Flervariabelanalys med Matlab

Thomas Wernstål
Institutionen för Matematiska Vetenskaper vid
Chalmers Tekniska Högskola

3 januari 2018

Innehåll

1 Kurvor och ytor

1.1 Funktionsytor

I detta kompendium kommer vi på olika sätt studera funktioner från $\mathbb{R}^n$ till $\mathbb{R}^m$, där n och m är $1, 2$ eller 3, och i kapitel 1 och 4 skall vi speciellt se hur sådana funktioner kan visualiseras geometriskt. Vi börjar i detta avsnitt med att studera funktioner från $\mathbb{R}^2$ till $\mathbb{R}$ dvs reellvärda funktioner av två variabler. Innan vi går in på olika plottkommandon så kan det vara praktiskt att titta på hur man skapar en anonym funktion i MATLAB, där funktionen beror av två variabler. Funktionen $f(x, y) = x \sin y^2$ skapar vi med kommandot

```
>> f=@(x,y) x.*sin(y.^2);
```

När man definierar funktioner så är det viktigt att tänka på att MATLAB arbetar med matriser och matrisoperationer. Vill man att funktionen skall klara elementvisa kalkyler (som i de flesta fall i denna kurs) måste man använda "punkterade operationer" (dvs. `.^` `.*` och `./` istället för bara `^` `*` och `/`). Om en viss anonym funktion saknar punkter för att klara elementvisa kalkyler så kan man också be MATLAB att sätta ut dem på de rätta ställena med `vectorize` och `str2func`. Testa t.ex.

```
>> f=@(x,y) x*sin(y^2);
>> f=str2func(vectorize(f))
```

Att för hand rita funktionsytor är inte helt lätt. Låt oss därför titta på hur man kan använda MATLAB för att få en bild av grafen till en funktion av två variabler. Antag att vi vill plotta grafen till en funktion $f(x, y)$ över en rektangel $-1 \leq x \leq 2, 0 \leq y \leq 3$. Vi börjar då med att skapa koordinatmatriser med hjälp av kommandot `meshgrid`. Detta kommando ger oss en massa punkter (x_i, y_j) i xy-planet i vilket vi sedan kan beräkna funktionsvärdena $f(x_i, y_j)$. Kommandot

```
>> [X Y]=meshgrid(-1:0.5:2,0:0.4:3)
```

skapar t.ex. två matriser `X` och `Y` med x- resp. y-koordinater som är sådana att alla rader i `X` består av talen $-1, -0.5, 0, ..., 2$ (dvs. de som genereras av `-1:0.5:2`) och alla kolonner i `Y` består av talen $0, 0.4, 0.8, ..., 2.8$ (dvs. de som genereras av `0:0.4:3`).

$$X = \begin{bmatrix} -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \\ -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \\ -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \\ -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \\ -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \\ -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \\ -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \\ -1 & -0.5 & 0 & 0.5 & 1 & 1.5 & 2 \end{bmatrix} \quad Y = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0.4 & 0.4 & 0.4 & 0.4 & 0.4 & 0.4 & 0.4 \\ 0.8 & 0.8 & 0.8 & 0.8 & 0.8 & 0.8 & 0.8 \\ 1.2 & 1.2 & 1.2 & 1.2 & 1.2 & 1.2 & 1.2 \\ 1.6 & 1.6 & 1.6 & 1.6 & 1.6 & 1.6 & 1.6 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 2.4 & 2.4 & 2.4 & 2.4 & 2.4 & 2.4 & 2.4 \\ 2.8 & 2.8 & 2.8 & 2.8 & 2.8 & 2.8 & 2.8 \end{bmatrix}$$

Genom att ta ett tal ur matrisen `X` och motsvarande tal ur matrisen `Y` så får vi koordinaterna för en punkt i xy-planet. T.ex. får vi punkten (x_5, y_3) (dvs. $(1, 0.8)$ i det här fallet) med kommandot

```
>>[X(3,5) Y(3,5)]
```

Genom att låta `i` och `j` variera så bildar `[X(j,i),Y(j,i)]` $= (x_i, y_j)$ ett gitter av punkter i rektangeln $-1 \leq x \leq 2, 0 \leq y \leq 3$.

Om man har med för många punkter i gittret (dvs. om matriserna `X` och `Y` är för stora) så riskerar man att beräkningar med matriserna tar för lång tid för MATLAB att utföra och i värsta fall kan det ”hänga sig”. Var därför försiktig och börja med ett grövre gitter om du känner dig osäker på vad som kan vara lämpligt. För att t.ex. plotta ytor så har vi i detta kompendium valt att beräkna funktionsvärdena i ca. 400 punkter. Detta innebär att intervallen på x- resp. y-axeln indelas med ca 20 delningspunkter. Om vi har lika många punkter på de två axlarna så blir koordinatmatriserna kvadratiska. Då finns en risk att utelämnade punkter i uttrycket för beräkning av funktionsvärden inte upptäcks. Beräkningar av typen `x*y, x^3` är ju fullt möjliga att utföra om `x` och `y` är kvadratiska matriser. Vi väljer därför i fortsättningen att inte ha lika många punkter i x-led som i y-led. Följande kommandon ger koordinatmatriser för området $-1 \leq x \leq 2$, $0 \leq y \leq 3$ med 21 punkter på x-axeln och 20 punkter på y- axeln (matriserna `X` och `Y` består alltså av 420 element)

```
>> x=linspace(-1,2,21);y=linspace(0,3,20);
>> [X Y] = meshgrid(x,y);
```

Nu när vi skapat våra gitterpunkter så är vi redo att beräkna funktionsvärdena.

Kommandot

```
>> Z = f(X,Y);
```

ger en 20×21-matris `Z` där `Z(j,i)` $= f(x_i, y_j)$.

Vi kan nu rita funktionsytan med kommandot

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

eller med kommandot

```
>> surf(X,Y,Z)
```

Om man går in på Tools och väljer Rotate 3D, i figurfönstrets menyer, så kan man rotera ytan och se den från olika håll (håll inne vänster musknapp och rör på musen).

Med kommandot `surfl` kan vi även styra belysningen på ytan. Till exempel kan vi plotta ytan, belyst med en ljuskälla placerad i punkten $(3, 4, 5)$

```
>> surfl(X,Y,Z,[3,4,5])
```

Se även kommandona `light` och `camlight`.

Man kan också ändra färgläggningen av ytorna. Om vi t.ex. vill jämna ut ytans färger så att de inte ändras så tvärt (från en meshruta till en annan) så kan vi använda kommandot

```
>> shading interp
```

Om inget annat anges så bestäms själva färgen på ytan av höjden över xy-planet (dvs. z-koordinatens värde). Vi kan dock själva bestämma den färg som skall kopplas till varje meshpunkt genom att i plottkommandot ange en matris (med samma storlek som koordinatmatriserna) som innehåller information om färg på respektive punkt. Pröva t.ex.

```
>> P=rand(size(X));
>> surf(X,Y,Z,P)
```

Man kan också ändra färgskalan. Pröva t.ex.

```
>> colormap copper
```

Färgsättningen på ytan kan också hämtas från ett fotografi (texture mapping). Pröva t.ex

```
>> P = imread('testpat1.png');
>> warp(X,Y,Z,P)
```

Fler kommandon som anger hur ytor presenteras får du med kommandot.

```
>> help graph3d
```

Vi skall nedan titta på några fler exempel som visar att funktionsytor inte alltid är så snälla och fina som den till funktionen ovan. Ibland kan det till och med vara svårt att se ur den plottade grafen om funktionen är kontinuerlig eller ej. **Resten av detta avsnitt kan betraktas som överbetygsnivå och kan hoppas över vid en första genomläsning**.

Exempel 1: Följande kommandon plottar grafen till funktionen $f(x,y) = x/\left(x^2+y^2\right)$.

```
>> f1=@(x,y) x./(x.^2+y.^2);
>> x=linspace(-1,1,21);y=linspace(-1,1,20);
>> [X Y] = meshgrid(x,y);Z=f1(X,Y);
>> surf(X,Y,Z)
```

Funktionen är ju inte definierad i punkten $(x,y) = (0,0)$ och det är oklart från figuren om det finns något gränsvärde då $(x,y) \to (0,0)$. Vad vi ser är att funktionsvärdena tycks variera mycket för (x,y) nära origo. Det är då skäl att bli lite misstänksam. I själva verket är det ju faktiskt så att $f(x,0) \to \pm\infty$ då $x \to 0$, så funktionen $f(x,y) = x/(x^2+y^2)$ saknar gränsvärde då $(x,y) \to (0,0)$. □

Låt oss titta på några exempel till.

Exempel 2: Om vi plottar grafen till funktionen $f(x,y) = (x^2+y^2)/(x+y)$

```
>> f2=@(x,y) (x.^2+y.^2)./(x+y);
>> x=linspace(-1,1,21);y=linspace(-1,1,20);
>> [X Y] = meshgrid(x,y);Z=f2(X,Y);
>> surf(X,Y,Z)
```

så ser vi att dess funktionsyta har höga toppar nära linjen $x+y=0$. Observera att funktionen inte är definierad i punkter (x,y) där $x+y=0$. Trots att ytan verkar ganska okej nära origo så saknar den gränsvärde där ty $f(x,0) \to 0$ och $f(x,x^2-x) \to 2$ då $x \to 0$ (visa det!). □

En funktion kan emellertid ha gränsvärde i en punkt utan att vara definierad där.

Exempel 3: Betrakta funktionen $f(x,y) = x^3/(x^2+y^2)$.

```
>> f3=@(x,y) x.^3./(x.^2+y.^2);
>> x=linspace(-1,1,21);y=linspace(-1,1,20);
>> [X Y] = meshgrid(x,y);Z=f3(X,Y);
>> surf(X,Y,Z)
```

Funktionen är inte definierad i origo men vi har

$$|f(x,y)-0| = \left|\frac{x^3}{x^2+y^2}\right| = \frac{|x|x^2}{x^2+y^2} \leq \frac{|x|(x^2+y^2)}{x^2+y^2} = |x| \to 0 \ \text{, då } (x,y) \to (0,0).$$

så $f(x,y) \to 0$ då $(x,y) \to (0,0)$. □

Om vi vill studera en funktion som inte är definierad i en viss punkt `(x0,y0)` så bör denna inte finnas med som en punkt i det rutnät som `meshgrid` genererar. Vi kan kontrollera om så är fallet med kommandot `find((X==x0)&(Y==y0))`. Svaret på detta kommando är det/de index som ger den kritiska punkten. Om detta är annat än `[ ]` så kan vi förskjuta rutnätet en liten sträcka i x-led med kommandot `x=x+eps;`. Om vi vill undvika division med noll så kan vi alternativt addera `eps` i nämnaren på funktionen. Värdet på `eps` är förinställt och är det minsta representerbara positiva talet i MATLAB.

1.2 Nivåkurvor

Ett annat vanligt sätt att åskådliggöra en funktion av två variabler, som inte kräver att man avbildar tredimensionellt, är att rita nivåkurvor. För olika värden på konstanten c så ger lösningsmängden till ekvationen $f(x,y) = c$ en nivåkurva i xy-planet. Nivåkurvan $f(x,y) = c$ är helt enkelt mängden av alla punkter (x,y) för vilket funktionsvärdet är c. Man får på detta sätt en tvådimensinell (topografisk) karta, med vars hjälp man kan utläsa funktionsytans utseende. T.ex. använder vanliga orienteringskartor detta sätt för att beskriva olika höjder i naturen. Tekniken används också för bland annat väderkartor. Där förekommer isobarer och isotermer som nivåkurvor till de funktioner som beskriver lufttrycket respektive temperaturen på olika platser.

Om vi t.ex. vill titta på några nivåkurvor till funktionen $f(x,y) = x\sin y^2$ så måste vi som förut börja med att beräkna funktionsvärdena i en massa punkter enl.

```
>> f=@(x,y) x.*sin(y.^2);
>> x=linspace(-1,2,21);y=linspace(0,3,20);
>> [X Y] = meshgrid(x,y);
>> Z=f(X,Y);
```

Sedan får vi nivåkurvor med t.ex. kommandot

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

eller något fylligare med kommandot

```
>> contourf(X,Y,Z)
```

Man kan även lyfta upp nivåkurvorna till respektive nivå i rummet med kommandot

```
>> contour3(X,Y,Z)
```

Det är också möjligt att rita både yta och nivåkurvor i samma figur med t.ex. kommandot

```
>> surfc(X,Y,Z)
```

och med följande kommandon ser man nivåkurvor inlagda på ytan.

```
>> mesh(X,Y,Z), hold, contour3(X,Y,Z), hold
```

Det går också bra att själv styra vilka nivåkurvor som skall ritas. Låt oss illustrera detta med ett exempel.

Exempel 1: Om vi vill plotta nivåkurvorna $yx^4 + 3xy^4 = a$, för $a = 1, 5, 10$, så ger vi kommandona

```
>> f=@(x,y) y.*x.^4+3*x.*y.^4;
>> x=linspace(-3,3,21);y=linspace(-3,3,20);
>> [X Y] = meshgrid(x,y);
>> Z=f(X,Y);
>> contour(X,Y,Z,[1,5,10])
```

Om vi bara vill plotta en nivåkurva så måste man ange nivåvärdet två gånger t.ex. plottar vi nivåkurvan $yx^4 + 3xy^4 = 6$ med kommandot

```
>> contour(X,Y,Z,[6,6])
```

□

1.3 Funktioner av tre variabler

Det är lite svårare att visualisera funktioner av tre variabler. Vi kan inte rita grafen på samma sätt som för funktioner av en eller två variabler eftersom avstånd bara går att illustrera geometriskt i högst tre dimensioner. Det finns emellertid andra sätt. I avsnitt 1.4 skall vi se hur funktioner av tre variabler kan illustreras genom att plotta några nivåytor. Ett annat sätt är att använda färgen (istället för avstånd) för att ange funktionsvärdena utefter några plan i rummet. Vi kan då använda kommandot `slice`. Följande kommandon illustrerar funktionen $f(x, y, z) = xy - z$ utefter planen $x = -1.2, x = 0.8, x = 2, y = 2, z = -2$ och $z = -0.2$

```
>> f=@(x,y,z) x.*y-z;
>> x=linspace(-2,2,21);
>> y=linspace(-2,2,20);
>> z=linspace(-2,2,20);
>> [X,Y,Z] = meshgrid(x,y,z);
>> slice(X,Y,Z,f(X,Y,Z),[-1.2 .8 2],2,[-2 -.2])
```

För att se vilket funktionsvärde som hör till respektive färg så kan vi lägga in en färgskala i figuren med kommandot

```
>> colorbar
```

Om man inte har någon ”slice” i t.ex. z-led så anges en tom matris `[ ]` på motsvarande plats i `slice`-kommandot. T.ex. illustrerar följande kommando funktionens värde utefter planen $x = -1.2, x = 0.8$ och $y = 2$

```
>> slice(X,Y,Z,f(X,Y,Z),[-1.2 .8],2,[])
```

Vi kan också illustrera funktionsvärdena utefter andra ytor än plan t.ex. ger följande kommandon funktionens värden utefter ytan $z = x \sin y + y \cos x$

```
>> g=@(u,v) u.*sin(v)+u.*cos(v);
>> u=linspace(-2,2,21);v=linspace(-2,2,20);
>> [U,V] = meshgrid(u,v);W=g(U,V);
>> surf(U,V,W,f(U,V,W)), colorbar
```

Färgen i varje punkt (x, y, z) på ytan anger alltså värdet på funktionen $f(x, y, z)$. Alternativt kan vi plotta nivåkurvorna till funktionen på ytan;

```
>> shading interp
>> alpha(0.3)
>> contourslice(X,Y,Z,f(X,Y,Z),U,V,W)
```

Här använde vi bl.a. kommandot `alpha(0.3)`, som gör ytan delvis genomskinlig, för att nivåkurvorna skulle framträda lite tydigare (0 för helt transparant och 1 för helt solid).

