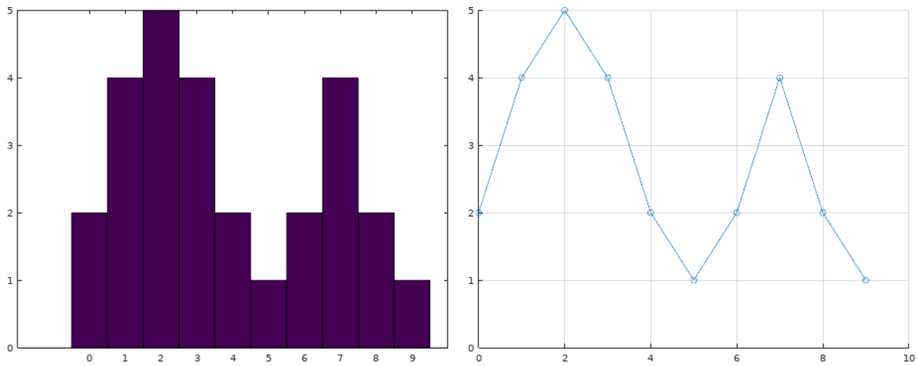
Övning (E), 8.10: Histogramanalys (Ma1c, Ma2c)

Titta i det nya histogrammet. Vad är den vanligaste skillnaden mellan två tärningar? Alltså, vad blev det oftast?

Övning (M), 8.11: Beräkna histogram själv (Ma1c, Ma2c)

Tänk om funktionen `hist` inte fanns, men du ändå väldigt gärna ville rita nån sorts histogram. Och tänk om funktionen `plot` fortfarande fanns. Du skulle kunna skriva egen kod som utgår från `random_numbers` och gör en sammanfattande beräkning och till slut använder `plot` för att rita sammanfattningen. Hur ser den koden ut? Observera att `plot(random_numbers)` inte är rätt svar. Din graf ska visa samma information som `hist`, förutom att den rent visuellt använder linjer istället för staplar.

Figur 8.2: Plot som motsvarar histogram



Övning (E), 8.12: Chans att slå Yatzy på ett slag (Ma1c)

Alex spelar spelet Yatzy. I det spelet får hon kasta fem tärningar. Beroende på vad hon slår så blir det olika poäng. Det som ger mest poäng är att alla fem tärningarna får samma tal - detta kallas också Yatzy, och antalet ögon på tärningarna spelar alltså ingen roll. Fem 1:or är lika mycket Yatzy som fem 6:or. Gör ett program som beräknar sannolikheten att slå Yatzy på ett slag.

Tips: Om du är osäker på var du ska börja, gå tillbaka till Övning 6.4, och hitta på ett sätt att testa om alla tärningarna visar samma tal eller inte. Kör sedan tusentals spelomgångar och håll räkningen på hur många av dem som blir Yatzy.

Övning (M), 8.13: Spara tärningar i Yatzy (Ma1c, Ma2c)

Alex fortsätter spela Yatzy (se föregående övning). Om hon inte är nöjd med sitt första försök, så får hon välja att låta valfritt antal tärningar ligga kvar på bordet, och plocka upp resten och slå om dem. Den lättaste strategin för att slå Yatzy är att spara de tärningar som har samma tal, och slå om resten.

Skriv ett program som analyserar en lista med fem tärningar, och skriver ut vilket värde som är smartast att spara nästa slag. Om det finns flera på delad förstaplats så spelar det ingen roll vilken av dem som programmet skriver ut. Testkör programmet med dessa listor:

- `[1 4 5 4 3]`. Det är smartast att spara 4:orna.
- `[1 4 5 4 1]`. Spara 4:orna eller 1:orna, vilket som.
- `[1 4 4 4 1]`. Spara 4:orna.
- `[1 4 5 6 3]`. Alla olika, spelar ingen roll vad som sparas.

Övning (S), 8.14: Chans att slå Yatzy (Ma1c, Ma2c)

Enligt de riktiga reglerna för Yatzy (jämför föregående övning) så får man *två* gånger välja ett valfritt antal tärningar och slå om dem. Alex vill veta vad sannolikheten för att slå Yatzy är. Hjälp henne att skriva ett program som ger henne svar på frågan genom att spela väldigt många (säg 10000, 100000 eller ännu fler) spel.

Exempel:

- Första kastet: Tärningarna visar 4 5 1 4 2
- Alex sparar de två 4:orna och kastar om de tre övriga.
- Andra kastet: De nykastade tärningarna visar 1 1 1. Totalt visar tärningarna alltså 4 1 1 4 1.
- Nu har ju Alex fler 1:or än 4:or, så hon sparar de tre 1:orna och kastar om resten.
- Tredje kastet: De nykastade tärningarna visar 5 3. Totalt visar tärningarna alltså 5 1 1 3 1.
- Alex har nu gjort tre kast och får inte göra fler. Alex fick tyvärr inte Yatzy.

Tips:

- *Börja med att skriva koden för att göra ett enda kast med fem tärningar (se Övning 8.12).*
- *Skriv sedan kod som räknar ut vilka tärningar som är bäst att spara (se Övning 8.13).*
- *Ändra sedan koden så att den upprepar ovanstående tre gånger. Detta är en spelomgång. Men kasta inte om alla tärningar, utan bara de som inte ska sparas.*
- *Kör tusentals spelomgångar och håll räkningen på hur många av dem som blir Yatzy.*

Övning (E), 8.15: Standardavvikelse (Ma2c)

Utgå från följande lista: `[4 9 10 7.5 8 9 3 9 4 -2]`

Skriv ett program som räknar ut listans medelvärde och standardavvikelse. Vi utgår från att du redan känner till vad det innebär. Använd iteration.

Övning (S), 8.16: Typvärde (Ma2c)

Utgå från följande lista: `[4 9 10 7.5 8 9 3 9 4 -2]`

Skriv ett program som räknar ut listans typvärde. Vi utgår från att du redan känner till vad det innebär. Använd iteration.

Tips: Om du tycker att övningen är svår, börja med att lösa Övning 8.13.

Övning (E), 8.17: Inbyggda funktioner för lägesmått och spridningsmått (Ma2c)

Matlab/Octave innehåller förstås färdiga funktioner för att räkna ut medelvärde, typvärde och standardavvikelse. Gör en ny version av ovanstående övningar. Sök information om de inbyggda funktionerna `mean`, `mode` och `std`, och använd dem istället för att göra beräkningen själv.

8.3 Numerisk lösning av linjära ekvationer

Följande övningar är lämpliga för Ma1c och är relaterade till följande centrala innehåll:

- ***Taluppfattning, aritmetik och algebra:*** *Algebraiska och grafiska metoder för att lösa linjära ekvationer och olikheter samt potensekvationer[...] (Ma1c)*

Övningarna lämpar sig även för Ma2c och Ma3c, då de är förkunskapskrav för att kunna lösa andragradsekvationer och för att kunna approximera talet e

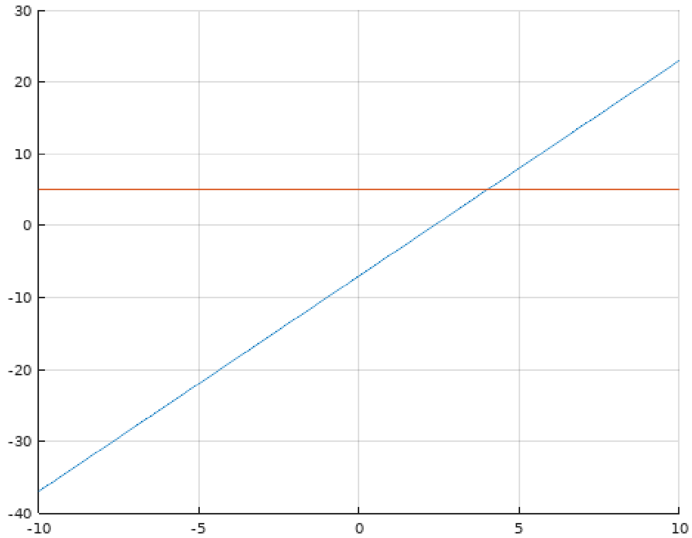
Grafer är bra för att utforska ekvationer. Om din vanliga mattebok t.ex. nämner ekvationen $3x - 7 = 5$ och ber dig att hitta vad x är, så ska du förstås kunna göra beräkningen för hand med penna och papper, men att även se ekvationen grafiskt kan vara ett komplement som ökar din förståelse, och ett sätt att dubbelkolla att du räknat ut rätt svar.

