

# Kort om programmering i MATLAB

## 1 Inledning

Redan första tillfället gjorde ni ett litet program. Ni skrev ett `script` eller en skriptfil som beräknade summan

$$\sum_{i=1}^5 i^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2$$

med en programkod som kanske såg ut så här

```
s=0;
for i=1:5
    s=s+i^2;
end
s
```

Nu skall vi lära oss lite mer om programmering. Vi börjar med att se på skript- och funktionsfiler som vi använder för att ge struktur åt program, därefter ser vi hur vi kan styra ut- och inmatning av data för att avslutningsvis se på kontrollstrukturer som `if`-, `for`- och `while`-satser med vilka vi styr flödet genom koden i ett program.

## 2 Skript och funktioner

Vi har redan sett lite på skript och funktioner. Nu måste vi bli lite grundligare.

### 2.1 Skript

Ett `script` eller en skriptfil är en textfil som innehåller det man skulle kunna skriva direkt i `Command Window`, och som utförs i MATLAB genom att man ger textfilens namn som kommando. För att MATLAB skall hitta filen, förutsätter det att katalogen där filen ligger är aktuell katalog eller man satt en sökväg med `path`, se hjälptexten. Utanför MATLAB får namnet på ett `script` tillägget `.m` för att skilja den från andra filer.

Programsatserna i en skriptfil opererar *globalt* på variablerna i arbetsarean (`Workspace`).

Alla utskrifter från programmet skrivs som standard i `Command Window`, liksom alla felmeddelanden. Man kan också styra utskrifter av beräkningsresultat till en fil.

Editorn i MATLAB markerar koden med olika färger för att visa vad som är kommentarer, nyckelord, textsträngar, etc., och har flera funktioner för att underlätta vårt arbete. Ett exempel: Automatisk indentering. Ifall man markerar en bit programkod, och väljer `Indent` på `EDITOR`-fliken så indenteras koden om (dvs. vänstermarginalen justeras). Det kan vara användbart när man klippt-och-klistrat lite kod.

## 2.2 Funktioner

Det finns flera olika sätt att göra egna funktioner i MATLAB. Om funktionen innehåller flera uttryck eller satser måste man göra en **function** eller funktionsfil, dvs. skapa en textfil med funktionsbeskrivningen. Består funktionen av ett enda uttryck så kan vi göra ett s.k. funktionshandtag (**function handle**) eller s.k. anonyma funktioner (**anonymous functions**).

En **function** är en textfil med samma namn som funktionen och som inleds med en funktionsdeklaration.

För större program kan man vilja använda andra sätt att skriva funktioner, exempelvis underfunktioner (**subfunction**). Vi kan t.ex. ha en funktion som behöver hjälpfunktioner som inte är av intresse utanför huvudfunktionen, då lägger vi dem som underfunktioner.

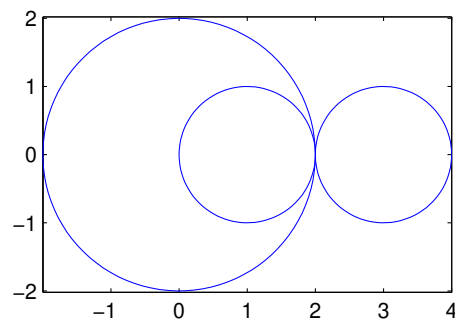
En funktionsfil påminner mycket om en skriptfil. Det som skiljer är att i första raden innehåller textfilen ordet *function* och att argument kan skickas med. Alla variabler som definieras inne i en funktionsfil är *lokala*, dvs. de opererar inte globalt på arbetsarean (**Workspace**).