1.4 Nivåytor

En nivåyta $f(x, y, z) = c$ är mängden av alla punkter (x, y, z) för vilket funktionsvärdet är c. Sådana nivåytor kan t.ex. användas för att beskriva temperaturnivåerna i ett tredimensionellt objekt. En funktionsyta $z = f(x, y)$ kan om man vill också betraktas som en nivåyta enl. $f(x, y) - z = 0$. För att plotta nivåytor i MATLAB använder vi kommandot `isosurface`.

Exempel 1: Sfären $x^2+y^2+z^2 = 0.5$ är exempel på en nivåyta och vi kan plotta den med följande kommandon

```
>> f=@(x,y,z) x.^2+y.^2+z.^2;
>> x=linspace(-2,2,21);
>> y=linspace(-2,2,20);
>> z=linspace(-2,2,20);
>> [X,Y,Z] = meshgrid(x,y,z);
>> isosurface(X,Y,Z,f(X,Y,Z),0.5)
```

Troligen ser det mer ut som en ellips än en cirkel när ni utför ovanstående kommandon. Skälet är naturligtvis att koordinataxlarna inte är dimensionerade på samma sätt. För att åstadkomma detta så ger vi kommandot

```
>> axis equal
```

Vi kan naturligtvis plotta nivåytor till andra funktioner i samma figur. Om vi t.ex. vill plotta nivåytan $x^2 + y^2 \sin z = 1$ i samma figur som sfären ovan så behöver vi inte beräkna om matriserna `X,Y` och `Z`

```
>> g=@(x,y,z) x.^2+y.^2.*sin(z);
>> T=g(X,Y,Z);
>> isosurface(X,Y,Z,T,1)
```

Vi kan plotta andra nivåytor till samma funktion utan att beräkna om matrisen `T`

```
>> isosurface(x,y,z,T,2)
```

Observera att föregående ytor ligger kvar i figuren när nästa nivåyta plottas, trots att man inte gett kommandot `hold on` (enklaste sättet att radera föregående plottar är att först stänga figurfönstret). Titta gärna på ytorna från lite olika vinklar genom att rotera figurfönstret. Välj t.ex. Rotate 3D i menyraden överst i figurfönstret. □

Om man vill kan man också plotta normalvektorer till en nivåyta. Man kan då använda kommandona `isonormals` och `quiver3` (detta kommando plottar vektorfält som vi skall studera närmare i kapitel 4).

Exempel 2: Antag att vi vill plotta nivåytan $x^2 + y^2 \sin z = 1$ och ett antal normalvektorer till ytan. Med matriserna `X,Y,Z` och `T` från föregående exempel så åstadkommer vi detta med följande kommandon

```
>> [p,q]=isosurface(X,Y,Z,T,1);
>> isosurface(X,Y,Z,T,1), hold on
>> N=isonormals(X,Y,Z,T,q); N=-N;
>> quiver3(q(:,1),q(:,2),q(:,3),N(:,1),N(:,2),N(:,3))
>> axis equal, hold off
```

Istället för att använda kommandot `isonormals` så kan man naturligtvis även beräkna normalvektorer $\mathbf{N}$ till en nivåyta $f(x,y,z) = c$ med hjälp av gradienten, ty gradienten ger ju vektorer som pekar i den riktning som funktionen växer mest och är (därför) vinkelräta mot nivåytorna. Vi lämnar åt den intresserade att i detta exempel själv beräkna gradienten och fundera ut vilka kommandon som plottar normalerna. □

1.5 Parametriserade kurvor

Vi kan plotta parametriserade kurvor såväl i planet som i rummet.

Exempel 1: Om vi vill plotta kurvan $x = \sqrt{|\cos 2t\,|}\,\cos t$, $y = \sqrt{|\cos 2t\,|}\,\sin t$, $0 \leq t \leq 2\pi$, i planet så ger vi följande kommandon

```
>> t=linspace(0,2*pi,200);
>> x=sqrt(abs(cos(2*t))).*cos(t);
>> y=sqrt(abs(cos(2*t))).*sin(t);
>> plot(x,y)
```
□

Exempel 2: Om vi vill plotta kurvan $x = e^t \cos 10t$, $y = e^t \sin 10t$, $z = t$, $-5 \leq t \leq 0$, i rummet så ger vi följande kommandon

```
>> t=linspace(-5,0,300);
>> x=exp(t).*cos(10*t);
>> y=exp(t).*sin(10*t);
>> z=t;
>> plot3(x,y,z)
```
□

1.6 Parametriserade ytor

Som tidigare noterats så kan en funktionsyta $z = f(x,y)$ betraktas som en nivåyta, men den kan också betraktas som en parametriserad yta. Det är då naturligt att använda x och y som parametrar enl. $x = u, y = v, z = f(u,v)$. Det är inte

heller så stor skillnad på att plotta funktionsytor och att plotta parametriserade ytor. Principen är den samma, beräkna matriser `X,Y,Z` och rita ytan med t.ex. `surf(X,Y,Z)`.

Exempel 1: Antag att vi vill plotta den yta som ges av parametriseringen

$$\begin{cases} x = (1 + \frac{1}{2}v\cos\frac{u}{2})\cos u \\ y = (1 + \frac{1}{2}v\cos\frac{u}{2})\sin u \\ z = \frac{1}{2}v\sin\frac{u}{2} \end{cases} \qquad 0 \le u \le 2\pi \ , \ -1 \le v \le 1$$

För att plotta ytan genererar vi först matriser för parametrarna u, v, och beräknar sedan matriser för x, y, z.

```
>> u=linspace(0,2*pi,20);v=linspace(-1,1,21);
>> [U,V]=meshgrid(u,v);
>> X=(1+V.*cos(U/2)/2).*cos(U);
>> Y=(1+V.*cos(U/2)/2).*sin(U);
>> Z=V.*sin(U/2)/2;
>> surf(X,Y,Z)
```

Ytan är ett s.k. Möbiusband, vilket ofta anges som exempel på en yta som inte är orienterbar. Orienterbara ytor har två sidor, som mer allmänt kallas för den positiva respektive negativa sidan, men som ibland (när det är tydligt vad man menar) kan beskrivas med t.ex. ovansidan/undersidan eller inuti/utanpå. Möbiusbandet är däremot inte orienterbart eftersom den bara har en ”sida”. □

Exempel 2: I avsnitt 1.4 såg vi hur man kan plotta en sfär genom att betrakta den som en nivåyta. Man kan också plotta en sfär genom att beskriva den som en parametriserad yta. Sfären $x^2 + y^2 + z^2 = 4$ kan t.ex. beskrivas med;

$$\begin{cases} x = 2\sin u \cdot \cos v \\ y = 2\sin u \cdot \sin v \\ z = 2\cos u \end{cases} \qquad 0 \le u \le \pi \ , \ 0 \le v \le 2\pi$$

och kan därför plottas med följande kommandon;

```
>> u=linspace(0,pi,20);v=linspace(0,2*pi,21);
>> [U,V]=meshgrid(u,v);
>> X=2*sin(U).*cos(V);
>> Y=2*sin(U).*sin(V);
>> Z=2*cos(U);
>> surf(X,Y,Z)
```

Vi kan här ändra parameterområdet om vi bara vill plotta en viss del av sfären;

```
>> u=linspace(pi/4,3*pi/4,20);v=linspace(-3*pi/4,pi,21);
```

```
>> [U,V]=meshgrid(u,v);
>> X=2*sin(U).*cos(V);
>> Y=2*sin(U).*sin(V);
>> Z=2*cos(U);
>> surf(X,Y,Z)
```

I ovanstående parametrisering av sfären så är radien hela tiden 2. Om vi låter radien variera med värdena på vinklarna u och v så får vi en annan yta kring origo. Pröva t.ex. följande kommandon;

```
>> u=linspace(0,pi,20);v=linspace(0,2*pi,21);
>> [U,V]=meshgrid(u,v);
>> R=(2+sin(2*U))./(2+cos(V));
>> X=R.*sin(U).*cos(V);
>> Y=R.*sin(U).*sin(V);
>> Z=R.*cos(U);
>> surf(X,Y,Z)
```

När man plottar ytor med hjälp av bl.a. `surf` så är det standard att ytan färgläggs med en färg som beror på höjden över xy-planet dvs. värdet på z i respektive punkt. Om man vill kan man också själv styra färgerna på varje liten ytbit. Följande kommando väljer t.ex. färgerna slumpvis;

```
>> surf(X,Y,Z,rand(size(Z)))
```

Pröva även att jämna ut färgerna med;

```
>> shading interp
```

□

Exempel 3: Om en kurva i xz-planet med parametrisering $\mathbf{r} = x(t)\mathbf{i}+z(t)\mathbf{k}$, $a \leq t \leq b$, roterar kring z-axeln skapas en s.k. rotationsyta. En sådan parametriseras naturligt genom;

$$\begin{cases} x = x(t)\cos v \\ y = x(t)\sin v \\ z = z(t) \end{cases} \qquad a \leq t \leq b\ ,\ 0 \leq v \leq 2\pi$$

Notera att parametriseringen av sfären i föregående exempel bygger på denna form av parametrisering (där kurvan är halvcirkeln $\mathbf{r} = \cos t\mathbf{i} + \sin t\mathbf{k}$, $-\frac{\pi}{2} \leq t \leq \frac{\pi}{2}$). Låt oss titta på några fler rotationsytor. Låt oss t.ex. plotta den rotationsyta som bildas då kurvan $\mathbf{r} = t\sin t\mathbf{i} + (t + \cos t)\mathbf{k}$, $0 \leq t \leq 3$ roterar kring z-axeln;

```
>> t=linspace(0,3,20);v=linspace(0,2*pi,21);
>> [T,V]=meshgrid(t,v);
```

```
>> X=T.*sin(T).*cos(V);
>> Y=T.*sin(T).*sin(V);
>> Z=T+cos(T);
>> surf(X,Y,Z)
```

Naturligtvis kan vi på liknande sätt beskriva rotationsytor kring x-axeln eller y-axeln. Låt oss t.ex. plotta den rotationsyta som bildas då cirkeln $(x-3)^2+y^2=4$ roterar kring y-axeln. Cirkeln parametriseras naturligt av $\mathbf{r} = (3+2\cos u)\mathbf{i} + 2\sin u\mathbf{j}$, $0 \leq u \leq 2\pi$, så rotationsytan kan plottas med;

```
>> u=linspace(0,2*pi,20);v=linspace(0,2*pi,21);
>> [U,V]=meshgrid(u,v);
>> X=(3+2*cos(U)).*cos(V);
>> Y=2*sin(U);
>> Z=(3+2*cos(U)).*sin(V);
>> surf(X,Y,Z)
>> axis equal
```

Den yta som bildas på detta sätt liknar en badring och kallas i matematiken för en torus.

Med koordinatbyten är det inte heller så svårt att åstadkomma rotationsytor kring andra fixa axlar. Mer allmänt kan vi bilda tubliknande ytor kring kurvor sådana att om vi snittar en sådan tub vinkelrät mot kurvans tangentriktning så bildas cirklar. Antag t.ex. att $\mathbf{r} = \mathbf{r}(t), a \leq t \leq b$, är en kurva i rummet och att $\mathbf{N}(t)$ och $\mathbf{B}(t)$ är ortogonala enhetsvektorer som är vinkelräta mot kurvans tangentvektor $\mathbf{r}'(t)$, för varje t i intervallet $[a,b]$. Då bildar $\mathbf{r}(t,v) = \mathbf{r}(t) + r\cos(v)\mathbf{N}(t) + r\sin v\mathbf{B}(t), a \leq t \leq b, 0 \leq v \leq 2\pi$, en parametrisering av en sådan tub kring kurvan, där snittkurvorna är cirklar med radie r. Vi kan också låta tubens tjocklek variera utefter kurvan genom att låta radien r beror på t. Låt oss titta på ett exempel.

Vi kan bilda en tub kring helixen $\mathbf{r}(t) = \cos t\mathbf{i} + t\mathbf{j} + \sin 2\mathbf{k}, 0 \leq t \leq 10$ genom parametriseringen $\mathbf{r}(t,v) = \mathbf{r}(t) + r(t)\cos(v)\mathbf{N}(t) + r(t)\sin v\mathbf{B}(t), 0 \leq t \leq 10, 0 \leq v \leq 2\pi$, där $r(t) = 0.2+0.1\cos t$, $\mathbf{N}(t) = \cos t\mathbf{i}+\sin t\mathbf{k}$ och $\mathbf{B}(t) = \sin t\mathbf{i}+\mathbf{j}-\cos t\mathbf{k}$. Notera här att $\mathbf{N}(t)$ och $\mathbf{B}(t)$ är ortogonala enhetsvektorer som är vinkelräta mot $\mathbf{r}'(t)$. I MATLAB plottar vi därför tuben med följande kommandon;

```
>> t=linspace(0,10,60);v=linspace(0,2*pi,61);
>> [T,V]=meshgrid(t,v);
>> R=0.2+0.1*cos(T);
>> X=cos(T)+R.*cos(V).*cos(T)+R.*sin(V).*sin(T);
>> Y=T+R.*sin(V);
>> Z=sin(T)+R.*cos(V).*sin(T)-R.*sin(V).*cos(T);
```

```
>> surf(X,Y,Z)
>> axis equal
```
□

I ovanstående parametriseringar var parameterområdena kvadratiska. I nästa exempel beskriver vi några sätt att illustrera ytor där parameterområdet inte är kvadratiskt.

