KTH Engineering Sciences

Laboration 1

Ekvationslösning

Före redovisningen ska ni skicka in MATLAB*-filer och vissa resultatfiler i Canvas. Vad som ska skickas in står angivet efter respektive uppgift. En sammanställning ges här:*

Uppgift 1: `fixpunkt.m`, `newton.m`, `konvplot1.png`, `konvplot2.png`

Uppgift 2: `punkter.m`, `langd.m`

Uppgift 3: `tidsplot.png`, `kanslighetLU.m`, `tidtabell`

På redovisningen ska ni (båda individuellt om ni är två i gruppen) kunna redogöra för teori och algoritmer som ni använt. Ni ska kunna svara på frågepunkterna (•) och förklara hur era MATLAB*-program fungerar. Kom väl förberedda!*

OBS! Era program ska inte använda MATLAB*s symboliska funktioner/variabler (*`syms`*) för beräkningar. Använd inte heller* MATLAB *Live Script (*`.mlx`*-filer).*

1. Olinjär skalär ekvation

Vi vill bestämma samtliga rötter till följande skalära ekvation,

$$x^2 - 8x - 5\sin(3x+1) + 12 = 0. \tag{1}$$

a) Plotta $f(x) = x^2 - 8x - 5\sin(3x+1) + 12$. Samtliga nollställen till f skall vara med. Notera ungefärliga värden på nollställena. (Dessa kommer ni använda som startgissningar i metoderna nedan.)

- Hur många nollställen finns det?

b) Skriv en MATLAB-funktion som beräknar nollställena till (1) med hjälp av fixpunktsiterationen

$$x_{n+1} = \frac{1}{12}x_n^2 + \frac{1}{3}x_n - \frac{5}{12}\sin(3x_n+1) + 1. \tag{2}$$

Funktionen ska ta som indata en startgissning x_0 och en feltolerans τ. Den ska skriva ut x_n och skillnaden $|x_{n+1} - x_n|$ efter varje iteration, samt returnera ett svar med ett fel som är mindre än τ. (Använd lämpligt avbrottsvillkor för att säkerställa detta.)

Använd funktionen för att empiriskt undersöka vilka nollställen till (1) som ni kan bestämma med fixpunktiterationen. Beräkna dessa nollställen med ett fel mindre än 10^{-10}. Vad blir värdena? Använd `format long` för att se alla decimaler.

- Motivera varför (2) är en fixpunktiteration för (1).
- Använd teorin för att förklara vilka nollställen fixpunktiterationen kan hitta.
- Hur ska skillnaderna $|x_{n+1} - x_n|$ avta enligt teorin? Verifiera att det stämmer i praktiken.

Skicka in: Funktionsfilen `fixpunkt.m`. Den ska kunna anropas som `"xrot = fixpunkt(x0,tau)"`. Variablerna `xrot`, `x0` och `tau` ska alla vara skalärer (ej vektorer).

c) Skriv en MATLAB-funktion som beräknar nollställena till (1) med hjälp av Newtons metod. Funktionen ska i övrigt bete sig på samma sätt som funktionen i (b) ovan. (Samma indata, utdata och utskrifter.) Beräkna samtliga nollställen med ett fel mindre än 10^{-10}. Vad blir värdena?

- Stämmer tumregeln för Newtons metod att antalet korrekta siffror dubblas i varje iteration?
- Hur ska skillnaderna $|x_{n+1} - x_n|$ avta enligt teorin? Verifiera att det stämmer i praktiken.

Skicka in: Funktionsfilen `newton.m`. Den ska kunna anropas som `"xrot = newton(x0,tau)"`. Variablerna `xrot`, `x0` och `tau` ska alla vara skalärer (ej vektorer).

d) Ni ska nu jämföra konvergensen för de två metoderna noggrannare. Välj den minsta av rötterna där bägge metoderna fungerar. Gör först en mycket noggrann (15 korrekta siffror) referenslösning med Newtons metod som ni använder för att beräkna felen nedan.