Vi kan betrakta de två sidorna i ekvationen som varsin linje. Vi kan rita båda linjerna tillsammans i samma graf, såhär:

```
hold on;
xmin = -10;
xmax = 10;
x = [xmin : xmax]
left_side = 3 .* x .- 7
plot(x, left_side); % det till vänster om likamedtecknet
plot([xmin xmax], [5 5]); % det till höger
```


Notera att vi använder `plot()` på två olika sätt, som vi lärde oss i delkapitel 4.1.3.

Figur 8.3: Ekvationen visualiserad som två linjer



Vad innebär nu egentligen ekvationen $3x - 7 = 5$? Den innebär att det finns något x som gör att vänstersidan och högersidan om likamedtecknet får samma värde. Grafiskt innebär det att de två linjerna någonstans korsar varandra. Titta på bilden! Det x-värde där linjerna korsar varandra är rätt svar på övningen. Alla de andra x-värdena i grafen gör att de två linjerna hamnar på olika höjd, så de är inte lösningar på övningen. Om du kommer fram till ett x när du räknar ut ekvationen på papper, och ett annat x när du tittar i grafen, så vet du att du har gjort fel någonstans.

När du tittar på bilden så använder du ögonen för att hitta det x-värde där linjerna möts. Men vi skulle kunna programmera datorn för att göra ungefär samma sak, utan ögon. Datorn kan snabbt testa jättemånga x-värden och hitta det värde där ekvationen stämmer. Detta kallas *numerisk lösning*.

Övning (E), 8.18: Enkel numerisk lösning (Ma1c)

Skriv ett program som går igenom alla heltal mellan -10 och 10, och testar om ekvationen stämmer. Med andra ord, börja med att låta x vara lika med -10, gör beräkningen $3x - 7$, och om resultatet blir lika med 5 så skriv ut värdet på x, annars öka x ett steg till -9 och upprepa. Programmet ska skriva ut svaret `4`.

Den enkla metoden för numerisk lösning har dock några svagheter:

Övning (E), 8.19: Enkel numerisk lösning, svaghet 1 (Ma1c)

Prova att ändra ekvationen till $3x - 7 = 29$. Ditt program kommer inte att hitta lösningen. Varför? Och vad är det lättaste sättet att fixa programmet så att det hittar lösningen?

Övning (E), 8.20: Enkel numerisk lösning, svaghet 2 (Ma1c)

Prova att ändra ekvationen till $3x - 7 = 4$. Ditt program kommer inte att hitta lösningen. Varför?

Övning (E), 8.21: Lös linjär ekvation med `solve` (Ma1c)

Det är förstås möjligt att skriva ett bättre program som inte har svagheterna i de två ovanstående övningarna. Men det finns redan en färdig, inbyggd funktion som inte har dessa två svagheter. Funktionen heter `solve` och finns i Matlab och i Octave Online, men inte i det nedladdade&installerade Octave (i skrivande stund). Den används såhär:

```
syms x;
solve(3*x-7==4, x)
```

`syms x` bestämmer att `x` är en okänd variabel.

Övning: Om du kör Matlab eller Octave Online, använd `solve` för att lösa ekvationen: $4x + 15 = -9$. Vad är x?

Använd sedan `solve` på några andra ekvationer från din vanliga mattebok.

Övning (S), 8.22: Klurig numerisk lösning med intervallhalvering (Ma1c)

Skriv ett eget program som kan lösa ekvationen $3x - 7 = 4$ och valfri annan ekvation av samma sort, och som inte har ovannämnda svaghet nummer två. (Använd inte den inbyggda `solve`.)

Det räcker inte att ta kortare steg, för hur korta steg ska vi ta för att vara säkra på att inte hoppa över svaret? Om vi tar steg med längden

0.1 så kommer vi ju att hoppa från x=3,6 till x=3,7 och därmed hoppa över svaret.

Tricket är att, istället för att bara leta efter *ett* x där vänsterledet blir exakt lika med 4, så kan vi utgå ifrån *två olika* x som ligger på varsin sida om rätt svar, och sedan ta oss närmare och närmare svaret med hjälp av samma metod som den optimala strategin för ”gissa talet”-spelet vi jobbade med i Övning 7.1.

Den här uppgiften är avsiktligt lite klurig. Börja med att fundera på problemet geometriskt med penna och papper, och se om du kan se samband mellan denna övning och hur du bäst löser gissa talet-spelet.

8.4 Andragradsekvationer

Följande övningar är lämpliga för Ma2c och Ma3c och är relaterade till följande centrala innehåll:

- ***Taluppfattning, aritmetik och algebra:*** *Algebraiska och grafiska metoder för att lösa exponential-, andragrads- och rotekvationer [...] (Ma2c)*
- ***Samband och förändring****: Konstruktion av grafer till funktioner [...] med digital verktyg. (Ma2c)*
- ***Samband och förändring****: Egenskaper hos andragradsfunktioner. (Ma2c)*
- ***Samband och förändring****: Algebraiska och grafiska metoder för lösning av extremvärdesproblem (Ma3c)*

Övning (E), 8.23: Lös andragradsekvation med `solve` (Ma2c)

Använd `solve` (som vi lärde oss i Övning 8.21) för att hitta de två rötterna till andragradsekvationen $x^2 + 5x - 6 = 0$. Vilka är rötterna?

Övning (E), 8.24: Lös andragradsekvation med `roots` (Ma2c)

`solve` är inte den enda inbyggda funktionen i Matlab/Octave för att lösa ekvationer. Det finns en annan som heter `roots` och som bara fungerar på en viss typ av ekvationer som kallas *polynomekvationer*. Följande är exempel på polynomekvationer:

- $x = 5$
- $3x + 7 = 0$
- $x^2 - 2x - 12 = 0$

- $4x^2 + 9 = 0$
- $2x^3 + 6x^2 + 13x - 8 = 0$

Följande är *inte* polynomekvationer och går alltså inte att lösa med `root`:

- $7/x = 0$
- $3^x = 0$
- $sin(x) = 0$

För att lösa en polynomekvation ser vi först till att den har rätt form, annars slår vi ihop och flyttar över termer tills den har rätt form. Sen tar vi bort alla x och x^2, och gör en lista av det som blir kvar på vänstersidan. Generellt så blir ekvationen $ax^2 + bx + c = 0$ listan `[a b c]`. T.ex ekvationen $3x^2 = 4$ gör vi först om till $3x^2 + 0x - 4 = 0$ och sedan till listan `[3 0 -4]`. Slutligen ger vi denna lista som ett argument till `roots` och skriver ut resultatet:

```
disp(roots([3 0 -4]));
```

Övning: lös så många som möjligt av dessa ekvationer med hjälp av `roots`:

- $6x^2 - 13x + 5 = 0$
- $4x^2 - 2x - x/x = 5$
- $2/x - 18 = 0$
- $2x^2/2 + 2x^2 - x - x = 0$

Övning (S), 8.25: Klurig numerisk lösning, del 2 (Ma2c)

Ekvationen i Övning 8.22 är nog ärligt talat lättare att lösa för hand än genom att skriva ett klurigt datorprogram. Men när du klarat den övningen, testa att använda samma program för att lösa den mycket mer komplexa ekvationen $x^6 - sin(x) - 3^x + 7 = 0$.

Vilket värde har x? Starta sökningen i intervallet $-50 <= x <= 50$. Om du har lyckats göra programmet tillräckligt generellt så ska det fungera!

Hade du kunnat lösa ut x för hand i denna ekvation?

Övning (S), 8.26: Klurig numerisk lösning, del 3 (Ma2c)

Testa att använda samma program för att lösa ekvationen $x^2 + 6x + 9 = 0$. Fungerar det? Om inte, varför inte?