Exempel 4: Låt oss rita den yta som ges av $x = u\cos v, y = u\sin v, z = uv, u^2 \leq v \leq 4$. Vi väljer en rektangel som omfattar området: $-2 \leq u \leq 2, \quad 0 \leq v \leq 4$ och plottar som i tidigare exempel, men beräknar och ritar samtidigt bilden av randkurvan där $v = u^2$.

```
>> u=linspace(-2,2,20);v=linspace(0,4,21);
>> [U,V]=meshgrid(u,v);
>> X=U.*cos(V);Y=U.*sin(V);Z=U.*V;
>> surf(X,Y,Z), hold on
>> t=linspace(-2,2,200);
>> xr=t.*cos(t.^2);yr=t.*sin(t.^2);zr=t.^3;
>> plot3(xr,yr,zr,'-k'), hold off
```

Om vi vill kan vi ta bort den del av ytan som motsvarar parametervärden som inte uppfyller villkoret $u^2 \leq v$. I så fall byter vi ut de värden i matrisen `Z` vars motsvarande element i matriserna `U` och `V` är sådana att $u^2 > v$, mot värdet `NaN` (Not a Number). När MATLAB plottar så utesluts nämligen linjer till sådana värden.

```
>> Z(find(U.^2>V))=NaN;
>> surf(X,Y,Z), hold on
>> plot3(xr,yr,zr,'-k'), hold off
```

Kanten på ytan blir då lite hackig eftersom MATLAB utesluter hela meshbitar. En snyggare plot av ytan med bättre kant kan åstadkommas med ett parameterbyte. Vi vill i så fall byta parametrarna u, v mot två nya parametrar, säg s, t, sådana att parameterområdet i uv-planet motsvaras av en rektangel i st-planet. I detta fall kan vi t.ex. sätta;

$$\begin{cases} u = t \\ v = 4s + (1-s)t^2 \end{cases}$$

I så fall motsvaras området $u^2 \leq v \leq 4$ i uv-planet en-entydigt av rektangeln $-2 \leq t \leq 2, 0 \leq s \leq 1$ i st-planet. Om vi nu istället gör en mesh i st-planet och plottar motsvarande del av ytan ovan så får vi;

```
>> t=linspace(-2,2,60);s=linspace(0,1,61);
>> [T,S]=meshgrid(t,s);
```

```
>> U=T; V=4*S+(1-S).*T.^2;
>> X=U.*cos(V);Y=U.*sin(V);Z=U.*V;
>> surf(X,Y,Z), hold on
>> shading interp
>> plot3(xr,yr,zr,'-k'), hold off
```
□

Om man vill kan man också plotta normalvektorer till en parametriserad yta. Man kan då använda kommandona `surfnorm` och `quiver3`.

Exempel 5: Följande kommandon plottar funktionsytan $z = xe^{-x^2-y^2}$ och ett antal normalvektorer till ytan.

```
>> x=linspace(-2,2,20);y=linspace(-1,1,21);
>> [X,Y]=meshgrid(x,y);Z=X.*exp(-X.^2-Y.^2);
>> [Nx,Ny,Nz]=surfnorm(X,Y,Z);
>> quiver3(X,Y,Z,Nx,Ny,Nz),hold on
>> surf(X,Y,Z), hold off
>> shading interp, axis equal
```

Istället för att använda kommandot `surfnorm` så kan man naturligtvis även beräkna normalvektorer $\mathbf{N}$ till en parametriserad yta $\mathbf{r} = \mathbf{r}(u, v)$, där $\mathbf{r}(u, v) = (x(u, v), y(u, v), z(u, v))$, med hjälp av formeln $\mathbf{N} = \mathbf{r_u} \times \mathbf{r_v}$. Men eftersom `cross` som beräknar vektorprodukter i MATLAB inte utför elementvisa operationer på ”vanligt” sätt så blir detta lite mer komplicerat. Vi lämnar därför detta till läsaren själv att fundera ut (i mån av tid). □

2 Ickelinjära ekvationssystem och optimering

2.1 Newtons metod för system av ekvationer

I detta avsnitt skall vi studera Newtons metod för lösning av ekvationssystem. Den som vill kan också läsa om metoden i avsnitt 13.6 i kursboken *Calculus* (av Adams och Essex). Låt oss t.ex. betrakta ett system av typen

$$\begin{cases} f(x,y) = 0 \\ g(x,y) = 0 \end{cases}$$

Om funktionerna $f(x,y)$ och $g(x,y)$ är linjära kan vi använda oss av kända metoder från linjär algebra kursen för att lösa systemet. Situationen är dock (i allmänhet) betydligt mer komplicerad om funktionerna inte är linjära.

Låt oss börja med att titta på härledningen av Newtons metod i en variabel dvs för lösning av en ekvation med en obekant. Antag att $\tilde{x}$ ligger nära en lösning x^* till ekvationen $f(x) = 0$. Om vi Taylorutvecklar $f(x)$ kring $\tilde{x}$ t.o.m. första ordningen får vi $f(x) \approx f(\tilde{x}) + f'(\tilde{x})(x - \tilde{x})$. Denna approximation stämmer bra i en nära omgivning av $\tilde{x}$ och då speciellt för $x = x^*$ vilket ger oss att

$$0 = f(x^*) \approx f(\tilde{x}) + f'(\tilde{x})(x^* - \tilde{x}) \quad \Leftrightarrow$$

$$f'(\tilde{x})(x^* - \tilde{x}) \approx -f(\tilde{x})$$

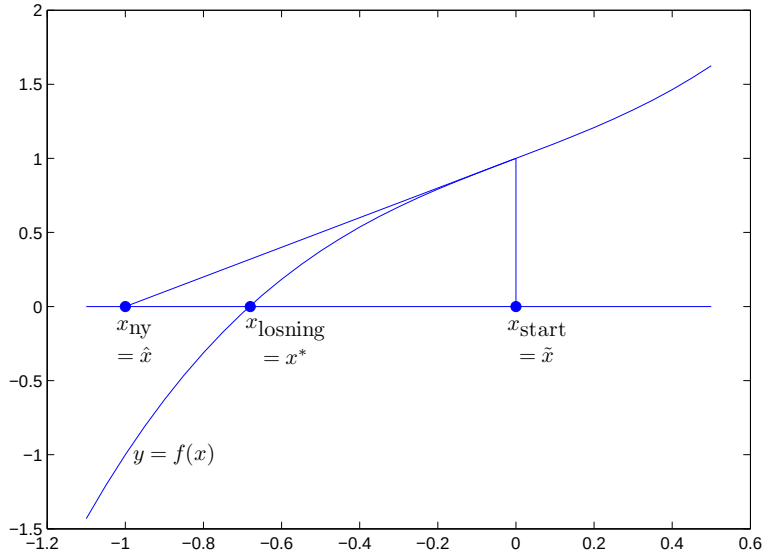
Om vi löser ut x^* får vi att $$x^* \approx \tilde{x} - \frac{f(\tilde{x})}{f'(\tilde{x})}$$

Felet i denna approximation är av storleksordningen $|x^* - \tilde{x}|^2$ (ty nästa term i Taylorutvecklingen är av den storleksordningen), vilket är betydligt mindre än det ursprungliga felet $|x^* - \tilde{x}|$. Värdet på

$$\hat{x} = \tilde{x} - \frac{f(\tilde{x})}{f'(\tilde{x})}$$

ger därför en bättre approximation av x^* än vad $\tilde{x}$ gav. Såvida man inte känner sig nöjd med denna förbättrade approximation så är det högst naturligt att tänka sig upprepa proceduren för att ytterligare förfina approximationen. Newtons metod innebär att man på detta sätt successivt itererar sig närmare en lösning på ekvationen.

Metoden kan också åskådliggöras geometriskt. Lösningen på ekvationen $f(x) = 0$ är det x-värde för vilket grafen $y = f(x)$ skär x-axeln. När vi ersätter $f(x)$ med dess Taylorutveckling i $\tilde{x}$ t.o.m. första graden i ekvationen så innebär det att vi istället undersöker var tangenten till $y = f(x)$ i punkten $(\tilde{x}, f(\tilde{x}))$ skär x-axeln. Denna skärningspunkt blir sedan ny utgångspunkt för nästa iteration. Följande figur illustrerar ett steg i Newtons metod.



Vi kan nu på liknande sätt härleda en metod för att lösa system av ekvationer. Antag att vi vill lösa system av typen

$$\begin{cases} f(x,y) = 0 \\ g(x,y) = 0 \end{cases} \tag{1}$$

Om $(\tilde{x}, \tilde{y})$ ligger nära en lösning (x^*, y^*) till ekvationssystemet så har vi

$$\begin{cases} 0 = f(x^*, y^*) \approx f(\tilde{x}, \tilde{y}) + f'_x(\tilde{x}, \tilde{y})(x^* - \tilde{x}) + f'_y(\tilde{x}, \tilde{y})(y^* - \tilde{y}) \\ 0 = g(x^*, y^*) \approx g(\tilde{x}, \tilde{y}) + g'_x(\tilde{x}, \tilde{y})(x^* - \tilde{x}) + g'_y(\tilde{x}, \tilde{y})(y^* - \tilde{y}) \end{cases}$$

Detta är ett linjärt ekvationssystem i de obekanta x^* och y^* (om vi betraktar $(\tilde{x}, \tilde{y})$ som känd). Sambanden kan också uttryckas på matrisform enl.

$$\begin{bmatrix} f'_x(\tilde{x}, \tilde{y}) & f'_y(\tilde{x}, \tilde{y}) \\ g'_x(\tilde{x}, \tilde{y}) & g'_y(\tilde{x}, \tilde{y}) \end{bmatrix} \begin{bmatrix} x^* - \tilde{x} \\ y^* - \tilde{y} \end{bmatrix} \approx \begin{bmatrix} -f(\tilde{x}, \tilde{y}) \\ -g(\tilde{x}, \tilde{y}) \end{bmatrix}$$

Felet i denna approximation är av storleksordningen $||(x^*, y^*) - (\tilde{x}, \tilde{y})||^2$, vilket är avsevärt mindre än $||(x^*, y^*) - (\tilde{x}, \tilde{y})||$ som vi hade från början.

Lösningen $(\hat{x}, \hat{y})$ på ekvationen

$$\begin{bmatrix} f'_x(\tilde{x},\tilde{y}) & f'_y(\tilde{x},\tilde{y}) \\ g'_x(\tilde{x},\tilde{y}) & g'_y(\tilde{x},\tilde{y}) \end{bmatrix} \begin{bmatrix} \hat{x}-\tilde{x} \\ \hat{y}-\tilde{y} \end{bmatrix} = \begin{bmatrix} -f(\tilde{x},\tilde{y}) \\ -g(\tilde{x},\tilde{y}) \end{bmatrix} \tag{2}$$

bör därför ge en bättre approximation till lösningen (x^*, y^*) än vad $(\tilde{x}, \tilde{y})$ gav.

Geometriskt innebär metoden följande: Lösningen på systemet (1) är den punkt (x^*, y^*) i vilket funktionsytorna $z = f(x, y)$ och $z = g(x, y)$ skär varandra i xy-planet. När vi ersätter $f(x, y)$ och $g(x, y)$ med respektive Taylorutveckling i $(\tilde{x}, \tilde{y})$ t.o.m. första graden i systemet så innebär det att vi istället undersöker var tangentplanen till $z = f(x, y)$ och $z = g(x, y)$, där $(x, y) = (\tilde{x}, \tilde{y})$, skär varandra i xy-planet. Denna skärningspunkt blir sedan ny utgångspunkt för nästa iteration och man upprepar proceduren tills båda funktionsvärdena är tillräckligt små.

Låt oss titta på ett exempel;

Exempel 1: Antag att vi vill hitta en lösning (x^*, y^*) till ekvationssystemet

$$\begin{cases} xy(x-y) = 1 \\ x^3y^2 + x^2 + y^4 = 3 \end{cases}$$

Ett sätt att försöka hitta lämpliga startvärden för Newtons metod är att plotta nivåkurvorna $xy(x-y)-1=0$ och $x^3y^2+x^2+y^4-3=0$ på något område. Låt oss pröva följande;

```
>> f=@(x,y) x.*y.*(x-y)-1; g=@(x,y) x.^3.*y.^2+x.^2+y.^4-3;
>> x=linspace(-3,3,30);y=linspace(-3,3,31);
>> [X,Y]=meshgrid(x,y);
>> contour(X,Y,f(X,Y),[0 0],'r'), hold on
>> contour(X,Y,g(X,Y),[0 0],'b'), grid, hold off
```

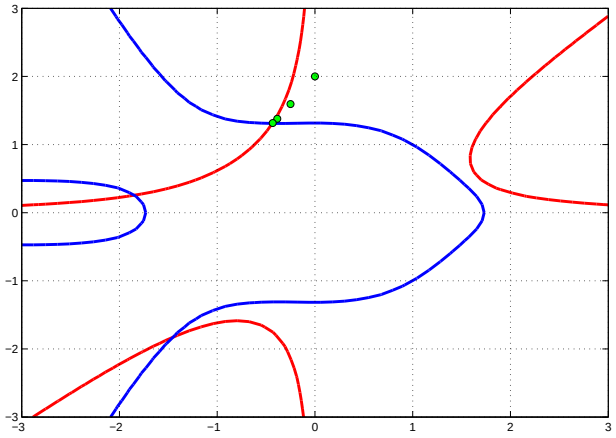
Vi inser då att det bl.a. finns en lösning (skärningspunkt) i närheten av $(0, 2)$ och använder Newtons metod för att förbättra denna (grova) approximation till lösningen.

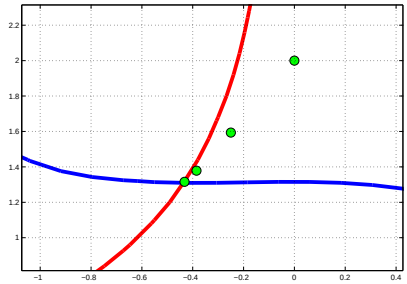
```
>> fx=@(x,y) 2*x.*y-y.^2; fy=@(x,y) x.^2-2*x.*y;
>> gx=@(x,y) 3*x.^2.*y.^2+2*x; gy=@(x,y) 2*x.^3.*y+4*y.^3;
>> x1=0,y1=2
>> J=[fx(x1,y1) fy(x1,y1);gx(x1,y1) gy(x1,y1)];
>> ny=[x1;y1]+J\[-f(x1,y1);-g(x1,y1)]
>> x1=ny(1); y1=ny(2);
```

Här har vi löst det linjära ekvationssystemet (2) med hjälp av backslash-kommandot (ekvationsystem av typen $Ax = b$ kan lösas i MATLAB med kommandot `x=A\b`). Ett steg med Newtons metod ger (enligt ovan) att $x^* \approx -0.25$ och $y^* \approx 1.59$. Om vi vill stega oss närmare den exakta lösningen så är det bara att upprepa de tre sista kommandoraderna ovan.

```
>> J=[fx(x1,y1) fy(x1,y1);gx(x1,y1) gy(x1,y1)];
>> ny=[x1;y1]+J\[-f(x1,y1);-g(x1,y1)]
>> x1=ny(1); y1=ny(2);
```

vilket ger att $x^* \approx -0.38$ och $y^* \approx 1.38$. Man kan sedan upprepa iterationen ytterligare några gånger tills man känner sig nöjd (t.ex. med en liten snurra). Nedanstående figurer illustrerar hur Newtons metod ger punkter som successivt närmar sig lösningen (den högra bilden är en uppförstorning av den vänstra).