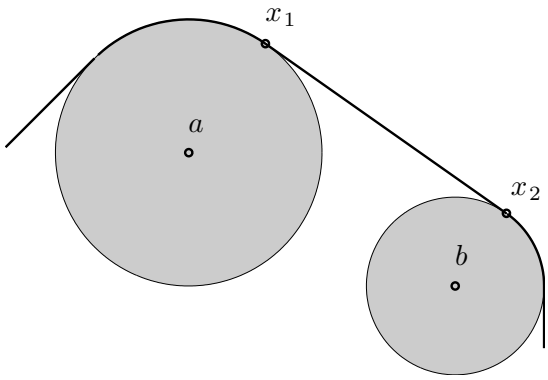
1. Plotta felet $e_n = |x_n - x|$ efter iteration n som funktion av n för båda metoderna i samma figur när de initieras med samma startgissning x_0. Använd `semilogy`-plot för att få logskala på y-axeln.
   - Vilken av metoderna konvergerar snabbast?
2. Ett precist sätt att kvantifiera hur snabbt metoden konvergerar är konvergensordningen. Den beskriver hur mycket mindre felet e_{n+1} är jämfört med e_n; konvergensordningen är p om $e_{n+1} \sim e_n^p$ när $n \to \infty$. Plotta därför e_{n+1} som en funktion av e_n i en `loglog`-plot för båda metoderna i samma figur.
   - Uppskatta metodernas konvergensordning med hjälp av figuren. Stämmer det med teorin?

Tips: Modifiera dina funktionsfiler från (b) och (c) så att de returnerar en vektor med *alla* approximationer $x_0, x_1, x_2, \ldots$ istället för bara den sista (bästa) approximationen `xrot`.

Skicka in: Figurerna `konvplot1.png` och `konvplot2.png`. (Använd kommandot `"print -dpng konvplot.png"`.) Ingen Matlab-kod behöver skickas in här.

2. Snöre runt cirklar

Ett snöre spänns runt två cirkelskivor enligt figuren. Cirklarna har radierna r_a respektive r_b och är centrerade i punkterna $\boldsymbol{a}$ respektive $\boldsymbol{b}$. Låt $\boldsymbol{x}_1$ och $\boldsymbol{x}_2$ vara de punkter (markerade i figuren) där snöret släpper från cirkeln. Dessa punkter kommer att satisfiera ekvationssystemet



$$
\begin{aligned}
|\boldsymbol{x}_1 - \boldsymbol{a}|^2 &= r_a^2, \\
|\boldsymbol{x}_2 - \boldsymbol{b}|^2 &= r_b^2, \\
(\boldsymbol{x}_1 - \boldsymbol{x}_2) \cdot (\boldsymbol{x}_1 - \boldsymbol{a}) &= 0, \\
(\boldsymbol{x}_1 - \boldsymbol{x}_2) \cdot (\boldsymbol{x}_2 - \boldsymbol{b}) &= 0.
\end{aligned}
$$

(Med $\boldsymbol{x} \cdot \boldsymbol{y}$ menas här skalärprodukten mellan vektorerna $\boldsymbol{x}$ och $\boldsymbol{y}$.) De två första ekvationerna kommer från kravet att $\boldsymbol{x}_1$ och $\boldsymbol{x}_2$ ligger på respektive cirkelrand. De två sista ekvationerna ges av att snöret är tangentiellt med cirkelranden vid punkterna, så att vektorn $\boldsymbol{x}_1 - \boldsymbol{x}_2$ är vinkelrät mot både $\boldsymbol{x}_1 - \boldsymbol{a}$ och $\boldsymbol{x}_2 - \boldsymbol{b}$.

a) Förberedande arbete (tas även upp på Övning 2, 1/2)

- Låt $\boldsymbol{x}_1 = (x_1, y_1)$ och $\boldsymbol{x}_2 = (x_2, y_2)$. Skriv om ekvationssystemet på formen $\boldsymbol{F}(x_1, y_1, x_2, y_2) = 0$ där $\boldsymbol{F}$ är en vektorvärd funktion med fyra komponenter. Radierna r_a, r_b och cirklarnas medelpunkter $\boldsymbol{a} = (x_a, y_a)$, $\boldsymbol{b} = (x_b, y_b)$ kommer in som parametrar i ekvationerna.
- Beräkna jakobian-matrisen till $\boldsymbol{F}$.