**Exempel 1.** Vi vill rita cirklar med olika medelpunkt  $(a,b)$  och radie  $r$ . Vi gör en funktion med namnet `cirkel.m` med  $a,b$  och  $r$  som indata och två vektorer  $\mathbf{x}$  och  $\mathbf{y}$  som utdata enligt

```
function [x,y]=cirkel(a,b,r)
    t=linspace(0,2*pi);
    x=a+r*cos(t);
    y=b+r*sin(t);
```

I de två vektorerna har vi koordinater för punkter jämnt fördelade på cirkeln. Vi ritar några cirklar med

```
>> [x,y]=cirkel(0,0,2);
>> plot(x,y)
>> axis equal
>> hold on
>> [x,y]=cirkel(1,0,1);
>> plot(x,y)
>> [x,y]=cirkel(3,0,1);
>> plot(x,y)
>> hold off
```



**Uppgift 1(a).** Gör funktionen i exempel 1 som en funktionsfil, med namnet `cirkel.m` förslagsvis. Prova funktionen, rita någon cirkel, vilken typ av vektorer ger den som resultat (rad eller kolonn) och hur långa är de. Ändra er funktion så att ni får 50 punkter på cirkeln, eller varför inte 5 punkter. Rita de ”nya” cirklarna.

**(b).** Om vi lägger till kommentarer direkt under **function** enligt

```
function [x,y]=cirkel(a,b,r)
% Hjälpstext, det man skriver direkt under function kommer att skrivas
% ut som hjälpstext om man använder help i kommandofönstret
    t=linspace(0,2*pi);
    x=a+r*cos(t);
    y=b+r*sin(t);
```

och skriver `help cirkel` i Command Window så kommer hjälpstexten skrivas ut. Vilken hjälpstext tycker ni är bra till vår funktion? Skriv in er hjälpstext i funktionsfilen och pröva med `help`.

### 3 In- och utmatning

I samband med programmering behöver vi olika former av in- och utmatning. Ibland kan man t.ex. vilja ha inmatning till ett `script` medan det utförs (körs) eller att beräkningsresultat skall skrivas ut med ett visst format, t.ex. ett visst antal decimaler.

Med `input` kan vi mata in ett värde enligt

```
>> antal=input('Ange antal kast: ');
Ange antal kast: 5
```

När MATLAB kommer till `input`-kommandot så skrivs texten `'Ange antal kast:'` ut och programmet väntar på raden tills vi skriver ett svar, i vårt fall 5, variabeln `antal` ges detta värde och programmet fortsätter sedan med nästa kommando. Semikolonet (`;`) efter `input` gör att vi inte får någon utskrift då `antal` får sitt värde, 5:an vi ser är den vi skrev. (Variabeln får givetvis ha vilket namn som helst, vi valde namnet `antal` i exemplet.)

Vill vi mata in en textsträng med `input` lägger vi till ett `'s'` och vill vi få en ny rad för vårt svar får vi det med `\n` enligt

```
>> svar=input('Hej, hur mår du?\n','s');
Hej, hur mår du?
Bra
```

Om vi inte använder semikolon (`;`) efter t.ex. en tilldelning så skrivs variabelnamnet ut tillsammans med det värde variabeln fått. Vill vi bara skriva ut värdet av variabeln kan vi använda `disp`. Vi testar på variabeln `svar`

```
>> disp(svar)
Bra
```

Med `sprintf` och `fprintf` kan vi skriva formaterad text. Vill vi ha utskriften till en textsträng använder vi `sprintf` och vill vi ha den till en textfil använder vi `fprintf`. Som exempel skriver vi ut  $\pi$  med 7 decimaler i Command Window enligt

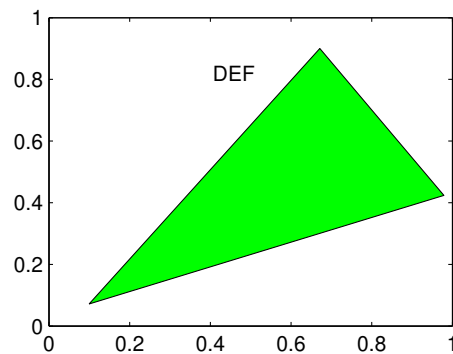
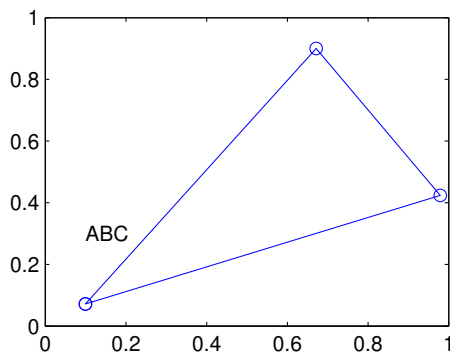
```
>> disp(sprintf('Pi =%10.7f',pi))
Pi = 3.1415927
```