Övning (M), 8.27: Andragradsfunktionens minsta värde, grafiskt (Ma2c, Ma3c)

Diogenes har fått i uppdrag att tillverka en liten tunna som ska rymma 6.2832 dm^3 vätska. Tunnan ska ha formen av en cylinder. Han tänker göra den av en rektangular plåtbit som rullas ihop till en cylinder, plus två cirkulära plåtbitar som täpper till ändarna på cylindern. Beroende på hur långsmal han gör den så kommer det gå åt olika mycket plåt. Uppdragsgivaren bad Diogenes att använda så lite plåt som möjligt.

Fråga: Vilken radie och höjd ska Diogenes välja på sin tunna?

Eftersom vi vet att volymen ska vara 6.2832 dm^3 så kan radien och höjden inte variera oberoende av varann. Om Diogenes väljer en radie i dm, så ges höjden h automatiskt av:

$h = 6.2832/(\pi * radie^2)$

Ekvationen för hur mycket plåt som behövs är i dm^2:

$area = 2 * \pi * radie * (radie + h)$

Problemet går att lösa analytiskt, men vi ska lösa det genom att titta i en graf. Skapa en lista med ett antal tänkbara radier, till exempel från 0 till 10. Använd sedan ovanstående ekvationer för att beräkna en lista med höjder, och en lista med areor. Använd slutligen funktionen `plot()` för att grafiskt se vilken radie som ger den minsta arean. Du kan behöva ändra på din lista med radie-förslag flera gånger för att ”zooma in” på rätt del av grafen.

8.5 Potensfunktioner och exponentialfunktioner

Följande övningar är lämpliga för Ma1c och Ma2c och är relaterade till följande centrala innehåll:

- ***Samband och förändring:*** *Begreppen förändringsfaktor och index. Metoder för beräkning av räntor och amorteringar för olika typer av lån [...] (Ma1c)*
- ***Samband och förändring:*** *egenskaper hos [...] exponentialfunktioner. (Ma1c)*
- ***Samband och förändring:*** *Konstruktion av grafer till funktioner [...] med digital verktyg. (Ma2c)*
- ***Taluppfattning, aritmetik och algebra:*** *Algebraiska och grafiska metoder för att lösa [...] exponentialekvationer (Ma2c)*
- ***Sannolikhet och statistik:*** *Metoder för beräkning av olika lägesmått och spridningsmått inklusive standardavvikelse [...]. (Ma2c)*

Övning (E), 8.28: Exponentialfunktion (Ma1c, Ma2c)

En nystartad sommarfestival släpper 1 000 biljetter som blir slutsålda direkt. Arrangörerna vill att festivalen ska växa i en lagom takt, så de bestämmer sig för att släppa 10% fler biljetter varje år. Hur många biljetter släpps till den 21:a festivalen, när det alltså gått 20 år sedan starten? Använd iteration! Som bonus kan du även plotta en graf över antalet biljetter per år.

Övning (E), 8.29: Numerisk lösning av exponentialekvation (Ma1c, Ma2c)

Utgå från föregående övning. Vilken festival i ordningen blir den första att släppa 5 000 eller fler biljetter? Programmet ska skriva ut svaret. Använd iteration! Tänk på att festival nummer 2 inträffar 1 år efter starten, festival nummer 3 inträffar 2 år efter starten, och så vidare.

Övning (M), 8.30: Oregelbunden exponentialfunktion (Ma1c, Ma2c)

Det finns olika sätt att lösa ovanstående två matteproblem. Ett sätt är med algebraisk/analytisk matematik, genom att lösa ut relevanta variabler ur ekvationen $y = C * a^x$. Det numeriska sättet är att upprepa en uträkning i en loop, där varje varv i loopen t.ex. representerar att det gått ett år.

Det finns för- och nackdelar med de olika sätten. I de ovanstående övningarna vore nog faktiskt det analytiska sättet mycket enklare än det numeriska. Men ju mer ”oregelbundet” ett matematiskt problem är, desto svårare brukar det vara att lösa analytiskt, och desto lämpligare blir då det numeriska. T.ex. om ökningstakten i en exponentialfunktion inte är samma procent varje år, utan varierar.

En annan årlig festival bestämmer sig för att busa med matematiker och ändrar biljettantalet enligt följande sicksack-mönster:

Festival nummer	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	
Ökning sen förra året	4%	2%	5%	3%	6%	4%	7%	5%	osv...

Övning: Den första festivalen har 100 biljetter. Hur många biljetter har den 26:e festivalen, när det alltså gått 25 år sedan start?

Hade du kunnat lösa denna övning analytiskt? Hade du orkat göra det numeriskt utan programmering?

Övning (S), 8.31: Oregelbunden exponentialfunktion, del 2 + slumpförsök och statistik (Ma1c, Ma2c)

Det är förstås inte så vanligt i verkligheten att ökningstakten i en exponentialfunktion går upp och ner i ett exakt sicksack-mönster. Det är betydligt vanligare att vi inte vet i förväg exakt hur stor ökningstakten kommer bli, men att vi vill göra någon sorts kvalificerad gissning ändå.

Till exempel: En population på 20 kaniner har utplanterats i ett område där det inte finns några rovdjur, men en del andra faror. Vi utgår från att antalet kaniner ökar med mellan 10% och 28% per månad.

Fråga: Ungefär hur många kaniner finns det efter 24 månader?

Tips: Vi kan välja ett slumpmässigt tal mellan 10% och 28% och låtsas att det talet är ökningstakten just den månaden. Vi kan välja ett nytt slumptal nästa månad, och så vidare, upp till 24 månader. Detta ger oss en gissning, ett tänkbart framtidsscenario för hur många kaniner som skulle kunna finnas om 24 månader. Skriv ett program som skapar en sådan gissning!

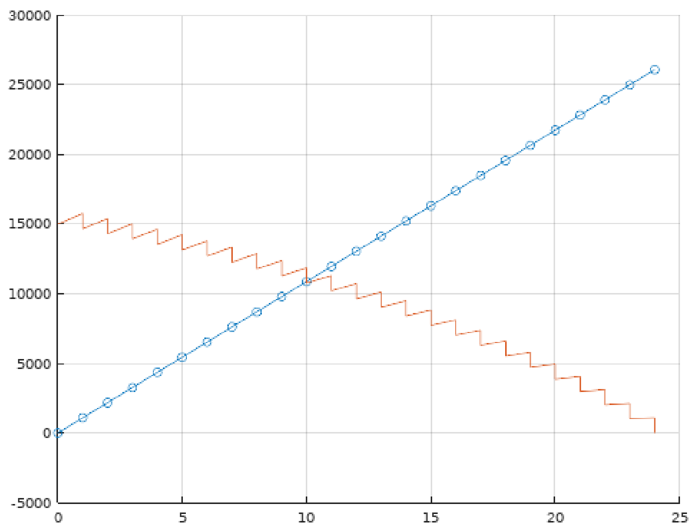
Utöka sedan programmet så att det gör om hela beräkningen 100 gånger. Du får då 100 olika gissningar för hur stor kaninpopulationen kommer vara efter 24 månader. Beräkna medelvärdet av alla dessa gissningar. Som bonus kan du även välja att plotta alla 100 scenarion i en och samma graf.

Denna metod brukar kallas en Monte Carlo-simulering, efter det berömda casinot i Monaco.

Övning (S), 8.32: Avbetalning av annuitetslån (Ma1c, Ma2c)

Gizem lånar 15 000 kr. Det är ett annuitetslån på 24 månader, vilket innebär att hon betalar tillbaka samma summa varje månad: 1087.07 kr. Men den delen av lånet som hon ännu inte betalat tillbaka, växer med månadsräntan 5%. Följande graf illustrerar lånet.

Figur 8.4



Den stigande blå linjen visar hur mycket pengar Gizem har betalat totalt efter varje månad. Den fallande orange trapp-liknande linjen visar hur den återstående skulden förändras. Notera att den går både uppåt och nedåt - upp pga räntan, ner pga återbetalningen. Skriv koden för att skapa samma graf själv!

Tips: Rita linjerna med metoden som vi lärde oss i delkapitel 4.1.3. Den orange linjen behöver två värden för varje månad, inte bara ett värde.