. □

Om bara startapproximationen är tillräckligt bra kommer Newtons metod att konvergera mot en lösning väldigt snabbt ty konvergenshastigheten är ”kvadratisk”. Det skall dock påpekas att en dålig startapproximation också kan leda till att Newtons metod divergerar dvs. inte närmar sig någon lösning alls.

Låt oss nu titta lite mer allmänt på Newtons metod för system med n ekvationer och n obekanta;

$$\begin{cases} f_1(x_1, ..., x_n) = 0 \\ \quad\vdots \\ f_n(x_1, ..., x_n) = 0 \end{cases} \tag{3}$$

Om vi sätter

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{f} = \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}, \quad \mathbf{0} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

så kan systemet (3) kortare skrivas

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{4}$$

Antag att vi på något sätt erhållit en approximativ lösning $\mathbf{x}_k$ till detta system, som vi vill förbättra. I analogi med ovan ersätter vi då $\mathbf{f}$ med sin linjärisering i $\mathbf{x}_k$ (motsvarar Taylorutveckling av komponentfunktionerna t.o.m. ordning 1) dvs.

$$\mathbf{L}(\mathbf{x}) = \mathbf{f}(\mathbf{x}_k) + \mathbf{Df}(\mathbf{x}_k)(\mathbf{x} - \mathbf{x}_k)$$

där $\mathbf{Df}(\mathbf{x})$ är Jacobimatrisen för funktionen $\mathbf{f}(\mathbf{x})$, innehållande de partiella derivatorna (se avsnitt 12.6 i *Calculus*, av Adams och Essex). Lösningen till det erhållna linjära ekvationssystemet

$$\mathbf{L}(\mathbf{x}) = \mathbf{0} \tag{5}$$

ger oss då en ny (och förhoppningsvis bättre) approximation $\mathbf{x}_{k+1}$ av lösningen till det ursprungliga systemet (4).

Om vi inför $\mathbf{h} = \mathbf{x} - \mathbf{x}_k$ kan (5) skrivas

$$\mathbf{Df}(\mathbf{x}_k)\mathbf{h} = -\mathbf{f}(\mathbf{x}_k)$$

Detta är ett linjärt ekvationssystem som vi löser med avseende på $\mathbf{h}$ (förslagsvis med hjälp av backslash-kommandot \ i MATLAB). Lösningen $\mathbf{h}_k$ på detta system ger den vektor med vilket vi skall stega oss fram till nästa punkt dvs.

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{h}_k$$

Vi kan sedan upprepa processen, och förbättra approximationen ytterligare, genom att istället utgå från $\mathbf{x}_{k+1}$ som startapproximation.

Vi behöver nödvändigtvis inte för hand exakt bestämma de partiella derivatorna i Jacobimatrisen $\mathbf{Df}(\mathbf{x})$ utan det duger (för våra ändamål) bra med approximationer med hjälp av differenskvoter. Den j:te kolonnen i $\mathbf{Df}(\mathbf{x})$ kan t.ex. ersättas med

$$\frac{\mathbf{f}(\mathbf{x} + \delta\mathbf{e}_j) - \mathbf{f}(\mathbf{x} - \delta\mathbf{e}_j)}{2\delta}$$

för något lämpligt litet tal δ (förslagsvis 10^{-5}), och där $\mathbf{e}_j$ är j:te enhetsvektorn.

Låt oss illustrera detta med ett exempel.

Exempel 2: Antag att vi vill hitta en lösning till ekvationssystemet

$$\begin{cases} y^2 + z^2 &= 3 \\ x^2 + z^2 &= 2 \\ x^2 - z &= 0 \end{cases}$$

Vi börjar då med att skapa följande funktion;

```
>> f=@(x) [x(2).^2+x(3).^2-3; x(1).^2+x(3).^2-2; x(1).^2-x(3)];
```

Låt oss utgå från $\mathbf{x}_1 = (2, 1, 0)$ som startapproximation i Newtons metod. Vi börjar med att beräkna Jacobimatrisen för $\mathbf{f}$ i punkten $\mathbf{x}_1$. Med en liten snurra beräknar vi Jacobimatrisen (`J`), kolonn för kolonn, med differenskvoter;

```
>> x=[2;1;0];
>> delta=10^(-5);
>> J=zeros(3,3);
>> for j=1:3 ...
       ej=zeros(3,1);
       ej(j,1)=1;
       J(:,j)=(f(x+delta*ej)-f(x-delta*ej))/(2*delta);
   end
```

Sedan stegar vi oss fram till nästa approximation med följande kommandon;

```
>> h=J\(-f(x));
>> x=x+h
```

Notera att vi här tilldelar variabeln `x` den nya approximationen och därmed skriver över den föregående approximationen.

Man kan naturligtvis upprepa processen ovan (t.ex. med en liten snurra) för att förbättra approximationen ytterligare. Utför ytterligare några steg och försök avgöra vilken lösning vi närmar oss. □

2.2 Andra metoder för ekvationslösning

I MATLAB löser man normalt linjära ekvationssystem med s.k. vänsterdivision (kommandot \ ). Det finns även andra kommandon som kan vara användbara vid lösning av sådana system t.ex. `rref, inv, det, rank, lu, qr, chol`.

Om ekvationssystemet inte är linjärt så kan man istället använda kommandot `fsolve`. Kommandot finns inte inbyggt i MATLABs grundpaket utan kräver ett speciellt programpaket, en sk. toolbox, som är till för att lösa optimeringsproblem (Optimization Toolbox, ge kommandot `help optim` för att se tillgängliga kommandon). Chalmers har en MATLAB-licens för denna toolbox (och en mängd andra), så om du jobbar på en Chalmersdator bör det inte vara några några problem att använda kommandot `fsolve` och andra funktioner i detta programpaket.

För linjära ekvationssytem vet vi att det antingen saknas lösningar, finns en enda lösning eller så har systemet oändligt många lösningar. Till ickelinjära ekvationssystem kan det dock finnas fler än en lösning utan att det nödvändigtvis finns oändligt många t.ex. två, tre eller fler lösningar.

Programmet `fsolve` kräver som indata (precis som Newtons metod) en startgissning dvs. en första approximation av lösningen. För 2×2-system kan vi (som i föregående avsnitt) ev. bilda oss en uppfattning om hur många lösningar det finns (och vilka dessa är), på ett visst område, genom att plotta nivåkurvor. Varje skärningspunkt mellan kurvorna motsvarar en lösning till ekvationssystemet. Om man inte har någon ytterligare information (om t.ex. det bakomliggande problemet och vad variablerna representerar) kan det dock vara svårt att uppskatta i vilket område man skall börja söka efter lösningar (endast genom att studera ekvationerna).

Exempel 1: Betrakta ekvationssystemet $\begin{cases} x^2 + y = 2 \\ x - y^3 = 1 \end{cases}$

Låt oss undersöka om det måhända kan finnas några lösningar i kvadraten $-4 \leq x \leq 4, -4 \leq y \leq 4$.

```
>> f=@(x,y) x.^2+y-2; g=@(x,y) x-y.^3-1;
>> [X,Y]=meshgrid(-4:0.1:4);
>> contour(X,Y,f(X,Y),[0 0],'r'), hold on
>> contour(X,Y,g(X,Y),[0 0],'b'), grid, hold off
```

Vi inser att det finns två skärningspunkter och därmed två lösningar på ekvationssystemet i kvadraten. För att närmare bestämma den lösning som ligger i närheten av punkten $(1, 0.2)$ kan vi nu ge följande kommandon;

```
>> fun=@(x) [f(x(1),x(2)),g(x(1),x(2))];
>> losn=fsolve(fun,[1,0.2])
```

Observera här att vi måste skriva `x(1)` och `x(2)` istället för x och y när vi definierar den anonyma funktionen.

Om man vill kan man få MATLAB att redovisa detaljer om de olika stegen som `fsolve` utför för att hitta lösningen (eller snarare en förbättrad approximation av lösningen). Ge i så fall följande kommando;

```
>> options=optimset('Display','iter');
>> losn=fsolve(fun,[1,0.2],options)
```

Algoritmen innehåller villkor för när iterationen skall avbrytas, och om man vill kan man förändra dessa villkor. Den som vill kan läsa mer om detta t.ex. genom att ge kommandot `help fsolve`. □

Vi kan även lösa system med fler än två ekvationer och obekanta med `fsolve`. Om ekvationssytemet består av tre ekvationer och tre obekanta så finns (precis som för 2×2-system) också en möjlighet att tolka lösningarna geometriskt. Varje ekvation representerar då en nivåyta och ev. lösningar motsvarar gemensamma skärningspunkter i de tre ytorna (dvs. punkter som tillhör alla de tre ytorna). Det kan dock vara svårt att lokalisera lösningarna grafiskt.

Exempel 2: Betrakta ekvationssystemet

$$\begin{cases} 2x - 6x^2z &= 0 \\ 2y - 3y^2z &= 0 \\ 2x^3 + y^3 &= 10 \end{cases}$$

För att få en uppfattning om ev. lösningar till systemet prövar vi med att plotta de delar av nivåytorna $2x - 6x^2z = 0, 2y - 3y^2z = 0$ och $2x^3 + y^3 = 10$ som ligger i området $-4 \leq x \leq 4, -4 \leq y \leq 4, -4 \leq z \leq 4$.

```
>> f1=@(x,y,z) 2*x-6*x.^2.*z;
>> f2=@(x,y,z) 2*y-3*y.^2.*z;
>> f3=@(x,y,z) 2*x.^3+y.^3;
>> [X,Y,Z]=meshgrid(-4:0.3:4);
>> clf
>> isosurface(X,Y,Z,f1(X,Y,Z),0)
>> isosurface(X,Y,Z,f2(X,Y,Z)+5,5)
>> isosurface(X,Y,Z,f3(X,Y,Z),10)
```

Här valde vi att addera en femma i den andra ekvationens båda led. Med detta lilla trick kunde vi på ett enkelt sätt se till att de tre ytorna plottas med olika färg (olika nivåer i samma figur ger olika färg). Av figuren kan det dock vara svårt att avgöra antalet skärningspunkter och därmed hur många lösningar som ekvationssystemet har och det kan vara ännu knepigare att grafiskt bestämma närmevärden till lösningarna. Genom att vända och vrida på figuren inser man dock i detta fall att det bör finnas tre lösningar; nära punkterna $(2, 0, 0), (0, 2, 0)$ resp. $(1, 2, 0)$. Låt oss bestämma dessa lite bättre med hjälp av `fsolve`;

```
>> f=@(x) [f1(x(1),x(2),x(3)),
           f2(x(1),x(2),x(3)),
           f3(x(1),x(2),x(3))]
>> losn1=fsolve(f,[2,0,0])
>> losn2=fsolve(f,[0,2,0])
>> losn3=fsolve(f,[1,2,0])
```

□

Det går även bra att använda kommandot `fsolve` för att lösa system med fler än tre ekvationer och obekanta.

2.3 Gradientmetoden för optimering

Vi skall börja detta avsnitt med att undersöka den geometriska betydelsen av gradienten till en funktion. Låt oss t.ex. betrakta funktionen

$$f(x,y) = \frac{60 - 2y^2 - 4xy - x^4}{20}$$

Vi börjar med att generera två koordinatmatriser `X` och `Y`;

```
>> x=linspace(-2,2,20);y=linspace(-1,3,21);
>> [X,Y]=meshgrid(x,y);
```

och beräknar sedan funktionsvärdena;

```
>> f=@(x,y) (60-2*y.^2-4*x.*y-x.^4)/20;
>> Z=f(X,Y);
```

Eftersom $f'_x(x,y) = (-y - x^3)/5$ och $f'_y(x,y) = (-y - x)/5$ kan vi beräkna gradienten i varje punkt i rutnätet med följande kommandon;

```
>> fx=@(x,y) (-y-x.^3)/5; fy=@(x,y) (-y-x)/5;
>> GX=fx(X,Y);GY=fy(X,Y);
```

Alternativt kan vi beräkna de partiella derivatorna (approximativt) med differenskvoter;

```
>> h=10^(-5);
>> fx=@(x,y) (f(x+h,y)-f(x-h,y))/(2*h);
>> fy=@(x,y) (f(x,y+h)-f(x,y-h))/(2*h);
>> GX=fx(X,Y);GY=fy(X,Y);
```

Vi kan sedan plotta nivåkurvor och gradientvektorer i samma figur med kommandot;

```
>> contour(X,Y,Z), hold on, quiver(X,Y,GX,GY), hold off
```

Om man dessutom ger kommandot `axis equal` blir vinklarna korrekta och man ser att gradientvektorerna är vinkelräta mot nivåkurvorna. Om man använder `quiver` som ovan så skalar MATLAB om vektorernas längd för att ge en bättre illustration av hur de varierar på området. Vektorernas längd relativt varandra stämmer dock. Om man trots allt vill plotta gradientvektorernas faktiska längd så kan man ge kommandot `quiver(X,Y,GX,GY,0)` (pröva gärna och jämför).

Gradienten är en vektor som pekar i den riktning funktionsvärdena ökar mest (se t.ex. avsnitt 12.7 i *Calculus*, av Adams och Essex). Kraftigare ökning betyder längre gradientvektor. Vi kan använda denna observation för stega oss fram mot större och större funktionsvärden tills vi ev. hittar ett maximum. Följande schema beskriver denna idé;

(1) Välja en startpunkt (x_1, y_1).

(2) Beräkna gradienten $\nabla f(x_1, y_1)$.

(3) Stega fram till nästa punkt (x_2, y_2) genom;

$$(x_2, y_2) = (x_1, y_1) + r \cdot \nabla f(x_1, y_1)$$

där r är ett lämpligt (litet) tal.