MATLAB hämtar kommandona `sprintf` och `fprintf` från C, därför ser formatkoderna ut som i programspråken C och Java. Det är inget vi behöver lära oss just nu, vid behov ser man i hjälptexterna.

Om man vill läsa in koordinater med musen kan man använda `ginput`. Vi kan placera ut text i ett koordinatsystem med kommandona `text` och `gtext`. Med `text` får vi ge koordinater för var texten skall placeras och med `gtext` använder du musen för att peka på önskad plats. Vi ser på detta med hjälp av ett exempel.

**Exempel 2.** Tidigare ritade vi en triangel genom att ge koordinaterna med siffervärden. Ibland är det smidigare att använda `ginput`, vi pekar på punkter i koordinatsystemet och trycker på en musknapp.

```
>> clf
>> subplot(1,2,1)
>> [x,y]=ginput(3);           % Triangelns 3 hörn, x och y blir kolonner
>> x=[x; x(1)]; y=[y; y(1)]; % Vi sluter polygontåget, så alla sidor ritas
>> plot(x,y,'-o')
>> axis([0 1 0 1])
>> text(0.1,0.3,'ABC')       % Koordinatplacerad text på grafen
>> subplot(1,2,2)
>> fill(x,y,'g')
>> axis([0 1 0 1])
>> gtext('DEF')              % Med markören placerad text på grafen
```



Att vi sluter polygontåget innan vi ritar upp beror på att vi vill att triangelns alla sidor skall ritas, men vi vill inte behöva försöka markera det första hörnet en gång till.

Med `[x; x(1)]` bildar vi en vektor med alla ursprungliga  $x$ -koordinater ( $x$ ) först och till dessa lägger vi första  $x$ -koordinaten ( $x(1)$ ) en gång till (vi sluter). Hakparanterna (`[ ]`) används för att bygga upp den nya (större) vektorn. Sedan får `x` detta värde. På motsvarande sätt gör vi i  $y$ -led.

Med `[x,y,b]=ginput(1)` läser man in koordinater för en punkt och får samtidigt reda på vilken musknapp det trycktes på. Vänster musknapp ger `b` värdet 1, mellersta ger `b` värdet 2, osv. Detta kommer vi ha nytta av lite senare.

**Uppgift 2.** Mata in punkter med `ginput` så att du får ett polygontåg som du sedan sluter. Du skall inte bestämma antal punkter i förväg, men det skall vara fler punkter än i en triangel. Färglägg sedan området som innesluts med någon färg. Sök på `ginput` i hjälptexterna så du kan ta reda på hur man läsa in koordinater för flera punkter, utan att i förväg behöva bestämma hur många.

## 4 Kontrollstrukturer

Nu skall vi lära oss lite mer om kontrollstrukturer som `if`-, `for`- och `while`-satser med vilka vi styr flödet genom programkoden.

### 4.1 Logiska uttryck och operationer

Vi kommer behöva använda logiska villkor av typen  $x > 5$ . Detta uttryck är sant om ett tal vi betecknar med  $x$  är strängt större än talet 5, annars falskt. Vi skriver detta uttryck i MATLAB som `x>5` helt enkelt. Det logiska värdet sant beskrivs i MATLAB av talet 1 och falskt beskrivs av talet 0.

Relationsoperatorerna `<`, `≤`, `>`, `≥`, `=` och `≠` skrivs i MATLAB med `<`, `<=`, `>`, `>=`, `==` respektive `~=`. Observera dubbla likhetstecken i MATLAB för att beteckna (logisk) likhet, enkelt likhetstecken används ju för tilldelning, dvs. att ge en variabel ett värde. Vidare har vi ibland nytta av funktionerna `any` och `all` som arbetar på vektorer (se gärna hjälptexten i MATLAB).