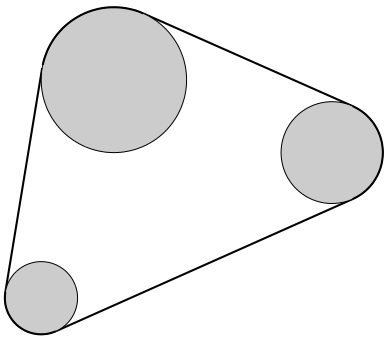
b) Skriv en MATLAB-funktion som löser ekvationsystemet med Newtons metod. Funktionen ska ta som indata en startgissning på vektorn $\mathbf{X} = (x_1, y_1, x_2, y_2)^T$, en feltolerans τ och värden på parametrarna för det aktuella fallet, dvs radierna r_a, r_b och mittpunkterna $\boldsymbol{a} = (x_a, y_a)^T$, $\boldsymbol{b} = (x_b, y_b)^T$. Funktionen ska returnera en lösningsvektor $\mathbf{X}_{\text{rot}}$ med ett fel mindre än τ. Funktionen ska också skriva ut mellanresultat (tex skillnaden mellan successiva iterationer) som visar att implementationen har kvadratisk konvergensordning.

Lös ekvationssystemet för fallet $r_a = 1.3$, $r_b = 0.6$, $\boldsymbol{a} = (-2,\ 2)^T$, $\boldsymbol{b} = (1,\ 0)^T$ och toleransen 10^{-10}. Plotta cirklarna och snöret i en figur och skriv ut svaret (punkternas koordinater).

- Hur många olika lösningar finns det? Hur ser de ut?

Skicka in: Funktionsfilen `punkter.m`. Den ska kunna anropas som "`Xrot=punkter(X0,ra,rb,a,b,tau)`", där `Xrot, X0` är en kolumnvektorer med 4 element, `a, b` är kolumnvektorer med två element och `ra, rb, tau` är skalärer. (OBS! Var noga med att använda kolumnvektorer, inte radvektorer.)

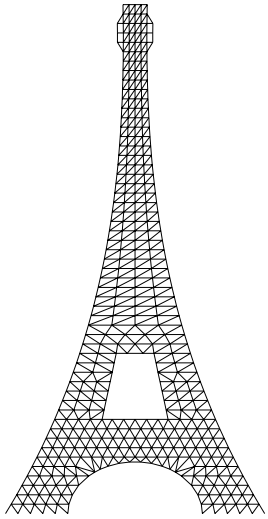
c) Ni ska nu beräkna längden på ett snöre som spänts runt tre cirklar med radierna $r_a = 0.9$, $r_b = 1.4$, $r_c = 0.6$ centrerade i punkterna $\boldsymbol{a} = (0.0,\ 1.3)^T$, $\boldsymbol{b} = (3.0,\ 0.5)^T$, $\boldsymbol{c} = (0.5,\ -2.0)^T$. Skriv ett MATLAB-program (ett script) som använder funktionen `punkter.m` som ni skrev i (b) för att göra detta. Plotta cirklarna och snöret i en figur. Programmet ska skriva ut snörets längd med tio korrekta decimaler.



Skicka in: Filen `langd.m` som innehåller programmet.

3. Stora matriser

I många realistiska tillämpningar måste man lösa *stora* linjära ekvationsystem, med miljontals obekanta. Det är i dessa fall som effektiva algoritmer blir viktiga att använda. Som exempel ska ni här räkna på ett komplicerat fackverk: en modell av Eiffeltornet. Ett fackverk består av stänger förenade genom leder i ett antal noder. Ni ska beräkna deformationen av fackverket när noderna belastas av yttre krafter. Ekvationerna för deformationen härleds i hållfasthetsläran, och baseras på att förskjutningarna i varje nod är små, och att Hookes lag gäller för förlängningen av varje stång.



Modellen i `eiffel2.mat`, med 399 noder (798 obekanta).

I slutändan får man ett linjärt ekvationssystem på formen $A\boldsymbol{x} = \boldsymbol{b}$. När antalet noder i fackverket är n kommer antalet obekanta vara $2n$ och $A \in \mathbb{R}^{2n\times 2n}$. Matrisen A brukar kallas *styvhetsmatrisen*. Högerledet $\boldsymbol{b}$ innehåller de givna yttre krafterna som verkar på noderna,

$$\boldsymbol{b} = \Big(F_1^x, F_1^y, F_2^x, F_2^y, \ldots, F_n^x, F_n^y\Big)^T, \qquad \boldsymbol{b} \in \mathbb{R}^{2n},$$

där $\boldsymbol{F}_j = (F_j^x, F_j^y)^T$ är kraften i nod j. Lösningen $\boldsymbol{x}$ innehåller de resulterande (obekanta) förskjutningarna,

$$\boldsymbol{x} = \Big(\Delta x_1, \Delta y_1, \Delta x_2, \Delta y_2, \ldots, \Delta x_n, \Delta y_n\Big)^T, \qquad \boldsymbol{x} \in \mathbb{R}^{2n}.$$

Här är alltså $(\Delta x_j, \Delta y_j)^T$ förskjutningen av nod j när fackverket belastas med krafterna i $\boldsymbol{b}$.