8.6 Derivata

Följande övningar är lämpliga för Ma3c och är relaterade till följande centrala innehåll:

- ***Samband och förändring:*** *Orientering när det gäller kontinuerlig och diskret funktion samt begreppet gränsvärde. (Ma3c)*
- ***Samband och förändring:*** *Begreppen [...] ändringskvot och derivata för en funktion. (Ma3c)*
- ***Samband och förändring:*** *Algebraiska och grafiska metoder för bestämning av derivatans värde för en funktion. (Ma3c)*
- ***Samband och förändring:*** *Introduktion av talet e och dess egenskaper. (Ma3c)*

Övning (E), 8.33: Förändringshastighet enligt tabell (Ma3c)

Koldioxidhalten i jordens atmosfär ökar varje år. Följande tabell innehåller årliga medelvärden av koldioxidhalten i PPM (parts per million), uppmätta på berget Mauna Loa på Hawaii:

```
x = [2007 2008 2009 2010 2011 2012 2013 2014 2015
   2016 2017];
y = [383.79 385.60 387.43 389.90 391.65 393.85 396.52
   398.65 400.83 404.21 406.53];
% källa:
% https://www.esrl.noaa.gov/gmd/ccgg/trends/data.html
```

Fråga: Hur snabbt ökade koldioxidhalten år 2016? Svaret ska ha enheten PPM/år.

Skriv ett program som approximerar derivatan i punkten $x = 2016$ med hjälp av ”central ändringskvot” - mät med andra ord den genomsnittliga ökningen från 2015 till 2017.

Övning (E), 8.34: Förändringshastighet enligt tabell, del 2 (Ma3c)

Utgå från föregående övning. Förbättra programmet så att det kan approximera derivatan för valfritt år. När programmet körs får användaren mata in det årtal de är intresserade av (alltså med funktionen `input`) mellan 2008 och 2016.

Tips: Kom ihåg funktionen `find` *som vi gick igenom i delkapitel 3.6.*

Övning (E), 8.35: Ändringskvot för funktion (Ma3c)

Utgå från föregående övning, men byt från en tabell till funktionen $f(x) = (x^2 + 1)/x$

Till skillnad från ovanstående tabell, där vi jobbade med hela årtal, så är det nu inte självklart hur långt framåt och bakåt vi ska gå för att välja ”grannar” till x. För denna övning, välj avståndet $h = 0.4$ till grannarna! Använd programmet för att approximera derivatan för $x = 1$, så grannarna blir alltså 0.6 och 1.4.

Övning (M), 8.36: Testa olika värden för h (Ma3c)

Bygg vidare på föregående program och låt datorn testa en massa olika värden för h. Börja med $h = 0.4$, beräkna ändringskvoten, halvera h, beräkna ändringskvoten igen, och så vidare. Upprepa 8 gånger. Plotta en graf med ändringskvoten på y-axeln, och antalet halveringar på x-axeln. Baserat på grafen, vad tror du att den exakta derivatan är?

Övning (E), 8.37: Testa den inbyggda `gradient` (Ma3c)

Det finns (som vanligt) inbyggda funktioner för att mäta ändringskvoten. I Matlab och Octave Online kan vi använda följande *symbolhanterande* metod:

```
1 syms x;
2 xp = 1; % i vilken punkt vill vi veta lutningen
3 y = (x^2 + 1) / x; % vår funktion
4 g = gradient(y);
5 estimate = subs(g, xp);
6 disp(estimate);
```

Det smidiga med ovanstående metod är att den ger ett exakt svar, och vi behöver inte bry oss om att välja något h. För att testa en annan punkt eller funktion behöver du bara ändra på rad 2 och 3.

Men `syms` ingår som sagt inte i Octave (såvida vi inte installerat ett tilläggspaket). Utan tilläggspaketet kan vi använda följande *numeriska* metod som ger ett ungefärligt svar och kräver att vi väljer ett h själva:

```
xp = 1; % i vilken punkt vill vi veta lutningen
h = 0.001;
```



```
3 x = [xp - h : h : xp + h];
4 y = (x.^2 + 1) ./ x; % vår funktion
5 estimate = gradient(y, h)(2);
6 disp(estimate);
```

För att testa en annan punkt eller funktion behöver du bara ändra på rad 1 och 4.

Använd nu någon av ovanstående metoder för att beräkna $f'(1)$ för funktionen $f(x) = 3\sqrt{x} + 2/x^2$

Övning (S), 8.38: Approximera talet e (Ma3c)

Finns det något tal e som uppfyller att derivatan av e^x är lika med e^x för alla x? Ja, det finns det. Skriv ett program som hittar (ett närmevärde till) det talet genom att prova sig fram.

Du behöver ett sätt för datorn att prova sig fram och närma sig svaret, så gör först Övning 7.2, och sedan Övning 8.22.

Sen behöver du ett sätt att mäta ändringskvoten i en viss punkt, så utgå från koden du skrev i Övning 8.35. Kombinera allt du lärt dig i de övningarna, så kan du nog lösa även denna!

Observera att du inte behöver testa för alla x, utan bara *något* x, och det spelar ingen roll vilket x du väljer, för resultatet blir ändå detsamma. Använd förslagsvis $x = 1$, så blir koden lite enklare - du kan då söka efter ett e där ändringskvoten för e^x kring punkten $x = 1$ är lika med $e^1 = e$. Och vilket intervall ska du söka i? Tips: e är större än 0, så du behöver inte söka bland negativa tal.

9

Facit och lösningsförslag

Här följer facit till denna bok. I många av övningarna i denna bok ska du skriva kod. Den kod som presenteras här ska ses som lösningsförslag, snarare än ett absolut facit. Med programmering går det ju att lösa ett problem på många olika sätt.

Facit till kapitel 1 - Datorn som miniräknare

Övningarna i kapitlet behöver inget facit.

Facit till kapitel 2 - Variabler

Övning 2.1 - Höger eller vänster?

Efter att ha kört de tre kodraderna så är `a=2` och `b=2`. Alltså, när det står två variabler på varsin sida om tilldelningsoperatorn så ändras den på vänstra sidan.

Övning 2.2 - Kopplas variabler ihop för all framtid?

På rad 2 så får `b` det dåvarande värdet av `a`, alltså blir `b=1`. Efter att ha kört de tre kodraderna så är `b=1` fortfarande. Det är *inte* så att `b` ”kopplas ihop” med `a` för all framtid. När vi på rad 3 ändrar värdet på `a` så ändras alltså *inte* värdet på `b`.

Övning 2.3 - Summan och medelvärdet av tre tal

```
a = 23;
b = 45;
c = 67;
disp(a + b + c);
```

```
disp((a + b + c) / 3);
```

Övning 2.4 - Decimaltal till heltal

```
a = 11.534;
disp(round(a));
```

Facit till kapitel 3 - Listor

Övning 3.1 - Olika sätt att skapa samma lista

```
% omständigt sätt 1:
a = [];
a(1) = 20;
a(2) = 30;
a(3) = 40;
a(4) = 50;
a(5) = 60;
a(6) = 70;
a(7) = 80;
a(8) = 90;

% omständigt sätt 2:
a = [];
a(end + 1) = 20;
a(end + 1) = a(end) + 10;
a(end + 1) = a(end) + 10;
a(end + 1) = a(end) + 10;
a(end + 1) = a(end) + 10;
a(end + 1) = a(end) + 10;
a(end + 1) = a(end) + 10;
a(end + 1) = a(end) + 10;

% kortare sätt:
a = [20 30 40 50 60 70 80 90];

% ännu bättre sätt:
a = [20 : 10 : 90];
```

Övning 3.2 - `disp` med lista

Vi får följande utskrift:

```
0
6
```

Facit till kapitel 4 - Grafer och diagram

Övning 4.1 - Återskapa kod efter bild

```
hold on;
plot([-3 3], [4 5], '-o');
plot([-3 : 3], [6 7 5 6 5 4 3], '-o');
```

Övning 4.2 - Plotta given funktion

```
x = [-20 : 0.1 : 20];
y = sin(x) ./ x;
plot(x, y);
```

Notera att om vi bara använder `x = [-20 : 20]` så blir grafen fult kantig. Därför tar vi kortare steg med steglängden `0.1`.