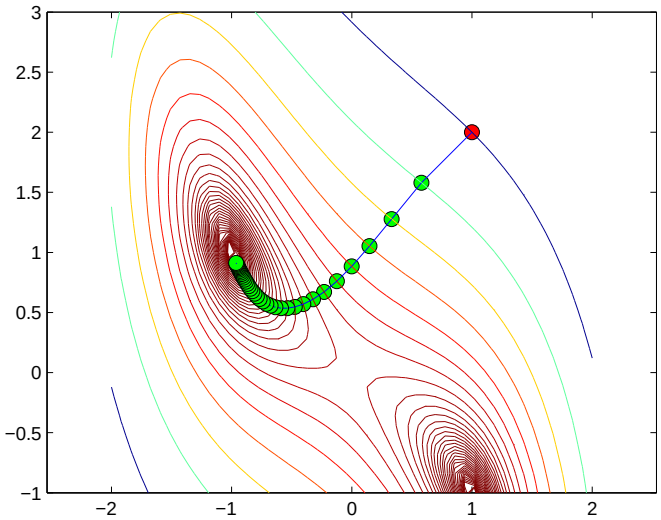
(Söker vi minimum skall vi istället gå i riktningen $-\nabla f(x_1, y_1)$)

(4) Beräkna differensen $f(x_2, y_2) - f(x_1, y_1)$.

(Troligen kommer $f(x_2, y_2)$ att vara större än $f(x_1, y_1)$ om inte r är för stort. Faktorn r kan behövas eftersom funktionen kan växa väldigt brant. Med $r = 1$ kan man missa maxpunkten helt eller kanske hoppa fram och tillbaka. Det finns metoder med vilket man kan bestämma bra val på r men vi fördjupar oss inte i det här)

(5) Upprepa (3) och (4) tills ändringen i funktionsvärde i (4) är tillräckligt liten.

Denna metod för att hitta lokala max/min-punkter kallas ofta för *gradientmetoden* (men benämns mer vanligen på engelska med "*method of steepest decent*"). Nedanstående figur illustrerar hur man med denna metod successivt kan närma sig en extrempunkt (startpunkten är markerad med rött och efterföljande steg med grönt).



Låt oss utföra några steg i gradientmetoden på funktionen ovan (med litet positivt värde på r) och se hur vi närmar oss en maxpunkt. För att tydligt följa vad som händer i varje steg så illustrerar vi stegen med plottar. Vi börjar med att plotta grafen till funktionen;

```
>> surf(X,Y,Z),alpha(0.2),shading interp,hold on
```

Vi har här valt att hålla kvar figurfönstret med kommandot `hold on` så att vi kan plotta fler saker i samma figur (se nedan). Vi har också valt att göra ytan lite transparant så att kommande plottar skall synas lite bättre. Vi väljer nu en startpunkt $p = (x_1, y_1)$, t.ex.

```
>> p=[1,2];
```

Av illustrativa skäl plottar vi nivåkurvan till f genom $(0, 1)$ och markerar punkten $(0, 1, f(0, 1)$ med en stjärna;

```
>> c=f(p(1),p(2));
>> contour(X,Y,Z,[c c]),axis equal
>> plot3(p(1),p(2),c,'r*')
>> plot3([p(1) p(1)],[p(2) p(2)],[0 c])
```

Sedan beräknar vi de partiella derivatorna till $f(x, y)$ i $(0, 1)$;

```
>> gx=fx(p(1),p(2));gy=fy(p(1),p(2));
```

Gradientmetodens ide är nu att stega fram i gradientens riktning till en ny punkt q, som förhoppningsvis ligger närmare en maximipunkt..

```
>> r=0.7;
>> q=p+r*[gx,gy]
```

Låt oss nu dra en linje mellan startpunkten p och den nya punkten q så att det tydligt framgår att vi stegat oss fram vinkelrät mot nivåkurvan (i den riktning för vilket funktionsvärdena växer snabbast);

```
>> d=f(q(1),q(2));
>> plot3(q(1),q(2),d,'r*')
>> plot3([p(1) q(1) q(1)],[p(2) q(2) q(2)],[0 0 d])
```

Analysera gärna närmare genom att förstora och rotera bilden.

Eftersom skillnaden i funktionsvärdena;

```
>> diff=d-c
```

är relativt stor så väljer vi att upprepa steg (3) och (4), och eftersom q är ny utgångspunkt för våra beräkningar sätter vi;

```
>> p=q;
```

Då är det bara att upprepa våra kommandon ovan;

```
>> c=f(p(1),p(2));
>> contour(X,Y,Z,[c c])
>> gx=fx(p(1),p(2));gy=fy(p(1),p(2));
>> q=p+r*[gx,gy];
>> d=f(q(1),q(2));
>> diff=d-c
>> plot3(q(1),q(2),d,'r*')
>> plot3([p(1) q(1) q(1)],[p(2) q(2) q(2)],[0 0 d])
>> p=q
```

Antag nu att vi vill upprepa iterationen tills skillnaden mellan funktionsvärdena i två efterföljande steg (`diff`) är tillräckligt liten (vilket indikerar att vi kommit nära ett maximum). Upprepningen underlättas om vi gör en snurra;

```
while diff>10^(-4)
    c=f(p(1),p(2));
    contour(X,Y,Z,[c c])
    gx=fx(p(1),p(2));gy=fy(p(1),p(2));
    q=p+r*[gx,gy]
    d=f(q(1),q(2));
    diff=d-c
    plot3(q(1),q(2),d,'r*')
    plot3([p(1) q(1) q(1)],[p(2) q(2) q(2)],[0 0 d])
    p=q;
    pause(0.5)
end
hold off
```

Vilken lokal maximipunkt närmar vi oss i detta fall?

Gradientmetoden kan naturligtvis även användas för att hitta max/min av funktioner av fler än två variabler. Låt oss titta på ett exempel där funktionen beror på tre variabler och där vi med plottar försöker illustrera hur metoden stegar sig fram till ett maximum.

Exempel 1: Betrakta funktionen $f(x,y,z) = \dfrac{1+x^2+y^2+z^2}{1+x^2+2y^2+z^4}$.

Låt oss börja med att illustrera funktionsvärdena utefter några plan;

```
>> f=@(x,y,z) (1+x.^2+y.^2+z.^2)./(1+x.^2+2*y.^2+z.^4);
>> [X,Y,Z]=meshgrid(linspace(-3,3,20));
>> W=f(X,Y,Z);
>> slice(X,Y,Z,W,[-3 0 3],[-3 0 3],[])
>> shading interp
>> alpha(0.35)
>> axis equal
>> hold on
```

Följande snurra illustrerar sedan hur vi med gradientmetoden kan närma oss ett maximum för funktionen f;

```
p=[1,1.5,2.5];
diff=1;
h=1e-5;
while diff>0.001
   x=p(1);y=p(2);z=p(3);
   grad(1)=(f(x+h,y,z)-f(x-h,y,z))/(2*h);
   grad(2)=(f(x,y+h,z)-f(x,y-h,z))/(2*h);
   grad(3)=(f(x,y,z+h)-f(x,y,z-h))/(2*h);
   q=p+grad;
   diff=f(q(1),q(2),q(3))-f(x,y,z);
   quiver3(x,y,z,q(1)-x,q(2)-y,q(3)-z,'LineWidth',2)
   p=q;
   pause(0.2)
end
hold off
```

Här startade vi iterationen i punkten $(1, 1.5, 2.5)$. Pröva gärna lite andra startpunkter och se vad som händer. □

2.4 Andra optimeringsmetoder

Om man vill bestämma maximi- och minimipunkter till en reellvärd funktion av en variabel kan man t.ex. använda kommandot `fminbnd`. T.ex. ger;

```
>> f=@(x) x.*sin(x);
>> xmin=fminbnd(f,1,3), ymin=f(xmin)
```

en lokal minimipunkt (`xmin`) på intervallet $[1, 3]$ till funktionen $f(x) = x \sin x$, samt funktionsvärdet (`ymin`) i denna minimipunkt.

Notera att `fminbnd` nödvändigtvis inte ger den punkt där funktionen antar sitt minsta värdet på intervallet, utan bara en lokal minimipunkt, som i exemplet ovan (plotta grafen till funktionen och verifiera att `fminbnd` inte gav det minsta värdet på intervallet).

Det finns dessvärre inget kommando som på liknande sätt hittar maximum av funktioner. Detta är dock inget större bekymmer ty maximipunkterna till $f(x)$ är desamma som minimipunkterna till $-f(x)$. Ett lokalt maxima hittar vi därför med följande kommandon;

```
>> g=@(x) -f(x);
>> xmax=fminbnd(g,1,3), ymax=f(xmax)
```

För att hitta lokala minimipunkter till funktioner av flera variabler kan vi istället använda `fminunc` (om funktionen är differentierbar) eller `fminsearch` (om ej differentierbar). Pröva t.ex.

```
>> f=@(x) x(1).*sin(x(2))+x(2)-cos(x(1));
>> xymin=fminunc(f,[7,-8]), zmin=f(xymin)
```

Notera hur vi här definierat den anonyma funktionen med `x(1)` resp. `x(2)` istället för `x` och `y`. Som input i `fminunc` måste man ange en startgissning (i detta fall punkten $(7, -8)$) nära den sökta minimipunkten. Utifrån detta värde stegar sig metoden successivt närmare minimipunkten, tills tillräcklig noggrannhet uppnås. Om man vill kan man få MATLAB att redovisa detaljer om de olika stegen som `fminunc` utför för att hitta minimipunkten (eller snarare en förbättrad approximation av minmipunkten). Ge i så fall följande kommando;

```
>> options=optimset('Display','iter');
>> xymin=fminunc(f,[7,-8],options)
```

Algoritmen innehåller villkor för när iterationen skall avbrytas, och om man vill kan man ändra på dessa villkor. Den som vill kan läsa mer om detta t.ex. genom att ge kommandot `help fminunc`.

Precis som för funktioner i en variabel finns inget färdigt kommando för att bestämma maximum av funktioner av flera variabler, utan man studerar istället minimum av funktionen $-f$. T.ex. ger;

```
>> g=@(x) -f(x);
>> xymax=fminunc(g,[5,-9]), zmax=f(xymax)
```

en lokal maximipunkt till funktionen $f(x,y) = x\sin y + y - \cos x$ i närheten av punkten $(5,-9)$. Försök övertyga dig om att ovanstående kommandon verkligen ger lokala maximi- och minimipunkter, genom att plotta funktionsytan och/eller nivåkurvor nära punkterna.

Ett annat alternativ om man söker max/min till en differentierbar funktion av flera variabler är att utnyttja det faktum att en max/min-punkt också är en stationär punkt dvs. en punkt där $\nabla f = \mathbf{0}$. För att bestämma lösningar till ett sådant system av ekvationer kan vi sedan använda teknikerna från avsnitt 2.1 & 2.2. En fördel med denna metod är att den (förutom max/min) också ger oss ev. sadelpunkter till funktionen. Låt oss titta på ett exempel.

Exempel 1: Betrakta åter igen funktionen $f(x,y) = x\sin y + y - \cos x$. De stationära punkterna till $f(x,y)$ är lösningar till följande ekvationssystem;

$$\begin{cases} \sin y + \sin x &= 0 \\ x\cos y + 1 &= 0 \end{cases}$$

Lösningar kan vi sedan bestämma med t.ex. Newtons metod (avsnitt 2.1) men låt oss för enkelhets skull använda det färdiga kommandot `fsolve` (se avsnitt 2.2);

```
>> fun=@(x) [sin(x(2))+sin(x(1)), x(1).*cos(x(2))+1];
>> stat=fsolve(fun,[8,-8])
```

Försök verifiera att den punkt som erhålls är en sadelpunkt till $f(x,y)$. Plotta t.ex. nivåkurvor i en omgivning av punkten och undersök Hessianen till funktionen f i den erhållna punkten. Pröva gärna också att byta startpunkt och se om du lyckas hitta samma max/min-punkter som med `fminunc` ovan. □

Ibland vill man bestämma max/min av en funktion $f(\mathbf{x})$ under ett eller flera bivillkor. Då kan man istället använda kommandot `fmincon`. Bivillkoren kan vara av lite olika typ t.ex. linjära bivillkor av typen $A\mathbf{x} \leq \mathbf{c}$ och/eller $B\mathbf{x} = \mathbf{d}$ där A, B är matriser och $\mathbf{c}, \mathbf{d}$ är vektorer. Man kan också som bivillkor ange enkla gränser på $\mathbf{x}$ av typen $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$, för några vektorer $\mathbf{l}, \mathbf{u}$. Det är också möjligt med icke-linjära bivillkor av typen $\mathbf{g}(\mathbf{x}) \leq 0$ eller $\mathbf{h}(\mathbf{x}) = 0$. Om sådana icke-linjära bivillkor finns med så skall syntax för `fmincon` vara på formen;

```
>> x=fmincon(f,x0,A,c,B,d,l,u,@nonlcon)
```

där `f` är målfunktionen (angiven som anonym funktion), `x0` är en startapproximation och `nonlcon` är en funktionsfil för de icke-linjära funktionerna **g** och **h** som skall ha formen;

```
function [g,h]=nonlcon(x)
g=g(x(1),x(2),...,x(n));
h=h(x(1),x(2),...,x(n));
```

Normalt har vi inte bivillkor av alla typer på samma gång i de problem vi vill lösa. Då ersätter vi motsvarande platser i anropet med `[]`.

Liksom `fminunc` behöver `fmincon` en startapproximation. Vi bör därför först bilda oss en uppfattning om det över huvudtaget finns några max/min-punkter och var vi i så fall kan förvänta oss att dessa finns. Ibland kan en plott vara till hjälp att undersöka detta. Låt oss titta på ett exempel.

Exempel 2: Antag att vi söker det största och minsta värdet på funktionen $f(x,y) = x\sin y + y(\cos x - 1)$ då (x,y) ligger på cirkeln $(x-4)^2 + (y-4)^2 = 10$.