På kurshemsidan finns filerna `eiffel1.mat`, `eiffel2.mat`, `eiffel3.mat` och `eiffel4.mat`. De innehåller fyra olika modeller av Eiffeltornet med växande detaljrikedom ($n = 261$, 399, 561, 1592). Varje modell består av nodkoordinater i vektorerna `xnod, ynod`, stångindex i matrisen `bars` (används bara för plottningen) och styvhetsmatrisen `A`.

a) Ladda in en av modellerna i MATLAB med kommandot `load`. Hämta funktionsfilen `trussplot.m` från kurshemsidan och anropa den med `trussplot(xnod,ynod,bars)` för att plotta tornet. Välj en av noderna och belasta den med en kraft rakt högerut med beloppet ett. (Sätt $F_j^x = 1$ för något j, och resten av elementen i $\boldsymbol{b}$ lika med noll, dvs `b=zeros(2*n,1); b(j*2-1)=1;` i MATLAB.) Lös systemet $A\boldsymbol{x} = \boldsymbol{b}$ med backslash för att få fram förskjutningarna i alla punkter. Beräkna de nya koordinaterna för det belastade tornet, $x_j^{\text{bel}} = x_j + \Delta x_j$, etc.:

```
xbel = xnod + x(1:2:end); ybel = ynod + x(2:2:end);
```

Plotta det belastade tornet. Använd `hold on` för att plotta de två tornen ovanpå varandra i samma figur. Markera vilken nod ni valt med en asterisk * i figuren.

b) Backslash-kommandot i MATLAB använder normalt vanlig gausseliminering. Undersök hur tidsåtgången för gausseliminering beror på systemmatrisens storlek genom att lösa ekvationsystemet $A\boldsymbol{x} = \boldsymbol{b}$ med ett godtyckligt valt högerled $\boldsymbol{b}$ (tex med `b=randn(N,1)`) för var och en av de fyra modellerna. Använd MATLAB-kommandona `tic` och `toc` (`help tic` ger mer info).[1] Plotta tidsåtgången mot antal obekanta $N = 2n$ i en `loglog`-plot; använd också `grid on`.

[1] För att få bra noggrannhet i tidsmätningen (speciellt om den är kort) kan man mäta totaltiden för flera upprepade beräkningar, och sedan dividera tiden med antalet upprepningar.

- Hur ska tidsåtgången bero på N enligt teorin? Stämmer det överens med din plot?

Skicka in: Figuren `tidsplot.png`. (Använd kommandto "`print -dpng tidsplot.png`".)

c) Ni ska nu skriva ett MATLAB-program som räknar ut i vilka noder fackverket är mest respektive minst känslig för *vertikal* belastning. Använd den minsta modellen, `eiffel1.mat`. Tag en nod i taget. Belasta den med kraften $F_j^y = -1$ (istället för F_j^x) och räkna ut resulterande förskjutningar $\boldsymbol{x}$. Notera storleken på den totala förskjutningen i euklidiska normen,

$$||\boldsymbol{x}|| = \left(\sum_{j=1}^{n} \Delta x_j^2 + \Delta y_j^2 \right)^{1/2} .$$

Fortsätt med nästa nod, etc. Systematisera beräkningarna med en `for`-slinga i ert MATLAB-program och spara storleken på förskjutningen för varje nod. Ta sedan reda på vilken nod som ger störst respektive minst total förskjutning. Programmet ska slutligen plotta tornet med `trussplot` och markera dessa mest och minst känsliga noder i figuren med cirkel `o` respektive asterisk *.

d) I c) löser ni samma stora linjära ekvationssystem med många olika högerled (n stycken). För de större modellerna blir detta mycket tidskrävande. Optimera programmet genom att använda LU-faktorisering av A (MATLAB-kommandot `lu`). Lös problemet i c) för var och en av de fyra modellerna, med och utan LU-faktorisering[2]. Bestäm tidsåtgången för varje fall. (Exklusive tiden för `load` och plottning!) Skapa en tabell av följande typ och fyll i tiderna i de första två kolumnerna.