Facit till kapitel 5 - Selektion (med if)

Övning 5.1 - input

Jag testar att skriva in mitt namn som variabeln `b`:

```
Ange variabeln a: 12
Ange variabeln b: Emil
error: 'Emil' undefined near line 1 column 1
```

Matlab/Octave säger att det inte existerar någon variabel som heter `Emil`. Du kan prova att först skapa en variabel och sen skriva in dess namn så kommer det att funka.

Övning 5.2 - Kontrollera vädret (fortsättning)

```
j = 1;
n = 0;
svaret = input('Är det fint väder? ');
if svaret == j
    disp('Vi går på picknick!');
end
if svaret == n
    disp('Vi stannar inne och läser en bok');
end
```

Övning 5.3 - Var är det kallast?

```
ostersund_temp = input('Ange temperaturen i Östersund: ');
goteborg_temp = input('Ange temperaturen i Göteborg: ');
if ostersund_temp < goteborg_temp
    disp('Det är kallast i Östersund');
end
if goteborg_temp < ostersund_temp
    disp('Det är kallast i Göteborg');
end
if goteborg_temp == ostersund_temp
    disp('Det är lika kallt');
end
```

Övning 5.4 - Felaktig if-sats

Felet är att vi har råkat använda tilldelningsoperatorn = istället för jämförelse-operatorn ==. Vi ville förstås egentligen skriva såhär:

```
x = 9;
if x == 10
    disp('den är 10!');
end
```

Facit till kapitel 6 - Iteration (med while)

Övning 6.1 - Tal mellan 1 och 20

```
tal = 1;
while tal <= 20
   disp(tal);
   tal = tal + 1;
end
```

Övning 6.2 - Tal mellan 1 och 100

```
tal = input('Ange tal: ');
```



```
while tal <= 100
   disp(tal);
   tal = tal + 1;
end
```

Övning 6.3 - Singla slant

```
antal_singlingar = input('Ange hur många gånger du vill
   singla slant: ');
i = 1;
while i <= antal_singlingar
   if randi(2) == 1
      disp('Krona');
   else
      disp('Klave');
   end
   i = i + 1;
end
```

Övning 6.4 - Yatzy

```
i = 1;
while i <= 5
   disp(randi(6));
   i = i + 1;
end
```

Övning 6.5 - Väderstationen

```
num_temperatures = input('Ange antal mätningar: ');
% skapa tom lista som kommer fyllas på med temperaturer:
all_temperatures = [];
i = 1; % räknare för antal iterationer
total = 0; % används för att räkna ut medelvärdet
while i <= num_temperatures
   temperature = input('Ange temperaturmätning: ');
   % lägg till mätningen i slutet av listan:
   all_temperatures(end + 1) = temperature;
   total = total + temperature;
   i = i + 1;
end
disp(all_temperatures);
% räkna ut medelvärdet:
average = total / num_temperatures;
disp(average);
```

Övning 6.6 - Multiplikationstabellen

```
i = 1; % det ena som ska gångras med...
while i <= 10
   j = 1; % ... det andra
   while j <= 10
      disp(i * j);
      j = j + 1;
   end
   i = i + 1;
   disp('---'); % separator
end
```

Facit till kapitel 7 - Problemlösning

Övning 7.1 - Gissa talet

```
answer = randi(100); % slumpa ett tal mellan 1 och 100

nr_guesses = 1;
guess = input('Gissa ett tal mellan 1 och 100: ');
while guess ~= answer
    if guess < answer
        guess = input('Fel! Mitt tal är högre. Gissa igen: ');
    end
    if guess > answer
        guess = input('Fel! Mitt tal är lägre. Gissa igen: ');
    end
    nr_guesses = nr_guesses + 1;
end
disp('Rätt! Såhär många gissningar behövde du:');
```



```
disp(nr_guesses);
```

Övning 7.2 - Datorn gissar talet

```
r = 1; % rätt
l = 2; % lägre
h = 3; % högre
user_input = 0; % ska hantera r, l eller h
min = 1;
max = 100;

nr_guesses=0;

disp('===============================================');

while user_input ~= r
   % vi avrundar nedåt för att bara jobba med heltal:
    guess = floor((max+min)/2);
    disp('Jag gissar på:');
    disp(guess);
    user_input = input('Är det [r]ätt? Eller är ditt tal [h]ögre eller [l]ägre? ');
    if user_input == h
        min = guess;
    end
    if user_input == l
        max = guess;
    end
   nr_guesses = nr_guesses + 1;
end

disp('Såhär många gissningar behövde jag:');
disp(nr_guesses);
```

Facit till kapitel 8 - Övningar

Övning 8.1 - Kontrollera faktorer (Ma1c)

```
all_factors = [3 5 7 17 23]; % här kan Kim ändra
expected_product = 41055; % här också
product = 1;
i = 1;
while i <= size(all_factors, 2)
   product = product * all_factors(i);
   i = i + 1;
end
if product == expected_product
   disp('Rätt');
else
   disp('Fel');
end
```


Övning 8.2 - Delbart med 3? (Ma1c)

```
the_number = input('Skriv ett heltal: ');
% blir det ingen rest om vi dividerar talet med 3?
if mod(the_number, 3) == 0
   disp('delbart');
else
   disp('ej delbart');
end
```

Övning 8.3 - Primtal eller ej (Ma1c)

```
the_number = input('Skriv ett heltal: ');
is_prime = 1;
factor = 2;
while factor <= sqrt(the_number)
   % blir det ingen rest om vi dividerar?
   % dvs, är den jämt delbar?
   % dvs, är "factor" en faktor i "the_number"?
   if mod(the_number, factor) == 0
      is_prime = 0;
   end
   factor += 1;
end
if is_prime
   disp('primtal');
else
   disp('ej primtal');
end
```

Övning 8.4 - Faktorisera ett heltal (Ma1c)

```
remaining = input('Skriv ett heltal: ');
factor = 2;
while remaining > 1
   % blir det ingen rest om vi dividerar?
   % dvs, är den jämt delbar?
   % dvs, är "factor" en faktor i "remaining"?
   if mod(remaining, factor) == 0
      % vi har hittat en faktor, skriv ut den
      disp(factor);
      % efter denna faktor, vad blir kvar av talet?
      remaining = remaining / factor;
   else
      factor = factor + 1;
   end
end
```

Övning 8.5 - Testa att programmet stämmer (Ma1c)

```
the_number = input('Skriv ett heltal: ');
% faktorisera:
remaining = the_number;
factor = 2;
all_factors = [];
while remaining > 1
   if mod(remaining, factor) == 0
      disp(factor);
      all_factors(end + 1) = factor;
      remaining = remaining / factor;
   else
      factor = factor + 1;
   end
end
% kontrollräkna:
product = 1;
i = 1;
while i <= size(all_factors, 2)
   product = product * all_factors(i);
   i = i + 1;
end
if product == the_number
   disp('Rätt');
else
   disp('Fel');
end
```

Övning 8.6 - Funktionen randi (Ma1c, Ma2c)

Övningen behöver inget facit.

Övning 8.7 - När blir datorn långsam? (Ma1c, Ma2c)

Övningen behöver inget facit.

Övning 8.8 - Hur förändras formen? (Ma1c, Ma2c)

När vi bara gör 100 kast så blir histogrammet ofta ganska ojämnt och ”taggigt”. Men ju fler kast, desto mer antar histogrammet formen av en likbent triangel.

Övning 8.9 - Skillnad mellan två tärningar (Ma1c, Ma2c)

```
random_numbers = [];
number_throws = 100; % den här kan man ändra
throw = 1;
while throw <= number_throws;
   dice1 = randi(6);
   dice2 = randi(6);
   difference = abs(dice1 - dice2);
   random_numbers(end + 1) = difference;
```



```
    throw = throw + 1;
end
hist(random_numbers,
   min(random_numbers):max(random_numbers));
```

Övning 8.10 - Histogramanalys (Ma1c, Ma2c)

Den vanligaste skillnaden mellan två tärningar är 1.