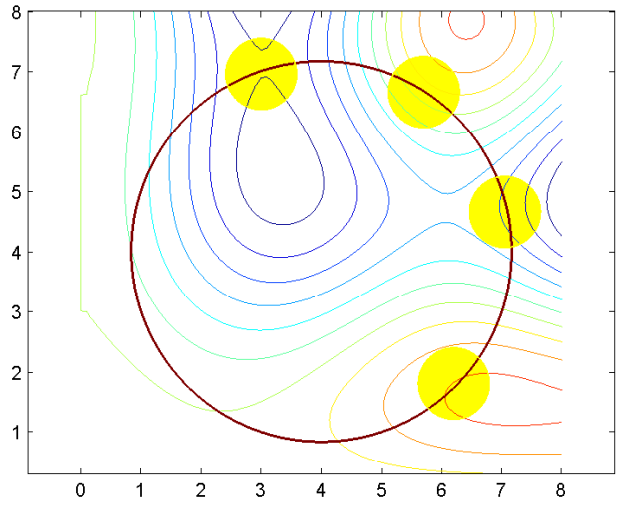
Låt oss börja med att plotta skärningskurvan mellan funktionsytan $z = f(x,y)$ och cylindern $(x-4)^2 + (y-4)^2 = 10$ (betraktad som en ekvation i x, y, z). Detta kan vi t.ex. åstadkomma med följande kommandon;

```
>> f=@(x,y) x.*sin(y)+y.*(cos(x)-1);
>> g=@(x,y) (x-4).^2+(y-4).^2;
>> [X,Y]=meshgrid(0:0.1:8); Z=g(X,Y);
>> q=contour(X,Y,Z,[10,10]);
>> q=q(:,[2:length(q)]); x=q(1,:); y=q(2,:);
>> z=f(x,y); plot3(x,y,z),grid
```

Av figuren framgår tydligt hur funktionsvärdena varierar och att det finns två maxima och två minima. Det kan dock vara lite svårt att avläsa var dessa ligger. En annan figur, som kanske bättre ger oss en uppfattning om var max/min-punkterna finns, får vi om vi plottar nivåkurvan $(x-4)^2 + (y-4)^2 = 10$ tillsammans med ett antal nivåkurvor till f;

```
>> contour(X,Y,f(X,Y)),grid,hold on
>> contour(X,Y,g(X,Y),[10,10]),hold off
```

Genom extrempunkterna vet vi att det går nivåkurvor till f som tangerar cirkeln. Dessa nivåkurvor finns (naturligtvis) inte med i figuren men vi kan uppskatta var de skulle kunna tangera.



T.ex. ser det ut som det ligger en minimipunkt nära $(7, 5)$. Låt oss använda `fmincon` för att bestämma denna extrempunkt närmare. Vi börjar då med att skapa följande funktionsfil;

```
function [g,h]=nonlcon(x)
g=[];
h=(x(1)-4)^2+(x(2)-4)^2-10;
```

Här satte vi `g=[]` eftersom vi inte har något bivillkor av typen $\mathbf{g}(\mathbf{x}) \leq 0$.

Sedan hittar vi extrempunkten med;

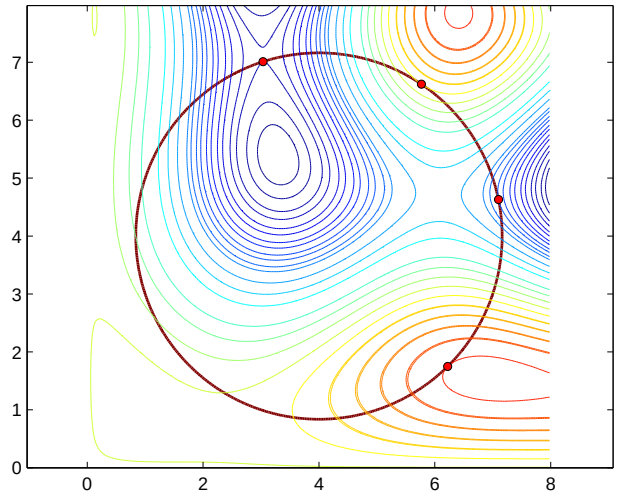
```
>> f=@(x) f(x(1),x(2));
>> x0=[7;5];
>> x=fmincon(f,x0,[],[],[],[],[],[],@nonlcon)
```

Med kommandot `ginput` kan man istället välja startpunkt genom att klicka på en punkt i figurfönstret. Det förenklar en del om man vill pröva sig fram och se vad som händer för olika startvärden. Ge i så fall följande kommandon (och upprepa dem t.ex. med en liten snurra);

```
>> x0=ginput(1)
>> x=fmincon(f,x0,[],[],[],[],[],[],@nonlcon)
```

Om man vill bestämma maximipunkterna så måste man som tidigare studera $-f$. Försök bestämma alla de fyra extrempunkterna med hjälp av `fmincon`. Gör

sedan en ny plott med nivåkurvor till f och cirkeln i samma figur (i stil med ovan), men se till att också få med nivåkurvorna genom extrempunkterna. Det kan se ut ungefär så här;



Notera speciellt hur nivåkurvorna till f genom extrempunkterna tangerar nivåkurvan $(x-4)^2+(y-4)^2 = 10$ (dvs. cirkeln). Eftersom gradienten till $f(x,y)$ och gradienten till $g(x,y) = (x-4)^2+(y-4)^2-10$ är vinkelräta mot resp. nivåkurvor måste de vara parallella i extrempunkterna dvs. $\nabla f = \lambda \nabla g$, för någon konstant λ. Denna observation ligger till grund för idén bakom *Lagrange multiplikatormetod* (se avsnitt 13.3 i *Calculus*, av Adams och Essex). Villkoret $\nabla f(x,y) = \lambda \nabla g(x,y)$, tillsammans med bivillkoret $g(x,y) = 0$, ger oss ett ekvationssystem med tre ekvationer i tre obekanta (x, y och λ). Systemen är i regel icke-linjära och kan vara knepiga (och oftast omöjliga) att lösa exakt för hand men kan lösas approximativt t.ex. med metoder från avsnitt 2.1 eller 2.3. I vårat exempel ovan erhålls följande ekvationssystem;

$$\begin{cases} \sin y - y\sin x - \lambda \cdot 2(x-4) &= 0 \\ x\cos y + \cos x - 1 - \lambda \cdot 2(y-4) &= 0 \\ (x-4)^2 + (y-4)^2 &= 10 \end{cases}$$

För att få en uppfattning om ev. lösningar till ekvationssystemet plottar vi de delar av ekvationssystemets nivåytor som ligger i området $0 \leq x \leq 8, 0 \leq y \leq 8, -4 \leq \lambda \leq 4$

```
>> e1=@(x,y,z) sin(y)-y.*sin(x)-z.*2.*(x-4);
>> e2=@(x,y,z) x.*cos(y)+cos(x)-1-z.*2.*(y-4);
>> e3=@(x,y,z) (x-4).^2+(y-4).^2;
>> [X,Y,Z]=meshgrid(0:0.3:8,0:0.3:8,-4:0.3:4);
```

```
>> clf
>> isosurface(X,Y,Z,e1(X,Y,Z),0)
>> isosurface(X,Y,Z,e2(X,Y,Z)+5,5)
>> isosurface(X,Y,Z,e3(X,Y,Z),10)
```

Vi ser att systemet verkar ha fyra lösningar nära punkterna $(6,2,0), (7,4,0), (6,6,1)$ resp. $(3,7,0)$. Detta stämmer även bra in på tidigare plottar. För att bestämma lösningarna lite närmare använder vi kommandot `fsolve`;

```
>> e=@(x) [e1(x(1),x(2),x(3)),
           e2(x(1),x(2),x(3)),
           e3(x(1),x(2),x(3))-10]
>> losn1=fsolve(e,[6,2,0])
>> losn2=fsolve(e,[7,5,-1])
>> losn3=fsolve(e,[6,7,1])
>> losn4=fsolve(e,[3,7,0])
```

Ovan har vi sett två olika sätt med vilket vi kan bestämma maximum och minimum av funktionen $f(x,y) = x\sin y + y(\cos x - 1)$ under bivillkoret $(x-4)^2 + (y-4)^2 = 10$. Metoderna är ganska generella och kan användas för andra liknande problem. I vissa fall (som ovan, och i de flesta andra liknande problem ni kommer stöta på i denna kurs), när bivillkoret beskriver en mängd som kan parametriseras, kan problemet dock förenklas. Om vi parametriserar cirkeln enl;

$$\begin{cases} x = 4 + \sqrt{10}\cos t \\ y = 4 + \sqrt{10}\sin t \end{cases}, \quad 0 \le t < 2\pi$$

och sätter;

$$g(t) = f(4 + \sqrt{10}\cos t, 4 + \sqrt{10}\sin t)$$

så reduceras problemet till att bestämma maximum och minimum av $g(t)$, för $0 \le t < 2\pi$. Detta är ett extremvärdesproblem i en variabel och utan några bivillkor, ett betydligt enklare problem än det ursprungliga. Ett lokalt minimum finner vi t.ex. med följande kommandon;

```
>> f=@(x,y) x.*sin(y)+y.*(cos(x)-1);
>> x=@(t) 4+sqrt(10)*cos(t); y=@(t) 4+sqrt(10)*sin(t);
>> fun=@(t) f(x(t),y(t));
>> tmin=fminbnd(fun,0,2*pi)
>> xmin=x(tmin), ymin=y(tmin), zmin=fun(tmin)
```

Bestäm även de övriga tre extrempunkterna och jämför med vad tidigare metoder gav. □

3 Multipelintegraler

3.1 Dubbelintegraler

I detta kapitel skall vi studera olika sätt på vilket man kan använda MATLAB för att beräkna multipelintegraler, och i detta första avsnitt skall vi speciellt titta på dubbelintegraler. Låt oss börja med ett exempel.

Exempel 1: Antag att vi vill beräkna dubbelintegralen;

$$\iint_D (y \sin x - x \cos y + 10)\, dxdy$$

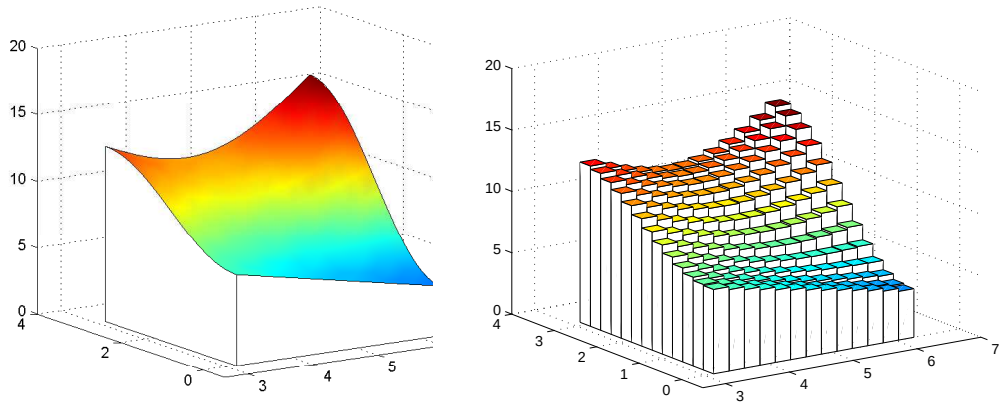
där D är rektangeln $D : \pi \leq x \leq 2\pi\ ,\ 0 \leq y \leq \pi$.

Just denna integral är inte svår att analytiskt beräkna exakt (gör det!) men i allmänhet är det svårt eller omöjligt att beräkna integraler, och vi får nöja oss med närmevärden. Låt oss därför studera några numeriska metoder med vilket integralen kan beräknas approximativt. Ett enkelt sätt att göra detta är genom Riemannsummor;

$$\sum_{i=1}^{n}\sum_{j=1}^{m} f(x_{ij}, y_{ij})\Delta x_i \Delta y_j$$

där x_i och y_j är indelningspunkterna i x- resp. y-led och där $\Delta x_i = x_{i+1} - x_i$, $\Delta y_j = y_{j+1} - y_j$, samt där (x_{ij}, y_{ij}) är en godtycklig punkt i delrektangeln R_{ij} : $x_i \leq x \leq x_{i+1},\ y_j \leq y \leq y_{j+1}$.

Eftersom integranden (i detta fall) är positiv på integrationsområdet kan dubbelintegralens värde tolkas som volymen av en kropp (se figuren till vänster nedan). Varje term i Riemannsumman kan också tolkas som volym, av ett rätblock med basen R_{ij} och höjden $f(x_{ij}, y_{ij})$. Summan av dessa rätblocks-volymer ger en god approximation av dubbelintegralens värde och i gräns, då "indelningens finhet" går mot 0, kommer Riemannsummorna att närma sig integralens exakta värde. Om vi i vårat exempel delar in integrationsområdet i t.ex. 13×13 delrektanglar och väljer $x_{ij} = x_i$ och $y_{ij} = y_j$ (motsvarar det nedre vänstra hörnet i varje delrektangel) så kan Riemannsumman illustreras med figuren till höger



neda

Om vi vill få en god approximation av dubbelintegralens värde bör vi dock dela in D i många fler delområden. Låt oss därför istället göra en partition av D med 1000×1000 delrektanglar, alla med arean $\pi^2/10^6$;

```
>> n=1000;
>> x=linspace(pi,2*pi,n+1); y=linspace(0,pi,n+1);
>> [X,Y]=meshgrid(x,y);
```

Om vi som ovan väljer $x_{ij} = x_i$ och $y_{ij} = y_j$ så kan motsvarande Riemannsumma beräknas med;

```
>> f=@ (x,y) y.*sin(x)-x.*cos(y)+10;
>> dA=pi^2/n^2;
>> I=sum(sum(f(X(1:n,1:n),Y(1:n,1:n))))*dA
```

Om vi jämför med integralens exakta värde $9\pi^2$ så är det relativa felet av storleksordningen 10^{-4}, vilket får anses ganska stort med tanke på det mycket stora antalet delrektanglar. För att minska relativa felet med en faktor 10, till storleksordningen 10^{-5}, får vi räkna med att öka antalet delrektanglar (och därmed antalet element i matriserna X och Y) med en faktor 100, till 10^8. Det tar lång tid för MATLAB att hantera så stora matriser som med denna metod behövs för att uppnå god noggrannhet, och det är framför allt inte speciellt effektivt. För att ha något att jämföra med när vi lite längre fram skall titta på andra metoder kan vi låta MATLAB mäta den tid kalkylerna tar. Pröva t.ex.

```
>> tic, I=sum(sum(f(X(1:n,1:n),Y(1:n,1:n))))*dA, toc
```

Skall man beräkna Riemannsummor med finare indelning krävs det att kalkylerna organiseras på annat sätt. □

För att beräkna approximativa värden på enkelintegraler (dvs. integraler av funktioner av en variabel) används ofta *Simpsons formel*. Den bygger på att man delar in integrationsområdet i små intervall, säg m stycken, och sedan approximerar integranden i vart och ett av intervallen med ett andragradspolynom. Detta polynom har samma värde som integranden i delintervallets ändpunkter och mittpunkt. Summan av integralerna av andragradspolynomen ger Simpsons formel;

$$\int_a^b f(x)dx \approx \frac{(b-a)}{m} \sum_{k=0}^{2m+1} w_k \cdot f(x_k).$$

Här är $x_k = a + \frac{k}{2m+1}(b-a)$ och w_k är i tur och ordning $\frac{1}{6},\ \frac{4}{6},\ \frac{2}{6},\ \frac{4}{6},\ \frac{2}{6}, \cdots \frac{4}{6},\ \frac{2}{6},\ \frac{4}{6}, \frac{1}{6}$.