	Naiv	LU	Gles (ej LU)	Gles+LU
`eiffel1.mat`				
`eiffel2.mat`				
`eiffel3.mat`				
`eiffel4.mat`				

Skicka in: Filen `kanslighetLU.m` med programmet som använder LU-faktorisering och den minsta modellen. Programmet ska inkludera plottningen av tornet+minst/mest känsliga noderna.

e) När en matris är *gles* kan man lösa ekvationssystemet med effektivare metoder än standard gausseliminering. Använd kommandot `spy(A)` för att förvissa er om att styvhetsmatrisen är gles (bandad). Tala om för MATLAB att matrisen är gles genom att konvertera format med kommandot `A=sparse(A)`. Bättre metoder kommer då automatiskt användas när backslash anropas. Gå igenom beräkningarna i d) igen och undersök tidsåtgången när MATLAB använder dessa glesa lösare. Fyll i dessa tider i de två sista kolumnerna i tabellen ovan.

- Varför går det snabbare att lösa problemet med LU-faktorisering?
- Vilken metod löser problemet snabbast? (Med/utan LU? Full/gles lösare?)
- För vilken modell blir tidsvinsten störst? Varför?

Skicka in: Tabellen `tidtabell` över de uppmätta tiderna för alla fyra metoderna, för de modeller som inte tar orimligt mycket tid. Formatet kan vara vanligt textformat eller PDF (men ej `.doc`, `.rtf` eller `.odt`).

[2]Ni kan hoppa över de fall som tar orimligt lång tid.

4. Frivillig uppgift: Ekvationer med osäkra parametrar

Vi återvänder till första uppgiften och antar nu att ett par parametrar i den bestämts genom mätningar och därför innehåller en viss osäkerhet. Mer precist studerar vi

$$f(x) = x^2 - 8x - a\sin(bx + 1) + 12 = 0,$$

när värdena på parametrarna a och b uppskattas till $a = 5.0 \pm 0.5$ och $b = 3.0 \pm 0.1$.

a) Visualisera osäkerheten i funktionen f för intervallet $x \in [1, 10]$ på följande sätt. Plotta först funktionen $y = f(x)$ som funktion av x, med $a = 5$ och $b = 3$. Stör sedan a och b slumpmässigt inom felgränserna, tex med MATLAB-kommandona `a = 5 + (2*rand-1)*0.5` och `b = 3 + (2*rand-1)*0.1`. Plotta funktionen med dessa a- och b-värden i samma figur (använd `hold on`). Upprepa många (>100) gånger.

b) Vi vill beräkna $y = f(x)$ när $x = 5$. Bestäm felgränsen för y med felfortplantningsformeln.

- Hur kan man uppskatta denna felgräns ur figuren i a)? Stämmer värdet med teorin?

c) Antag nu att även x kommer från en osäker källa så att $x = 5.0 \pm 0.1$. Bestäm felgränsen för y med felfortplantningsformeln i detta fall.

- Hur kan man uppskatta denna felgräns ur figuren i a)? Stämmer värdet med teorin?

d) Vi ska nu hitta rötterna till ekvationen $f(x) = 0$. På grund av osäkerheten i a och b kommer även rötterna ha en osäkerhet. Bestäm felgränser för den minsta och största roten med experimentell störningsräkning. (Beräkna rötterna med hjälp av Matlab-programmen du gjorde i uppgift 1.)

- Vilka av rötterna påverkas mest/minst av osäkerheten i parametrarna? Hur ser man det i figuren i a)?
- Vad händer när osäkerheten i parametrarna blir större, tex om $b = 3.0 \pm 0.3$? Speciellt, vad händer då i intervallet $[5, 7]$?

e) Bestäm felgränser för rötterna teoretiskt med felfortplantningsformeln. Eftersom rötterna är implicit definierade funktioner av parametrarna a, b behöver man använda implicit derivering.

- Jämför resultaten i d) och e). Varför blir de inte exakt samma?

f) Ni ska nu bestämma osäkerheten i snörets längd i uppgift 2, när vi antar att osäkerheten i alla parametrar (radier, mittpunkternas koordinater) är ± 0.05. Detta är mycket komplicerat att göra med felfortplantningsformeln, så ni ska använda experimentell störningsräkning.

Skriv först om MATLAB-programmet i (2c) till en funktion som tar de tre radierna och mittpunkterna som indata och returnerar snörets längd. I funktionen ska toleransen och startgissningarna vara fixa, valda så att de fungerar för problemet i (2c).

Skriv sedan ett huvudprogram i MATLAB som använder experimentell störningsanalys för att bestämma osäkerheten. Programmet ska anropa funktionen ovan högst 10 gånger och skriva ut osäkerheten.