Övning 8.11 - Beräkna histogram själv (Ma1c, Ma2c)

```
% ... detta är fortsättning på koden i övningen ovan
% med andra ord funkar denna kod inte för sig själv
low = min(random_numbers);
high = max(random_numbers);
num_points_in_histogram = 1 + high - low;
summary = zeros(1, num_points_in_histogram);
i = 1;
while i <= size(random_numbers, 2)
    one_number = random_numbers(i);
    index_in_summary = one_number + 1 - low;
    summary(index_in_summary) = summary(index_in_summary)
   + 1;
    i = i + 1;
end
xaxis = [low : high];
plot(xaxis, summary, '-o');
% gör att y-axeln alltid börjar på 0,
% men välj automatiskt var den slutar:
ylim([0 inf]);
```

Läs mer om `ylim` här: *http://se.mathworks.com/help/matlab/creating_plots/change-axis-limits-of-graph.html*

Övning 8.12 - Chans att slå Yatzy på ett slag (Ma1c)

Rätt svar är ca 0,077% men det krävs väldigt många spelomgångar för att få ett bra närmevärde.

```
total_nr_yatzy = 0; % hur många gånger vi fått yatzy
nr_games = 100000; % hur många spel vi vill köra
game = 1;
while game <= nr_games

  % slå en tärning:
  first = randi(6);
  % slå fyra tärningar till, se om alla blir samma:
```



```
  all_are_same = 1;
  die = 2; % för vi har redan slagit tärning 1
  while die <=5
    if randi(6) ~= first
      all_are_same = 0;
    end
    die = die + 1; % gå vidare till nästa tärning
  end

  if all_are_same == 1 % blev det yatzy?
    total_nr_yatzy = total_nr_yatzy + 1;
  end
  game = game + 1; % för att gå vidare till nästa spel
end

% skriv ut vårt närmevärde till sannolikheten för yatzy,
% i procent:
disp(100 * total_nr_yatzy / nr_games);
```

Övning 8.13 - Spara tärningar i Yatzy (Ma1c, Ma2c)

```
dices = [1 4 5 4 3]; % lista med våra fem tärningar

save_eyes = 0; % vilket tärningsvärde vi slog flest av
nr_save = 0; % antal tärningar med det värdet

% iterera sex gånger, en för varje tänkbart tärningsvärde
eyes = 1;
while eyes <= 6

  % räkna hur många tärningar som visar just detta värde
  counter = 0;
  die = 1;
  while die <= 5
    if dices(die) == eyes
        counter = counter + 1;
    end
    die = die + 1;
  end

  % ska vi byta vilket tärningsvärde vi ska spara?
  if nr_save < counter
    nr_save = counter;
    save_eyes = eyes;
  end

  eyes = eyes + 1;
end
disp('Bäst att spara alla tärningar med antal ögon:');
disp(save_eyes);
```


Alternativ lösning:

```
dices = [1 4 5 4 3]; % lista med våra fem tärningar
dice_histogram = zeros(1, 6);
die = 1;
while die <= 5
   eyes = dices(die);
   dice_histogram(eyes) = dice_histogram(eyes) + 1;
   die = die + 1;
end
% dice_histogram(1) är hur många 1:or vi slog
% dice_histogram(2) är hur många 2:or vi slog, osv
nr_save = 0;
eyes = 1;
while eyes <= 6
   if nr_save < dice_histogram(eyes)
      nr_save = dice_histogram(eyes);
      save_eyes = eyes;
   end
   eyes = eyes + 1;
end
disp(save_eyes);
```

(Se nästa sida för ytterligare en alternativ lösning)

Alternativ lösning för den som vill lära sig mer om hur man kan använda `hist`, `max` och `find`:

```
dices = [1 4 5 4 3]; % lista med våra fem tärningar
dice_histogram = hist(dices, 1:6);
save_eyes = find(dice_histogram == max(dice_histogram),
   1);
disp(save_eyes);
```

Ännu kortare lösning:

```
dices = [1 4 5 4 3]; % lista med våra fem tärningar
save_eyes = mode(dices);
disp(save_eyes);
```

För mer information om `mode`, se: *https://se.mathworks.com/help/matlab/ref/mode.html*:

Övning 8.14 - Chans att slå Yatzy (Ma1c, Ma2c)

Ungefär 4.6% chans.

```
dices = [0 0 0 0 0]; % lista med våra fem tärningar
total_nr_yatzy = 0; % hur många gånger vi fått yatzy

nr_games = 10000; % hur många spel vi vill köra
game = 1;
while game <= nr_games
  % save_eyes håller reda på vilket tärningsvärde vi har
  % flest av. Här sätter vi den till -1 för vi ska behöva
  % slå om alla tärningar första gången i ett nytt spel:
  save_eyes = -1;

  % roll är en räknare som går från 1 till 3.
  % vi itererar alltså tre gånger, en gång för varje slag:
  roll = 1;
  while roll <= 3

    % iterera fem tärningar, slå om några eller alla:
    die = 1;
    while die <= 5
      % om vi inte vill spara tärningen, slå om den:
      if dices(die) ~= save_eyes
        dices(die) = randi(6);
      end
      die = die + 1; % gå vidare till nästa tärning
    end
```



```

    % räkna ut vilket tärningsvärde som vi slog flest av.
    % det är de tärningarna som vi vill spara nästa slag.

    save_eyes = 0; % vilket tärningsvärde vi slog flest av
    nr_save = 0; % antal tärningar med det värdet

    % iterera 6 gånger, ett varv för varje
    % tänkbart tärningsvärde
    eyes = 1;
    while eyes <= 6

      % räkna hur många tärningar som visar just detta
      % värde
      counter = 0;
      die = 1;
      while die <= 5
        if dices(die) == eyes
          counter = counter + 1;
        end
        die = die + 1;
      end

      % ska vi byta vilket tärningsvärde vi ska spara?
      if counter > nr_save
        nr_save = counter;
        save_eyes = eyes;
      end

      eyes = eyes + 1;
    end

    roll = roll + 1; % för att gå vidare till nästa slag
  end
  if nr_save == 5 % blev det yatzy?
    total_nr_yatzy = total_nr_yatzy + 1;
  end
  game = game + 1; % för att gå vidare till nästa spel
end

% skriv ut vårt närmevärde till sannolikheten för yatzy,
% i procent
disp(100 * total_nr_yatzy / nr_games);
```


Övning 8.15 - Standardavvikelse (Ma2c)

Medelvärde: 6.1500, standardavvikelse: 3.8010

```
my_list = [4 9 10 7.5 8 9 3 9 4 -2];
nr_items = size(my_list, 2);

% räkna ut summan för alla tal i listan:
total = 0;
i = 1;
while i <= nr_items
    total = total + my_list(i);
    i = i + 1;
end

% räkna ut medelvärdet:
mid = total / nr_items;

% räkna ut standardavvikelse:
total = 0; % återställ total till 0
i = 1;
while i <= nr_items
    total = total + (my_list(i) - mid)^2;
    i = i + 1;
end

std_dev = sqrt(total/ (nr_items-1));
% nr_items-1 pga Bessels korrektion

disp(mid);
disp(std_dev);
```

Övning 8.16 - Typvärde (Ma2c)

Typvärde: 9

```
my_list = [4 9 10 7.5 8 9 3 9 4 -2];
nr_items = size(my_list, 2);

nr_most_common = 0; % antal element med typvärdet

i = 1;
while i <= nr_items

   % räkna antal element med "det här" värdet:
   counter = 0;
   j = 1;
   while j <= nr_items
      if my_list(j) == my_list(i)
         counter = counter + 1;
      end
```



```
        j = j + 1;
    end

    % ska vi ersätta det förra typvärdet?
    if nr_most_common < counter
        nr_most_common = counter;
        most_common = my_list(i);
    end

    i = i + 1;
end
disp('Typvärde: ');
disp(most_common);
```

Övning 8.17 - Inbyggda funktioner för lägesmått och spridningsmått (Ma2c)

```
my_list = [4 9 10 7.5 8 9 3 9 4 -2];
disp(mean(my_list)); % medelvärde
disp(mode(my_list)); % typvärde
disp(std(my_list)); % standardavvikelse
```

Övning 8.18 - Enkel numerisk lösning (Ma1c)

```
1 x = -10;
2 while x <= 10
3     if 3 * x - 7 == 5
4         disp(x);
5     end
6     x = x + 1;
7 end
```

Övning 8.19 - Enkel numerisk lösning, svaghet 1 (Ma1c)

Vi ändrar på rad 3 i föregående kodstycke till `if 3 .* x .- 7 == 29`. Programmet misslyckas för att lösningen då är 12, vilket ligger utanför intervallet -10 till 10 som ju är de värden vi testar. Den lättaste fixen är att testa ett större intervall, till exempel -1000 till 1000.