De logiska operatorerna ”och”, ”eller” samt ”negation” skrivs i MATLAB med `&`, `|` respektive `~`.

### 4.2 Villkorssatser

Det allmänna utseendet på en `if`-sats är någon av följande alternativ

<code>if uttryck</code>	<code>if uttryck</code>	<code>if uttryck</code>	<code>if uttryck</code>
<code>  satser</code>	<code>  satser</code>	<code>  satser</code>	<code>  satser</code>
<code>end</code>	<code>else</code>	<code>elseif uttryck</code>	<code>elseif uttryck</code>
	<code>  satser</code>	<code>  satser</code>	<code>  satser</code>
	<code>end</code>	<code>end</code>	<code>else</code>
			<code>  satser</code>
			<code>end</code>

där vi kan ha godtyckligt många `elseif` i de två sista alternativen. Med `uttryck` avser vi ett logiskt uttryck av den typen vi nämnde ovan.

**Exempel 3.** Vi har två värden  $a$  och  $b$  och vill att  $c$  skall ges det största av dessa värden. Detta kan göras med följande kod

```
if a<b
    c=b
else
    c=a
end
```

Om  $a < b$  är sant, så är  $b$  störst och  $c$  ges värdet av  $b$ , annars är  $a$  störst och  $c$  ges värdet av  $a$ .

**Exempel 4.** Gör en funktion som beräknar medianen av värdena i en vektor. Medianen är det mittersta värdet i storleksordning om vektorn har udda antal element, och medelvärdet av de två mittersta (i storleksordning) om antal element är jämnt.

Längden av en vektor (antal element) ges av `length` och elementen i en vektor sorteras i storleksordning av `sort`.

Funktionen `rem` ger resten vid heltalsdivision, t.ex. får `rem(n,2)` värdet 0 om  $n$  är ett jämnt heltal och värdet 1 om  $n$  är ett udda heltal.

Nu skriver vi vår funktion

```
function m=min_median(v)
% m = min_median(v) beräknar medianen av elementen i vektorn v
%
    s=sort(v);                % s sorterad version av v
    n=length(v);             % n antal element i v
    if rem(n,2)==0           % n jämnt
        m=(s(n/2)+s(n/2+1))/2;
    else                      % n udda
        m=s((n+1)/2);
    end
```

som vi lagrar under namnet `min_median.m` och ser hjälptexten med

```
>> help min_median
    m = min_median(v) beräknar medianen av elementen i vektorn v
```

och tar medianen av en slumpvektor (`rand` ger slumpantal mellan 0 och 1)

```
>> v=rand(1,6)
v =
    0.4103    0.8936    0.0579    0.3529    0.8132    0.0099

>> m=min_median(v)
m =
    0.3816
```

Nu har MATLAB en inbyggd funktion `median` för att bilda medianen som vi givetvis använder istället. Titta på hjälptexten för `median` för att se en "riktig" hjälptext.

Vi skulle kunna förbättra vår cirkel-funktion lite med en `if`-sats så att vi kan ge antal punkter på cirkeln som indata, om vi vill.

```
function [x,y]=cirkel(a,b,r,n)
    if nargin~=4 % nargin ger antal indata som finns med då funktionen används
        n=100; % Om antalet inte fyra, dvs n inte finns med som indata
    end % så ges n värdet 100
    t=linspace(0,2*pi,n);
    x=a+r*cos(t);
    y=b+r*sin(t);
```

Med `[x,y]=cirkel(a,b,r,100)` får vi 100 punkter och det får vi även med `[x,y]=cirkel(a,b,r)`, medan t.ex. `[x,y]=cirkel(a,b,r,50)` ger 50 punkter.

**Uppgift 3.** Prova att förändra din cirkel-funktion!

## 4.3 Repetitionssatser

För att upprepa en grupp av satser flera gånger används **for**-satser eller **while**-satser. Vet vi på förhand hur många gånger upprepningen skall ske, så är normalt en **for**-sats att föredra i annat fall är en **while**-sats lämpligare.