Genom upprepad integration, där Simpsons formel används först vid integralberäkning i ena variabeln och sedan i den andra, kan man härleda en tvådimensionell Simpsons formel;

$$\iint_D f(x,y)\,dx\,dy \approx \frac{(b-a)(d-c)}{m^2} \sum_{i=0}^{2m+1} \sum_{j=0}^{2m+1} w_{ij} \cdot f(x_i, y_j).$$

Här är $x_i = a + \frac{i}{2m+1}(b-a)$, $y_j = c + \frac{j}{2m+1}(d-c)$ och $w_{ij} = w_i \cdot w_j$.

forts. Exempel. Låt oss istället använda Simpsons formel för att beräkna dubbelintegralen;

$$\iint_D (y \sin x - x \cos y + 10)\,dxdy$$

där D är rektangeln $D : \pi \le x \le 2\pi\ ,\ 0 \le y \le \pi$.

Med 500 delintervall, och därmed 1001 punkter (inklusive mittpunkterna i delintervallen), får vi följden w_k i MATLAB med;

```
>> m=500; n=2*m+1; w=[1 [3*ones(1,n-2)+(-1).^(2:n-1)] 1]/6;
```

Vikterna w_{ij} samlar vi sedan i matrisen;

```
>> W = w' * w;
```

För att beräkna dubbelintegralen med Simpsons formel, och samtidigt ta beräkningstiden, ger vi sedan följande kommandon;

```
>> x=linspace(pi,2*pi,n); y=linspace(0,pi,n);
>> [X,Y]=meshgrid(x,y);
>> dA=pi^2/m^2;
>> tic, I=sum(sum(W.*f(X,Y)))*dA, toc
```

Tidsåtgången blir ungefär samma som tidigare, men relativa felet blir nu av storleksordningen $1/m^5$. □

MATLAB:s integralberäkningsprogram `quad` (för att beräkna enkelintegraler) använder Simpsons formel men med en adaptiv metod där intervallindelningen styrs av beräkningsresultaten. Där funktionen varierar mycket görs finare indelning, och där variationen är liten görs grov indelning. Det finns också ett par andra beräkningsmetoder att tillgå; `quadl` och `quadgk`. Du kan läsa mer om detta i MATLAB:s *Product Help* under rubriken *Numerical Integration (Quadrature*)

Dubbelintegraler över axelparallella rektanglar kan i MATLAB beräknas med hjälp av kommandot `dblquad`. I princip används upprepad integration genom att MATLAB först gör en indelning av den andra variabelns intervall, beräknar enkelintegralen i alla dessa delningspunkter med avseende på den första variabeln med hjälp av `quad` (om man vill kan man i `dblquad` välja annan metod för beräkning av enkelintegralerna). Simpsons formel tillämpas sedan med dessa beräknade integralvärden. Indelningen av andra variabelns intervall förfinas, integraler beräknas i de nya punkterna, Simpsons formel tillämpas igen. Proceduren upprepas till förändringen är liten nog. Även i detta fall görs indelningen finare där det behövs. Trippelintegralsberäkning, som vi kommer till i avsnitt 3.2 bygger på samma idé.

forts. Exempel. Betrakta åter igen dubbelintegralen;

$$\iint_D (y \sin x - x \cos y + 10)\, dxdy$$

där D är rektangeln $D : \pi \leq x \leq 2\pi\ ,\ 0 \leq y \leq \pi$.

Eftersom vi redan definierat integranden som en anonym funktion kan vi beräkna dubbelintegralen, med relativt fel 10^{-6} vilket är grundinställningen, genom kommandot;

```
>> I = dblquad(f,pi,2*pi,0,pi)
```

Om vi jämför med integralens exakta värde $9\pi^2$ så ser vi att det relativa felet i själva verket är ungefär 10^{-10}. För att minska beräkningstiderna kan vi ibland acceptera större relativt fel. Kommandot;

```
>> I = dblquad(f,pi,2*pi,0,pi,0.006)
```

ger ett större relativa fel än ovan men ändå mycket bättre än vad den första Riemannsumman gav, och dessutom mer än tio gånger så snabbt (kontrollera!). □

Vid beräkning av dubbelintegraler över ett allmännare område D, som ej nödvändigtvis är rektangulärt men som ligger inuti en axelparallell rektangel R, använder vi att;

$$\iint_D f(x,y)\, dxdy = \iint_R f(x,y)\chi_D(x,y)\, dxdy$$

där χ_D är karakteristiska funktionen för området D dvs.

$$\chi(x,y) = \begin{cases} 1 & , \text{ då } (x,y) \in D \\ 0 & , \text{ då } (x,y) \notin D \end{cases}$$

Exempel 2: Antag att vi vill beräkna dubbelintegralen;

$$\iint_D (y \sin x - x \cos y + 10)\, dxdy$$

där $D = \{(x, y) \colon 0 \leq x \leq 2\ ,\ 0 \leq y \leq x^2\}$.

Vi definierar då anonyma funktioner som motsvarar den karakteristiska funktionen, integranden, och för enkelhets skull, produkten av dessa två;

```
>> karD = @ (x,y) (0 <= x).* (x <= 2).*(0 <= y).* (y <= x.^2);
>> f = @(x,y) y.*sin(x)-x.*cos(y)+10;
>> fD = @(x,y) karD(x,y).*f(x,y);
```

Eftersom $D \subset \{(x, y) \colon 0 \leq x \leq 2\ ,\ 0 \leq y \leq 4\}$, så beräknar vi sedan integralen med kommandot;

```
>> I = dblquad(fD,0,2,0,4)
```

Varje annan rektangel som omfattar D duger (pröva det!). □

3.2 Trippelintegraler

Det är inte någon väsentlig skillnad på beräkning av trippelintegral och dubbelintegral med MATLAB. Motsvarigheten till `dblquad` för trippelintegraler heter `triplequad`.

Exempel 1: Låt oss se hur man använder `triplequad` för att beräkna integralen;

$$\iiint_D xy^2z^3\, dxdydz$$

där $D : 0 < x < 1\ ,\ 0 < y < 2\ , 1 < z < 2$.

Vi börjar som tidigare med att definiera integranden som en anonym funktion;

```
>> f=@(x,y,z) x.*y.^2.*z.^3;
```

och för att beräkna integralen ger vi därefter kommandot;

```
>> I=triplequad(f,0,1,0,2,1,2)
```

□

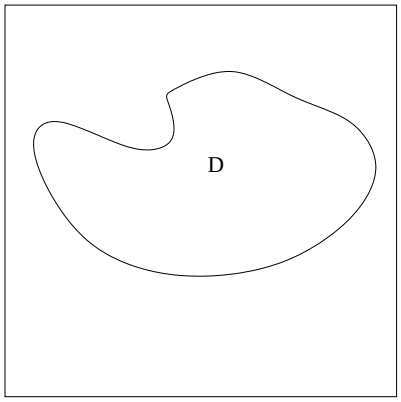
Om integrationsområdet inte är ett rätblock definierar vi en karakteristisk funktion för området, multiplicerar med den och integrerar, precis som i tvådimensionella fallet.

3.3 Monte Carlo metoden

I detta avsnitt skall vi se hur man kan beräkna flerdimensionella integraler approxmativt med den så kallade *Monte Carlo-metoden*. Vi kommer inte att göra någon matematisk analys av hur effektiv och noggrann metoden är, eftersom en sådan analys förutsätter kunskaper i matematisk statistik.

Som nämnts tidigare så använder programmen `quad`, `dblquad` och `triplequad` Simpsons formel på ett adaptivt sätt, där intervallindelningen styrs av beräkningsresultaten. Även om metoden och programmen fungerar ganska bra på de flesta integraler så finns det nackdelar. För det första krävs trots allt ganska många beräkningssteg för att ge önskad noggrannhet. Dessutom ökar beräkningstiden väsentligt med ökande dimension. Vidare har programmen problem med diskontinuiteter, som naturligt kommer in om integrationsområdet inte är en rektangel eller ett rätblock. Det finns heller inget färdigt MATLAB-program för n-dimensionella integraler med $n > 3$.

En enkel metod som kan vara en lösning på dessa problem är *Monte Carlo-metoden*. Låt oss illustrera idén bakom denna metod genom se hur vi kan beräkna arean A av ett område D som ligger inuti en kvadrat med arean 1 (se figur).



Om man väljer en punkt (x, y) slumpvis inuti kvadraten så är sannolikheten att punkten ligger i området D precis A (man måste ställa en del krav på hur området ser ut, för att detta påstående skall vara sant). Om man istället väljer N sådana punkter; (x_i, y_i), $i = 1, 2, ..., N$, så bör alltså i genomsnitt $A \cdot N$ av dessa hamna inuti D. Om M stycken av punkterna ”träffar” D så är alltså $M \approx A \cdot N$, och därmed $A \approx M/N$.

Mer allmänt, om området D istället ligger inom en axelparallell rektangel R med area W så tar man istället punkterna slumpvis i rektangeln R och får att $A \approx W \cdot M/N$.

Med denna metod skall vi alltså;

1. Välja N punkter (x_i, y_i) slumpvis i rektangeln R.
2. Undersöka hur många av de N punkterna som ligger i område D. Kalla detta antal för M.
3. Ta $A \approx W \cdot M/N$ som en approximation av områdets area.

Man kan visa att om området uppfyller vissa villkor (som uppfylls av de flesta områden man kan tänkas komma på) så är beräkningsfelet med denna metod av storleksordning $1/\sqrt{N}$. I det tvådimensionella fallet är detta jämförbart med Riemannsumman men i högre dimension är det väsentligt bättre.

Observera att om χ_D är områdets karakteristiska funktion, och (x_i, y_i) är de slumpvis utvalda punkterna, så ges antalet träffar i D av;

$$M = \sum_{i=1}^{N} \chi_D(x_i, y_i).$$

varpå vi får att;

$$A \;\approx\; \frac{W}{N} \sum_{i=1}^{N} \chi_D(x_i, y_i)\,.$$

Exempel 1: Antag att vi vill beräkna arean av det område D i xy-planet som ges av $1 \leq xy \leq 4,\ x \leq y \leq 4x,\ x > 0,\ y > 0$.

Vi börjar med att skapa den karakteristiska funktionen för området;

```
>> karD=@(x,y) (x.*y>=1).*(x.*y<=4).*(y>=x).*(y<=4*x);
```

De två hyperblerna skär linjerna i punkterna $(\frac{1}{2}, 2), (1, 1), (1, 4), (2, 2)$ och det är inte svårt att se att området D ligger inuti rektangeln $R : \frac{1}{2} \leq x \leq 2,\ 1 \leq y \leq 4$.

Vi behöver sedan generera slumpvisa punkter i rektangeln R. Detta åstadkoms genom att först generera slumpvisa punkter i kvadraten med sida 1, multiplicera med intervallens längder och sedan translatera så att vänster ändpunkt blir rätt;

```
>> N=1000; X=rand(N,2);
>> x=1/2+3/2*X(:,1); y=1+3*X(:,2);
```

Låt oss undersöka vilka punkter detta genererar genom att plotta alla punkter som hamnat inuti D med blå prick och övriga med röd prick;

```
>> in=find(karD(x,y)==1); plot(x(in),y(in),'b.'), hold on
>> out=find(karD(x,y)==0); plot(x(out),y(out),'r.'), hold off
```

Eftersom rektangelarean är $\frac{3}{2} \cdot 3 = \frac{9}{2}$ får vi arean med;

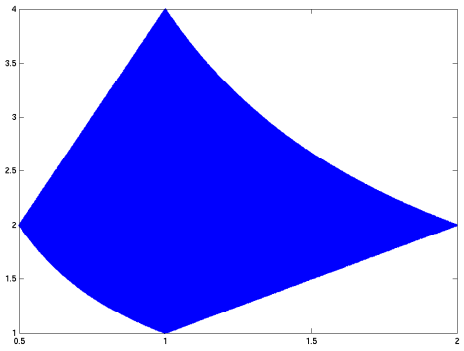
```
>> A=9/2*sum(karD(x,y))/N
```

Nu när vi ser att det hela fungerar kan vi öka antalet punkter t.ex. till en miljon;

```
>> N=10^6; X=rand(N,2);
>> x=1/2+3/2*X(:,1); y=1+3*X(:,2);
>> A=9/2*sum(karD(x,y))/N
```

Detta ger att arean av D är ≈ 2.08, vilket är att jämföra med det exakta värdet som är $3\ln(2)$ (≈ 2.076). En plott med alla punkter som hamnat inuti D ger en fin bild av området D;

```
>> in=find(karD(x,y)==1); plot(x(in),y(in),'.')
```



□

En väsentlig fördel med metoden är att varken beräkningarna eller felet påverkas av dimensionen. Om man har en n-dimensionell kropp som ligger i ett n-dimensionellt rätblock av volym W och slumpmässigt tar N punkter i detta rätblock, så är kroppens (n-dimensionella) volym $V \approx W \cdot \frac{M}{N}$, där M är antalet punkter som ligger i kroppen.

Om man istället vill beräkna en multipelintegral,

$$I \;=\; \int \cdots \int_{\Omega} f(\mathbf{x})\, d\mathbf{x}\,,$$

så gör man precis på samma sätt, fast nu blir

$$I \;\approx\; \frac{W}{N} \sum_{i=1}^{N} \chi_{\Omega}(\mathbf{x}_i) f(\mathbf{x}_i)\,.$$

Det finns förbättringar av metoden som vi inte går närmare in på här men vill du veta mer kan du t.ex. besöka web-sidan;

http://en.wikipedia.org/wiki/Monte_Carlo_method

Exempel 2: Antag att vi vill beräkna trippelintegralen;

$$\iiint_D 1/\sqrt{x^2+y^2+z^2}\; dxdydz$$

där $D = \{(x,y,z) : x^2+y^2+z^2 \leq 9,\ x>0,\ y>1,\ z>2\}$.

Notera att området D ligger inuti rätblocket $R:\ 0 \leq x \leq 3,\ 1 \leq y \leq 3, 2 \leq z \leq 3$ med volymen $W = 6$.

Som vanligt definierar vi områdets karakteristiska funktion och integranden som anonyma funktioner;

```
>> karD=@(x,y,z) (x.^2+y.^2+z.^2 < 9)
>> f=@(x,y,z) 1./sqrt(x.^2+y.^2+z.^2)
```

och produkten:

```
>> fD=@(x,y,z) karD(x,y,z).*f(x,y,z)
```

Vi låter sedan MATLAB bestämma en miljon punkter i rätblocket R slumpvis med kommandosekvensen;

```
>> N=10^6; X=rand(N,3); x=3*X(:,1); y=1+2*X(:,2); z=2+X(:,3);
```

Detta ger då integral-approximationen;

```
>> W=6;
>> I=sum(fD(x,y,z))*W/N
```

□

4 Vektoranalys

4.1 Vektorfält

Vi kan illustrera vektorfält, såväl i planet som i rummet, med kommandona `quiver` resp. `quiver3`. Först måste vi dock skapa ett rutnät med de punkter i vilket vi vill sätta pilar (som illustrerar vektorfältets storlek och riktning i respektive punkt). Det plana vektorfältet $\mathbf{F} = \sin(xy)\,\mathbf{i} + \cos(x-y)\mathbf{j}$ kan vi t.ex. illustrera med följande kommandon;

```
>> F1=@(x,y) sin(x.*y); F2=@(x,y) cos(x-y);
>> x=linspace(-3,3,20);y=linspace(-3,3,21);
>> [X,Y]=meshgrid(x,y);
>> quiver(X,Y,F1(X,Y),F2(X,Y))
```

Det är viktigt att tänka på att man inte sätter pilarna för tätt (dvs. har för många element i vektorerna `x` och `y`). Å andra sidan vill man ha tillräckligt med pilar för att få en bra uppfattning om vektorfältets skiftningar. Det kan vara svårt att få det bra från början och man kan behöva pröva sig fram.