Övning 8.20 - Enkel numerisk lösning, svaghet 2 (Ma1c)

Vi ändrar på rad 3 i föregående kodstycke till `if 3 .* x .- 7 == 4`. Lösningen på $3x - 7 = 4$ är x=3,6666... men programmet kan bara hitta heltalslösningar. När datorn testar `x=3` så blir vänsterledet lika med 2, och när datorn i nästa varv testar `x=4` så blir vänsterledet lika med 5. Vänsterledet ”hoppar” alltså direkt från 2 till 5, och hoppar över rätt svar. Det finns inget jättelätt sätt att fixa programmet - vi behöver byta till en mer avancerad metod.

Övning 8.21 - Lös linjär ekvation med `solve` (Ma1c)

$x = -6$

```
syms x;
solve(4*x+15==-9, x)
```

Övning 8.22 - Klurig numerisk lösning med intervallhalvering (Ma1c)

Precis som i ”gissa talet”-spelet så är det smart att gissa mitt emellan två tal. Sedan testar vi vår gissning `xmed` genom att mata in den i ekvationen, och kolla om vi hamnade *under* eller *över* 4. Den grundläggande strategin brukar kallas *binär sökning* eller *intervallhalvering*, och används som lösning på en mängd olika problem inom programmering.

```
xmin = -10; % dessa måste sättas manuellt,
xmax = 10; % så att de "omfamnar" svaret
i = 1;
while i <= 20 % ju fler iterationer desto mer exakt svar
   xmed = (xmin + xmax) / 2; % gissa mitt emellan
   % testa ekvationens (snarare olikhetens) värde
   % i tre punkter:
   ymin = 3 * xmin - 7 < 4;
   ymax = 3 * xmax - 7 < 4;
   ymed = 3 * xmed - 7 < 4;
   % kontrollera att vi är på vardera sidan om rätt svar:
   if ymin == ymax
      disp('fel! välj bättre xmin och xmax. Starta om');
       i = 10000; % avbryt loopen i förtid
   end
   % välj vilken halva vi ska söka vidare i:
   if ymin ~= ymed
      xmax = xmed;
   end
   if ymax ~= ymed
      xmin = xmed;
   end
   i = i + 1;
end
disp(xmin);
disp(xmax);
```

Det här programmet är för övrigt ett bra exempel på när det skulle löna sig att skapa våra egna funktioner i Matlab/Octave. Med en egen funktion skulle vi kunna slippa upprepa nästan samma ekvations-kod tre gånger. Om du har lust och tid över, sök information om det på nätet, t.ex. på *https://se.mathworks.com/help/matlab/ref/function.html*

Övning 8.23 - Lös andragradsekvation med `solve` (Ma2c)

$x_1 = -6, x_2 = 1$

```
syms x;
solve(x*x + 5*x - 6 == 0, x)
```

Övning 8.24 - Lös andragradsekvation med `roots` (Ma2c)

- `roots([6 -13 5])` ger $x_1 = 1.66667$ och $x_2 = 0.5$
- `roots([4 -2 -6])` ger $x_1 = 1.5$ och $x_2 = -1$
- $2/x - 18 = 0$ är inte en polynomekvation, går inte att lösa med `roots`
- `roots([3 -2 0])` ger $x_1 = 0.66667$ och $x_2 = 0$

Övning 8.25 - Klurig numerisk lösning, del 2 (Ma2c)

Samma program som i Övning 8.22, förutom att vi ändrar startvärdena `xmin` och `xmax`, samt ändrar ekvationen (olikheten). Programmet hittar lösningen $x = 14.667$

```
xmin = -50;
xmax = 50;
i = 1;
while i <= 20
   xmed = (xmin + xmax) / 2;
   ymin = xmin^6 - sin(xmin) - 3^xmin + 7 < 0;
   ymax = xmax^6 - sin(xmax) - 3^xmax + 7 < 0;
   ymed = xmed^6 - sin(xmed) - 3^xmed + 7 < 0;
   if ymin == ymax
      disp('fel! välj bättre xmin och xmax. Starta om');
       i = 10000; % avbryt loopen i förtid
   end
   if ymin ~= ymed
      xmax = xmed;
   end
   if ymax ~= ymed
      xmin = xmed;
   end
   i = i + 1;
end
disp(xmin);
disp(xmax);
```

Övning 8.26 - Klurig numerisk lösning, del 3 (Ma2c)

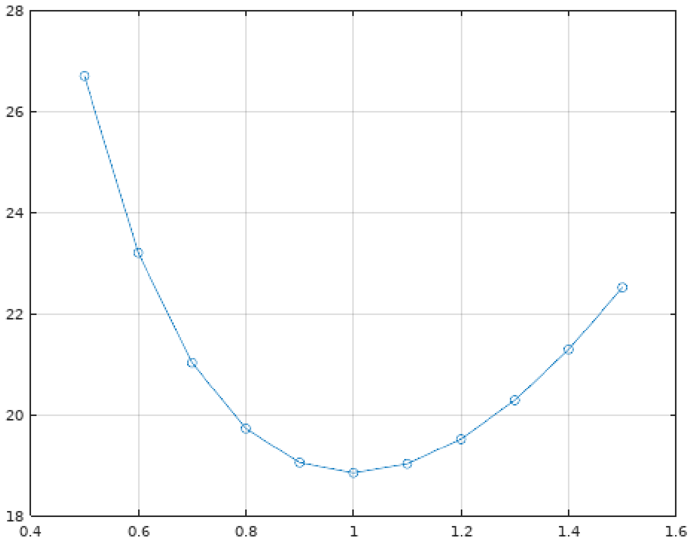
Nej, lösningsförslaget i den här boken klarar inte att lösa den ekvationen. Intervallhalverings-metoden bygger ju på att `ymin` och `ymax` ska vara på varsin sida rätt svar, så metoden kräver alltså att kurvan korsar linjen $y = 0$. Men polynomfunktionen $y = x^2 + 5x - 6$ bara nuddar vid linjen utan att korsa den! Det finns dock andra numeriska metoder som klarar att lösa sådana ekvationer.

Övning 8.27 - Andragradsfunktionens minsta värde, grafiskt (Ma2c, Ma3c)

Radie=1 dm, höjd=2 dm

```
radius = [0.5 : 0.1 : 1.5];
volume = 6.2832;
height = volume ./ (pi .* (radius.^2));
area = 2 .* pi .* radius .* (radius .+ height);
plot(radius, area, '-o');
```

Figur 9.1: Grafen för ovanstående kod



Övning 8.28 - Exponentialfunktion (Ma1c, Ma2c)

6727.5 biljetter (troligen avrundat uppåt eller nedåt).

```
tickets = [1000]; % ett element per år
yr = 1;
while yr <= 20
   tickets(yr+1) = tickets(yr) * (1 + 10/100);
   yr = yr + 1;
end
disp(tickets(end));
plot(tickets, '-o');
```

Övning 8.29 - Numerisk lösning av exponentialekvation (Ma1c, Ma2c)

Den 18:e festivalen.

```
tickets = [1000]; % ett element per år
yr = 1;
while tickets(end) < 5000
   tickets(yr+1) = tickets(yr) * (1 + 10/100);
   yr = yr + 1;
end
disp(yr);
```


Övning 8.30 - Oregelbunden exponentialfunktion (Ma1c, Ma2c)