### 4.3.1 for-satser

Det allmänna utseendet på en **for**-sats är

```
for variabel = uttryck
  satser
end
```

Här är **uttryck** en vektor av tal eller ett uttryck som bygger upp en sådan vektor. Successivt kommer **variabel** tilldelas värdena i **uttryck** i tur och ordning och samtidigt kommer alla **satser** ned till **end** att utföras. En gång för varje värde som **variabel** ges.

Allra vanligast är följande enkla variant

```
for variabel = start:steg:slut
  satser
end
```

Vi har redan sett på några summor som vi beräknat med **for**-sats och här kommer några ytterligare exempel.

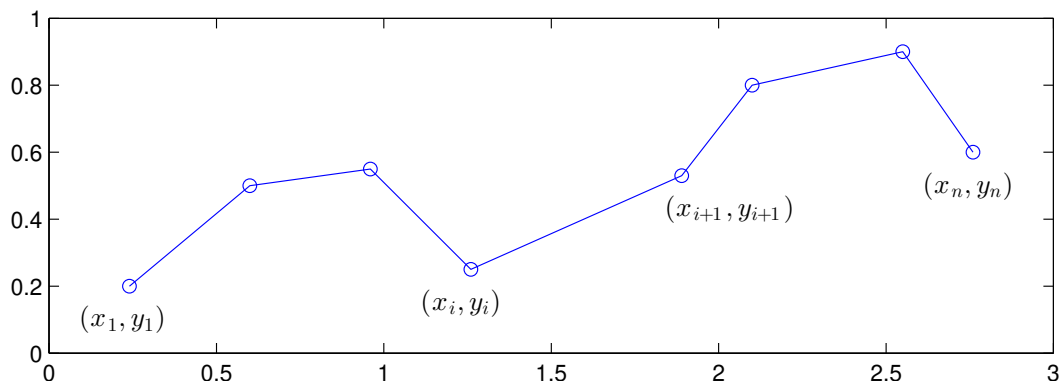
**Exempel 5.** Vi kan beräkna summan  $s = 1 + 3 + 5 + 7 + 9 + 11 + 13$  med en programkod som kanske ser ut så här

```
s=0;
for i=1:2:13
  s=s+i;
end
s
```

Vi kan se det som att vi har en låda **s** som vi samlar värden (termer) i. Först ser vi till att lådan från början är tom med  $s=0$ .

När **for**-satsen utförs kommer **i** successivt få värdena  $1, 3, 5, \dots, 13$ . För varje värde som **i** får kommer  $s=s+i$  utföras, dvs. vi kommer lägga aktuellt värde på **i** till det vi redan har i lådan **s**.

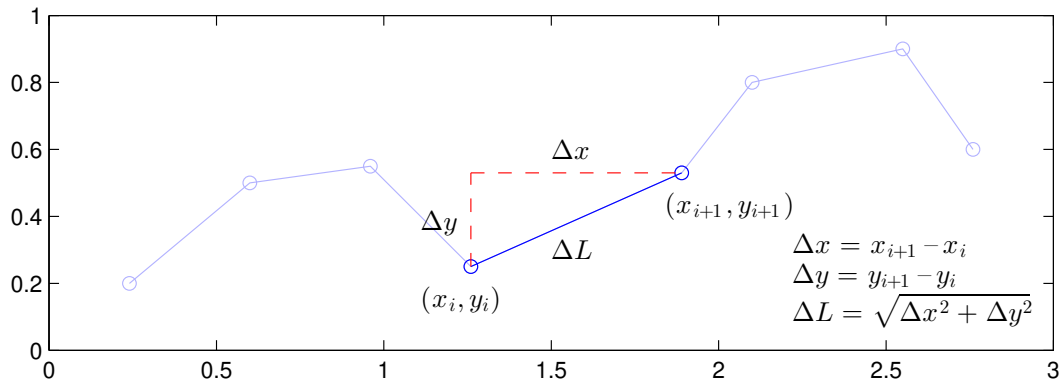
**Exempel 6.** Vi tänker oss att vi har ett polygontåg  $(x_1, y_1), (x_2, y_2) \dots, (x_n, y_n)$  som vi ritat en figur av



Vill vi beräkna polygontågets längd kan vi göra det med

$$L = \sum_{i=1}^{n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

Denna formel fås genom att använda Pytagoras sats på varje segment i polygontåget.