Låt oss även plotta ett vektorfält i rummet;

```
>> F1=@(x,y,z) x.^3.*y+z.^2;
>> F2=@(x,y,z) x.*y.^2+z.^2;
>> F3=@(x,y,z) x.^2+y.^2.*z;
>> x=linspace(-3,3,11);y=linspace(-3,3,10);z=linspace(-3,3,10);
>> [X,Y,Z]=meshgrid(x,y,z);
>> quiver3(X,Y,Z,F1(X,Y,Z),F2(X,Y,Z),F3(X,Y,Z))
```

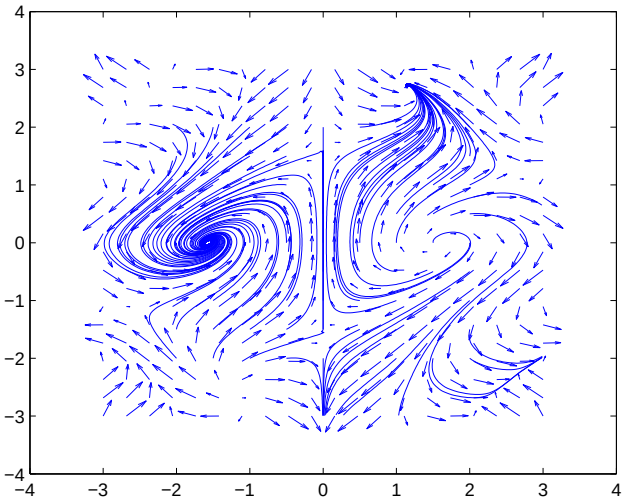
Om man här väljer för många punkter finns också risken att det tar för lång tid för MATLAB att plotta figuren (och att det hänger sig), så var försiktig när du skapar rutnätet med `meshgrid`. Börja hellre med ett lite grovare rutnät för att sedan förfina i den mån det behövs.

4.2 Fältlinjer och flöden

Vi kan även använda MATLAB för att plotta fältlinjer till vektorfält t.ex. med hjälp av kommandot `streamlines` (fältlinjerna kallas ju strömlinjer om vektorfältet representerar hastighetsvektorer i ett flöde). Man måste då också ange i vilka startpunkter som fältlinjerna skall börja i. Startpunkterna kan tex. skapas

med `meshgrid`. Följande kommandon plottar några fältlinjer till vektorfältet $\mathbf{F} = \sin(xy)\,\mathbf{i} + \cos(x-y)\mathbf{j}$ i samma figur som vektorfältet självt;

```
>> x=linspace(-3,3,20);y=linspace(-3,3,21);
>> [X,Y]=meshgrid(x,y);
>> F1=sin(X.*Y); F2=cos(X-Y);
>> quiver(X,Y,F1,F2)
>> [Sx,Sy]=meshgrid(-2:0.5:2);
>> streamline(X,Y,F1,F2,Sx,Sy)
```



Startpunkterna anges här av matriserna `Sx` och `Sy`. Vi kan naturligtvis plotta fältlinjerna genom samtliga punkter i vilket vi ritat en pil, men då riskerar vi att figuren blir för plottrig. Pröva och avgör själv;

```
>> streamline(X,Y,F1,F2,X,Y)
```

Det går även bra att plotta fältlinjer till vektorfält i rummet;

```
>> x=linspace(-3,3,11); y=linspace(-3,3,10); z=linspace(-3,3,10);
>> [X,Y,Z]=meshgrid(x,y,z);
>> F1=X.^3.*Y+Z.^2; F2=X.*Y.^2+Z.^2; F3=X.^2+Y.^2.*Z;
>> quiver3(X,Y,Z,F1,F2,F3)
>> [Sx,Sy,Sz]=meshgrid(-2:1:2);
>> streamline(X,Y,Z,F1,F2,F3,Sx,Sy,Sz)
```

Om man vill kan man också välja startpunkterna för fältlinjerna så att de ligger på en och samma (parametriserad) yta. Pröva t.ex.

```
>> [S,T]=meshgrid(0:0.1:1);
```

```
>> Sx=S.^2-T; Sy=S+T.^2; Sz=S-T;
>> surf(Sx,Sy,Sz)
>> streamline(X,Y,Z,F1,F2,F3,Sx,Sy,Sz)
```

Startpunkterna kan också ligga på en nivåyta. Pröva t.ex.

```
>> [S,T,R]=meshgrid(-3:0.3:3);
>> G=S.^2+T.^2+R.^2;
>> clf, isosurface(S,T,R,G,1);
>> [p,q]=isosurface(S,T,R,G,1);
>> Sx=q(:,1); Sy=q(:,2); Sz=q(:,3);
>> streamline(X,Y,Z,F1,F2,F3,Sx,Sy,Sz)
```

Man kan också studera hur partiklar rör sig (utefter fältlinjerna) under inverkan av vektorfältet;

```
>> verts = stream3(X,Y,Z,F1,F2,F3,Sx,Sy,Sz);
>> iverts = interpstreamspeed(X,Y,Z,F1,F2,F3,verts,0.01);
>> streamparticles(iverts,30,'animate',10,'FrameRate',50)
```

Om man vill illustrera någon typ av flöde/strömning genom en yta så kan det räcka med att plotta hastighetsvektorer på själva ytan (och inte i det omgivande området). Låt oss titta på ett exempel.

Exempel 1: Låt S vara den yta som parametriseras av $x = s+t, y = s^2-t^2, z = st$,
$0 \leq s,t \leq 1$ och låt $\mathbf{F} = (x-1)\mathbf{i} + 2z\mathbf{j} - y\mathbf{k}$ vara hastighetsvektorn i en vätskeströmning. Den vätskevolym som strömmar genom ytan S ges av normalytintegralen $\iint_S \mathbf{F}\bullet\hat{\mathbf{N}}\, dS$. För att illustrera hur strömningen varierar utefter ytan plottar vi ett antal hastighetsvektorer på ytan S.

```
>> [S,T]=meshgrid(0:0.05:1);
>> X=S+T; Y=S.^2-T.^2; Z=S.*T;
>> surf(X,Y,Z), hold on
>> F1=X-1;F2=2*Z;F3=-Y;
>> quiver3(X,Y,Z,F1,F2,F3), hold off
>> shading interp, axis equal
```

Om man är mer intresserad av hur mycket vätska som strömmar genom ytans olika delar och inte strömningens riktning så kan man plotta hastighetsvektorernas projektion i normalriktningen. Detta är lite mer komplicerat men kan t.ex. utföras med följande kommandorader.

```
>> [S,T]=meshgrid(0:0.05:1);
>> X=S+T; Y=S.^2-T.^2; Z=S.*T;
>> surf(X,Y,Z), hold on
>> [U,V,W]=surfnorm(X,Y,Z);
>> k=sqrt(U.^2+V.^2+W.^2); U=U./k; V=V./k; W=W./k;
>> [m,n]=size(U); r=m*n;
>> uq=U(1:r); vq=V(1:r); wq=W(1:r);
>> xq=X(1:r); yq=Y(1:r); zq=Z(1:r);
>> pq=dot([uq;vq;wq],[xq-1;2*zq;-yq]);
>> P=ones(size(U)); P(1:r)=pq;
>> quiver3(X,Y,Z,P.*U,P.*V,P.*W), hold off
>> shading interp, axis equal
```

Istället för att plotta normalvektorer till ytan som med sin längd anger strömningens storlek genom ytan så kan vi låta färgen på ytan variera med längden på vektorerna (om man inte anger något annat så bestäms färgen på ytan av höjden över xy-planet dvs. värdet på z i resp. punkt). Det kan då vara lättare att se åt vilket håll strömningen sker och hur stor den är på ytans olika delar. Istället för att använda `quiver3` (dvs. den näst sista kommandoraden ovan) så ger vi i så fall följande kommando;

```
>> surf(X,Y,Z,P), colorbar, shading interp
```

Det röda anger strömning genom ytan åt det ena hållet och blått åt det andra hållet. □

4.3 Divergens och rotation av vektorfält

Två viktiga begrepp inom vektoranalysen är *divergens* och *rotation*. För ett vektorfält $\mathbf{F}(x,y,z) = F_1(x,y,z)\mathbf{i} + F_2(x,y,z)\mathbf{j} + F_3(x,y,z)\mathbf{k}$ ger divergensen;

$$\text{div}\mathbf{F} = \frac{\partial F_1}{\partial x} + \frac{\partial F_2}{\partial y} + \frac{\partial F_3}{\partial z}$$

ett mått på vektorfältets tendens att ”stråla” ut från (eller in mot) en viss punkt. Antag till exempel att vektorfältet beskriver hastigheten hos molekylerna i en gas som hettas upp eller kyls ner. I områden där gasen expanderar har divergensen ett positivt värde (sådana områden kallas källor) och i områden där gasen komprimeras är divergensen negativ (sådana områden kallas sänkor).

Divergensen kan i MATLAB beräknas med kommandot `divergence` och för att illustrera storleken divergensen hos ett plant vektorfält kan vi t.ex. använda `contourf`;

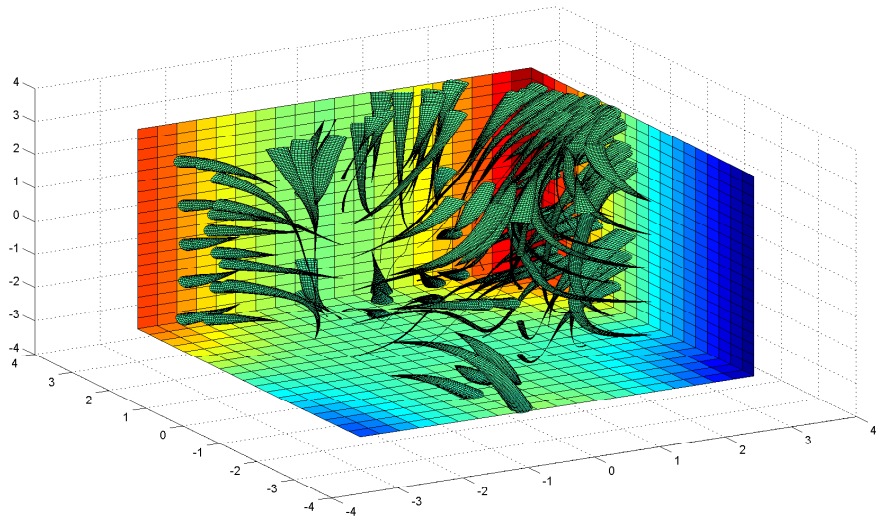
```
>> x=linspace(-3,3,30);y=linspace(-3,3,31);
>> [X,Y]=meshgrid(x,y);
>> F1=sin(X.*Y); F2=cos(X-Y);
>> DIV=divergence(X,Y,F1,F2);
>> contourf(X,Y,DIV), hold on
>> quiver(X,Y,F1,F2), hold off
```

och för vektorfält i rummet kan vi t.ex. använda `slice`;

```
>> x=linspace(-3,3,21); y=linspace(-3,3,20); z=linspace(-3,3,20);
>> [X,Y,Z]=meshgrid(x,y,z);
>> F1=X.^3.*Y+Z.^2; F2=X.*Y.^2+Z.^2; F3=X.^2+Y.^2.*Z;
>> DIV=divergence(X,Y,Z,F1,F2,F3);
>> slice(X,Y,Z,DIV,[3],[3],[-3])
```

Man kan också använda kommandot `streamtube` som plottar fältlinjer i form av tuber. Tjockleken på tuberna varierar proportionellt mot storleken på divergensen utefter resp. fältlinje;

```
>> [Sx,Sy,Sz]=meshgrid(-2:1:2);
>> streamtube(X,Y,Z,F1,F2,F3,Sx,Sy,Sz)
```



En annan typ av information om vektorfältet ger rotationen;

$$\text{curl}\mathbf{F} = \left(\frac{\partial F_3}{\partial y} - \frac{\partial F_2}{\partial z}\right)\mathbf{i} + \left(\frac{\partial F_1}{\partial z} - \frac{\partial F_3}{\partial x}\right)\mathbf{j} + \left(\frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y}\right)\mathbf{i}$$

Till skillnad mot divergensen (som är en skalär) så ger rotationen en vektor i varje punkt. Längden på vektorn ger ett mått på vektorfältets tendens att virvla i resp. punkt och riktningen på vektorn beskriver den axel kring vilket det virvlar mest. För att illustrera storleken på divergensen hos ett plant vektorfält kan vi som ovan t.ex. använda `contourf`;

```
>> x=linspace(-3,3,30);y=linspace(-3,3,31);
>> [X,Y]=meshgrid(x,y);
>> F1=sin(X.*Y); F2=cos(X-Y);
>> Rz=curl(X,Y,F1,F2);
>> contourf(X,Y,abs(Rz)), hold on
>> quiver(X,Y,F1,F2), hold off
```

och för vektorfält i rummet kan vi t.ex. använda `slice`;

```
>> x=linspace(-3,3,21); y=linspace(-3,3,20); z=linspace(-3,3,20);
>> [X,Y,Z]=meshgrid(x,y,z);
>> F1=X.^3.*Y+Z.^2; F2=X.*Y.^2+Z.^2; F3=X.^2+Y.^2.*Z;
>> [Rx,Ry,Rz]=curl(X,Y,Z,F1,F2,F3);
>> slice(X,Y,Z,sqrt(Rx.^2+Ry.^2.^2+Rz.^2),[3],[3],[-3])
```

Man kan också använda kommandot `streamribbon` som plottar fältlinjer i form av band. Där bandet vrider sig mycket runt fältlinjen är rotationen stor;

```
>> [Sx,Sy,Sz]=meshgrid(-1:1:1);
>> streamribbon(X,Y,Z,F1,F2,F3,Sx,Sy,Sz)
```

Det kan ta lite tid för MATLAB att utföra beräkningarna och plotta resultatet så ha lite tålamod. Tänk på att vara försiktig med storleken på matriserna, börja hellre med en gles indelning och förfina gradvis.