Den 26:e festivalen har 8110.5 biljetter (troligen avrundat uppåt eller nedåt).

```
nr_years = 25
changes = [];
% changes innehåller ökningen i procent från föregående
   festival
% sätt de två första ökningarna:
changes(2) = 4;
changes(3) = 2;
% räkna ut alla andra ökningar enligt mönstret:
yr = 3;
while yr <= nr_years
   changes(yr+1) = changes(yr-1) + 1;
   yr = yr + 1;
end

tickets = [1000]; % ett element per år
yr = 1;
while yr <= nr_years
   change = (1 + changes(yr+1) / 100);
   tickets(yr+1) = tickets(yr) * change;
   yr = yr + 1;
end
disp(tickets(end));
plot(tickets, '-o');
```

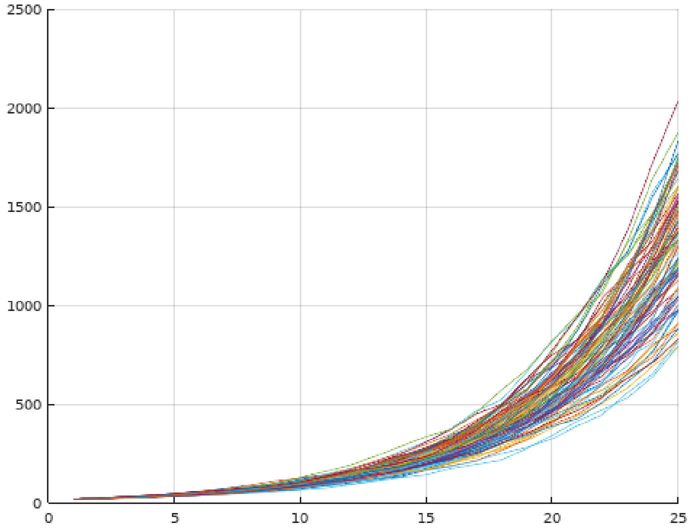
Övning 8.31 - Oregelbunden exponentialfunktion, del 2 + slumpförsök och statistik (Ma1c, Ma2c)

Ungefär 1300 kaniner.

```
hold on;
estimates = [];

i = 1;
while i <= 100
   rabbits = [20]; % ett element per månad
   mon = 1;
   while mon <= 24
      change = 1 + randi([10 28]) / 100;
      rabbits(mon+1) = rabbits(mon) * change;
      mon = mon + 1;
   end

   plot(rabbits);
   estimates(end + 1) = rabbits(end);
   i = i + 1;
end

```

```
% skriv ut medelvärde
disp('Ca antal kaniner: ');
disp(mean(estimates));
```

Figur 9.2



Övning 8.32 - Avbetalning av annuitetslån (Ma1c, Ma2c)

```
% konstanter:
interest = 5; % månadsränta i procent
amort_per_month = 1087.07;

% listor:
months = [0];
sum_payed = [0];
months_twice = [0];
debt = [15000];

while debt(end) > 0
  month = months(end) + 1;

  months(end+1) = month;
  sum_payed(end+1) = sum_payed(end) + amort_per_month;

  months_twice(end+1) = month;
  debt(end+1) = debt(end) * (1 + interest / 100);
  months_twice(end+1) = month;
  debt(end+1) = debt(end) - amort_per_month;
end

hold on;
plot(months, sum_payed, '-o');
plot(months_twice, debt);
```

Övning 8.33 - Förändringshastighet enligt tabell (Ma3c)

Med hjälp av följande kod:

```
x = [2007 2008 2009 2010 2011 2012 2013 2014 2015 2016
   2017];
y = [383.79 385.60 387.43 389.90 391.65 393.85 396.52
   398.65 400.83 404.21 406.53];

index = 10; % det 10e mätvärdet i listan
result = (y(index+1) - y(index-1)) / 2;
disp(result);
```

Får vi resultatet *2.8500* ppm/år.

Övning 8.34 - Förändringshastighet enligt tabell, del 2 (Ma3c)

```
x = [2007 2008 2009 2010 2011 2012 2013 2014 2015 2016
   2017];
y = [383.79 385.60 387.43 389.90 391.65 393.85 396.52
   398.65 400.83 404.21 406.53];

year = input('ange år mellan 2008 och 2016: ');

% hitta index för årtalet användaren vill se:
index = find(x == year);

% räkna ut den centrala differenskvoten kring det året
result = (y(index+1) - y(index-1)) / 2;
disp(result);
```

Övning 8.35 - Ändringskvot för funktion (Ma3c)

Ett närmevärde till derivatan är: -0.19048

```
x = input('ange x-värde att beräkna derivatan för: ');
h = 0.4;
x_left  = x - h;
x_right = x + h;
y_left  = (x_left^2 + 1) / x_left;
y_right = (x_right^2 + 1) / x_right;
estimate = (y_right - y_left) / (2 * h);
disp('ett närmevärde till derivatan är: ');
disp(estimate);
```

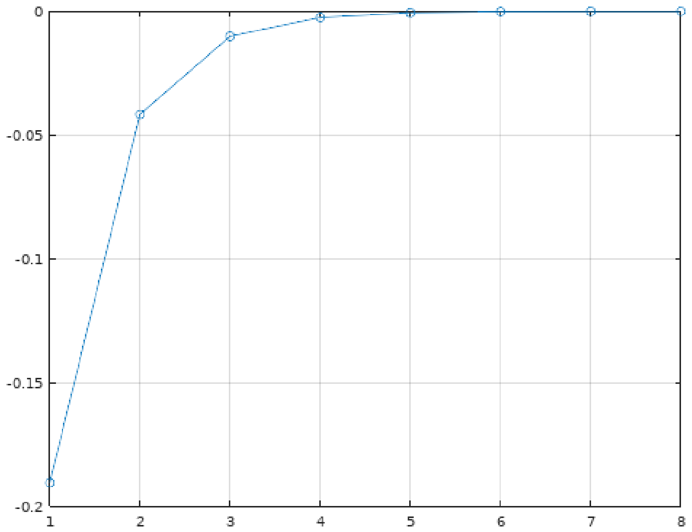
Övning 8.36 - Testa olika värden för h (Ma3c)

Följande kod:

```
estimates = [];
x = input('ange x-värde att beräkna derivatan för: ');
h = 0.4;
num_halvings = 1;
while num_halvings <= 8
    x_left  = x - h;
    x_right = x + h;
    y_left  = (x_left^2 + 1) / x_left;
    y_right = (x_right^2 + 1) / x_right;
    estimate = (y_right - y_left) / (2 * h);
    estimates(end + 1) = estimate;
    h = h / 2;
    num_halvings = num_halvings + 1;
end
plot(estimates, '-o');
```

Ger grafen:

Figur 9.3



I grafen ser det ut som att den exakta derivatan är: 0

Övning 8.37 - Testa den inbyggda `gradient` (Ma3c)

Den symbolhanterande metoden ger svaret i form av ett bråk: $f'(1) = -5/2$, medan följande numeriska kod ger approximationen $f'(1) = -2.5$

```
xp = 1; % i vilken punkt vill vi veta lutningen
h = 0.001;
x = [xp - h : h : xp + h];
y = 3 * sqrt(x) + 2 ./ x.^2; % vår funktion
estimate = gradient(y, h)(2);
disp(estimate);
```

Övning 8.38 - Approximera talet e (Ma3c)

Följande kod ger approximationen $e = 2.7183$

```
emin = 0; % dessa måste sättas manuellt,
emax = 10; % så att de "omfamnar" svaret.

% för ändringskvot. mindre h ger exaktare svar:
h = 0.00001;
i = 1;
while i <= 40 % ju fler iterationer desto exaktare svar
   emed = (emin + emax) / 2; % gissa mitt emellan
   % beräkna ändringskvoten kring emed^1:
   slope = (emed^(1+h) - emed^(1-h)) / (2 * h);
   % vi önskar att 'slope' ska bli exakt lika med emed,
   % så välj vilken halva vi ska söka vidare i:
   if slope > emed
      % vår gissning var för hög, sök i nedre halvan
      emax = emed;
   else
      % vår gissning var för låg, sök i övre halvan
      emin = emed;
   end
   i = i + 1;
end
if abs(slope - emed) < 0.001
   disp('Talet e är ungefär = ');
   disp(emed);
else
   disp('Hittade inte något tal e mellan emin och emax.');
   disp('Ändra startvärden och försök igen.');
end
```


10 SAKREGISTER