Antag att koordinaterna samlade i två vektorer  $\mathbf{x} = (x_1, x_2, \dots, x_n)$  och  $\mathbf{y} = (y_1, y_2, \dots, y_n)$ , då beräknar vi längden enligt

```
>> n=length(x);
>> L=0;
>> for i=1:n-1
    L=L+sqrt((x(i+1)-x(i))^2+(y(i+1)-y(i))^2);
end
>> L
```

Om polygontåget är slutet, dvs. att  $x_n = x_1$  och  $y_n = y_1$ , så omsluts ett område med arean

$$A = \left| \frac{1}{2} \sum_{i=1}^{n-1} (x_{i+1} + x_i)(y_{i+1} - y_i) \right|$$

Denna formel är lite svårare, men i läsperiod 3 kommer ni läsa den matematik som behövs för att ta fram den. Så här beräknar vi arean i alla fall

```
>> n=length(x);
>> A=0;
>> for i=1:n-1
    A=A+(x(i+1)+x(i))*(y(i+1)-y(i));
end
>> A=abs(A)/2
```

**Uppgift 4.** Skriv en funktion med namnet `polylen`, för beräkning av längden av ett polygontåg enligt exemplet ovan. Funktionen skall ha följande struktur

```
function L = polylen(x,y)
    satser
```

Pröva din funktion på en triangel och en rektangel, dvs. skapa två vektorer  $\mathbf{x}$  och  $\mathbf{y}$  med koordinater för t.ex. en triangel och beräkna längden av polygontåget (i detta fall omkretsen) med `Omkrets=polylen(x,y)`. Skriv även en funktion med namnet `polyarea`, för beräkning av arean. Pröva även denna funktion på triangeln och rektangeln. Är resultaten rimliga?



### 4.3.2 while-satser

En while-sats tillåter en grupp av satser att bli repeterade under kontroll av ett logiskt villkor:

```
while uttryck
    sats
end
```

Uttrycket i while-satsen är ett logiskt uttryck. Satserna repeteras så länge det logiska uttrycket är sant.

**Exempel 7.** Man kan beräkna  $\sqrt{c}$  med upprepade additioner och divisioner med iterationsformeln

$$x_{k+1} = \frac{1}{2} \left( x_k + \frac{c}{x_k} \right), \quad k = 0, 1, 2, \dots$$

där  $x_0 = c$ . Iterationen avbryts då  $d = |x_{k+1} - x_k| < \text{tol}$ , där tol är måttet på önskad noggrannhet i approximationen.

Vi beräknar en approximation av t.ex.  $\sqrt{2}$  med noggrannheten  $\text{tol} = 0.5 \times 10^{-10}$ , dvs. 10 korrekta decimaler. Först ser vi på några steg i iterationen

$$\begin{array}{ll} x_0 = c & x_0 = 2 \\ x_1 = \frac{1}{2} \left( x_0 + \frac{c}{x_0} \right) & x_1 = \frac{1}{2} \left( 2 + \frac{2}{2} \right) = 1.5 \\ x_2 = \frac{1}{2} \left( x_1 + \frac{c}{x_1} \right) & x_2 = \frac{1}{2} \left( 1.5 + \frac{2}{1.5} \right) = 1.4166 \dots \\ x_3 = \frac{1}{2} \left( x_2 + \frac{c}{x_2} \right) & x_3 = \frac{1}{2} \left( 1.4166 \dots + \frac{2}{1.4166 \dots} \right) = 1.4142 \dots \\ \vdots & \vdots \end{array}$$

Sedan använder vi MATLAB enligt

```
>> c=2;
>> tol=0.5e-10;
>> x=c;
>> d=1;
>> while d>tol           % så länge d>tol görs följande
    xny=(x+c/x)/2;
    d=abs(xny-x);
    x=xny;
>> end                   % slutet på while-satsen
>> x
x =
    1.4142
```

Vi kan gör en skriptfil av koden ovan, alternativt kan vi göra det som en function, (`min_sqrt.m`)

```
function x=min_sqrt(c)
    tol=0.5e-10;
    x=c; d=1;
    while d>tol
        xny=(x+c/x)/2;
        d=abs(xny-x);
        x=xny;
    end
```

Vi använder den så här (för att beräkna  $\sqrt{5}$ )

```
>> x=min_sqrt(5)
x =
    2.2361
```

**Uppgift 5.** Det gäller att

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots$$

Hur många termer måste man ta med i summan för att approximera  $\pi$  med fem korrekta decimaler?

Termerna i summan kan skrivas  $\frac{(-1)^i}{2i+1}$  för  $i = 0, 1, \dots$ .

(a). Bilda successivt delsummor

$$s_n = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots + \frac{(-1)^n}{2n+1}$$

för allt större heltal  $n$  och avbryt när  $s_n$  tillräckligt nära  $\frac{\pi}{4}$ . Använd en **while**-sats.

(b). Använd sedan en **for**-sats och beräkna  $\frac{\pi}{4}$  med 1000 termer.

Ibland har man nytta av följande konstruktion

```
while 1
    sats
    if uttryck
        break
    end
end
```

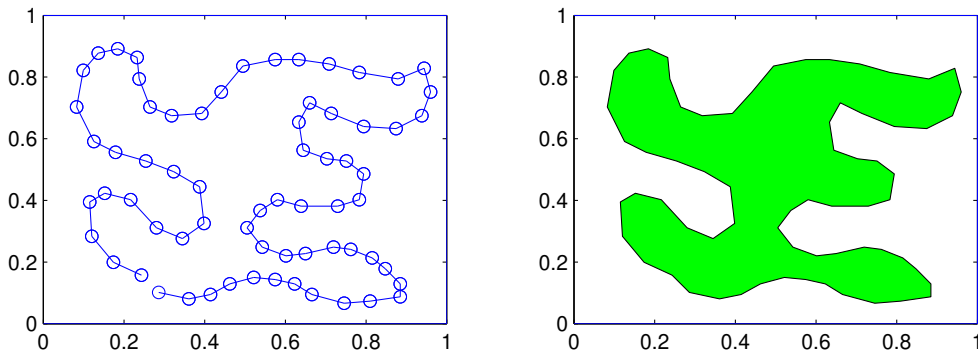
Här kommer upprepning ske ända tills uttrycket i **if**-satsen blir sant (får värdet 1), då avbryts **while**-satsen och programkoden efter denna utförs.

**Exempel 8.** Vi vill rita ett polygontåg genom att markera punkter med **ginput** och succesivt rita upp linjer som förbinder på varandra följande punkter.

Vi håller på att markera nya punkter och rita linjer så länge vi trycker på vänster musknapp, annars avbryter vi.

```
>> axis([0 1 0 1]), hold on
>> [x,y]=ginput(1);
>> plot(x,y,'o')
>> xpol=x;
>> ypol=y;
>> while 1
    [x,y,knapp]=ginput(1);
    if knapp~=1
        break
    end
    xpol=[xpol; x];
    ypol=[ypol; y];
    plot(xpol(end-1:end),ypol(end-1:end),'o-')
end
>> hold off
```

Lägg märke till att vi sparar alla koordinater så att vi kan göra annat med polygontåget, t.ex. beräkna dess längd.



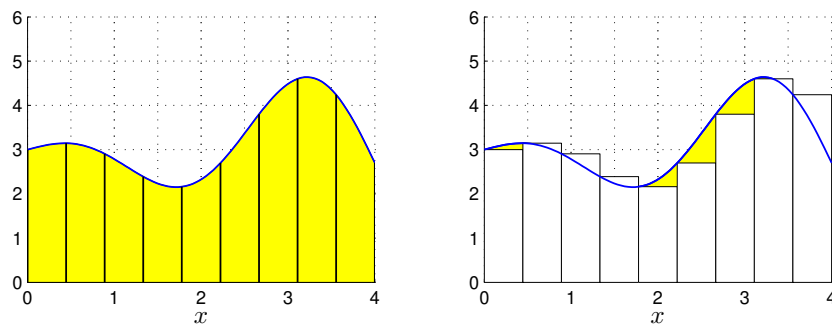
**Uppgift 6.** Skriv en skriptfil som gör det möjligt att markera hörnpunkter i ett polygonområde och som beräknar arean av polygonområdet samt längden av dess rand, med hjälp av funktionerna från uppgift 4. Områdets inre skall fyllas i med någon färg. Se exempel 2 och 8.

## 5 Vektorisering och effektivitet

Tidtagning är ett bra sätt att studera effektivitet. Det man kan tjäna i särklass mest på när det gäller effektivisering är fördimensionering av matriser och vektorer samt vektorisering, som vi skall se några exempel på nu.

**Exempel 9.** En integral kan approximeras med *vänster rektangelregel* enligt

$$\int_a^b f(x) dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x) dx \approx \sum_{i=1}^n h f(x_{i-1}), \quad \text{där } h = \frac{b-a}{n}$$



Vi kan beräkna t.ex.  $\int_0^1 x \sin(x) dx$  med vänster rektangelregel med  $n = 100$  säg, på följande sätt

```
>> a=0; b=1; f=@(x)x.*sin(x);
>> n=100; h=(b-a)/n
>> q=0;
>> for i=0:n-1
    x=a+i*h;
    q=q+h*f(x);
end
```

Att använda en `for`-sats är i regel ineffektivt i MATLAB. Vi genererar hellre en vektor av alla funktionsvärdena  $f(x_i)$  och sedan summerar dessa enligt

```
>> a=0; b=1; f=@(x)x.*sin(x);
>> n=100; h=(b-a)/n;
>> x=linspace(a,b,n+1);
>> q=sum(h*f(x(1:n)));
```

Detta sätt att organisera en beräkning kallas att vektorisera den, dvs. man genererar först en eller flera vektorer och utför sedan den önskade beräkningen på dem.

De elementvisa operationerna `.*` `./` `.^` är exempel på vektoriserade operationer. Vi använde funktionen `sum` som snabbt summerar en vektor.

**Exempel 10.**I exempel 6 såg vi på ett polygontåg, som gavs av  $(x_1, y_1), \dots, (x_n, y_n)$ . Som vi sett tidigare ges polygontågets längd av

$$L = \sum_{i=1}^{n-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

och vi beräknade längden med

```
>> n=length(x);
>> L=0;
>> for i=1:n-1
    L=L+sqrt((x(i+1)-x(i))^2+(y(i+1)-y(i))^2);
end
```

Betydligt effektivare blir det om vi använder vektorbaserade operationer enligt

```
>> L=sum(sqrt((x(2:end)-x(1:end-1)).^2+(y(2:end)-y(1:end-1)).^2))
eller varför inte
>> L=sum(sqrt(diff(x).^2+diff(y).^2))
```

**Exempel 11.** Beskriv följande funktion  $f$  och rita dess graf.

$$f(x) = \begin{cases} 0, & x < 1 \\ 1, & 1 \leq x \leq 2 \\ 0, & x > 2 \end{cases}$$

Alternativ med `for`-sats

```
function f=stepfun(x)
    f=zeros(size(x));
    for k=1:length(f)
        if 1<=x(k) & x(k)<=2
            f(k)=1;
        end
    end
end
```

Smartare alternativ med `find`

```
function f=stepfun(x)
    f=zeros(size(x));
    f(find(1<=x & x<=2))=1;
```

Vi ritas bild med

```
>> x=linspace(-1,3,200);
>> plot(x,stepfun(x))
>> axis([-1 3 -1 2])
```

