Tillämpningar i matematik med PYTHON

1 Inledning

Det finns tre språk som är vanliga att använda när man utför numeriska och tekniska beräkningar. Dessa är MATLAB, PYTHON och OCTAVE. MATLAB skapades på tidigt 1980-tal av Cleve Moler[1] och är en miljö för tekniska och numeriska beräkningar. PYTHON är ett programspråk som lanserades första gången 1991 av Guido van Rossum[2]. OCTAVE är en miljö för numeriska beräkningar och skapades från början som en fri kopia av MATLAB, det gör att OCTAVE och MATLAB är väldigt lika varandra. PYTHON och OCTAVE är s.k. fri programvara och du kan fritt ladda hem dem till din egen dator från nätet. PYTHON har hemsida `www.python.org` och OCTAVE har hemsida `https://octave.org`. MATLAB är licensierad programvara, men Chalmers prenumererar på licenser för alla studenter och anställda på Chalmers, därför kan du även fritt ladda ner MATLAB till din egen dator så länge som du är student på Chalmers. MATLAB ägs av företaget `Mathworks` som har hemsida `https://www.mathworks.com`.

Alla tre språken bygger på programspråket C och använder funktioner från paketen `Linpack` och `LAPACK` för numeriska beräkningar. MATLAB är äldst av de tre språken och fortfarande den mest tongivande miljön när det gäller numeriska beräkningar, även om PYTHON och OCTAVE numera också är väldigt stora.

Nedan följer laborationen Tillämpningar i matematik med MATLAB en gång till, fast denna gången med tillämpningar i PYTHON. Jag har skrivit MATLAB-koden och PYTHON-koden brevid varandra så att du ska kunna jämföra de två språken. Mycket är likt, men en hel del detaljer skiljer sig åt. Du får gärna lösa uppgifterna nedan i PYTHON, men det är helt frivilligt.

2 Nollställen

En lösning x^* till en ekvation $f(x) = 0$ kallas ett nollställe eller en rot. Om vi inte kan få fram någon formel för att lösa en ekvation så kan vi beräkna en approximation med t.ex. **Newtons metod**: Antag att x_k är en approximation av ett nollställe till ekvationen $f(x) = 0$. Följ tangenten i punkten $(x_k, f(x_k))$, dvs.

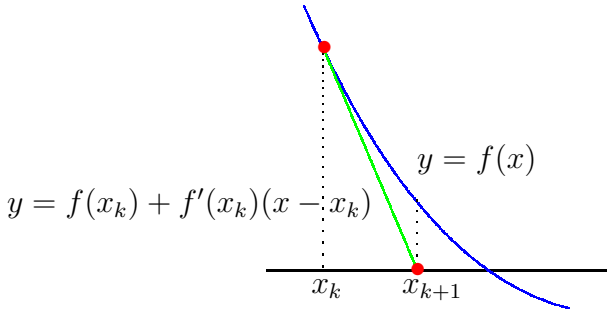
$$y = f(x_k) + f'(x_k)(x - x_k)$$

ned till x-axeln ($y = 0$) och tag skärningspunktens x-koordinat

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

som en ny approximation av nollstället.

[1] `https://en.wikipedia.org/wiki/Cleve_Moler`
[2] `https://en.wikipedia.org/wiki/Guido_van_Rossum`



Exempel 1. Vi skall använda Newtons metod för att beräkna nollställena till funktionen

$$f(x) = \cos(x) - x$$

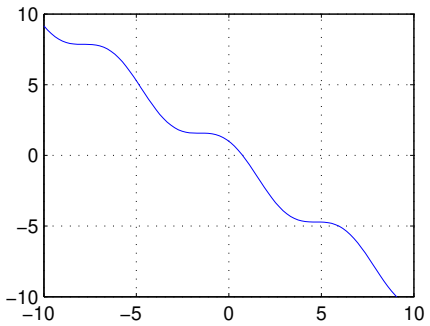
Vi börjar med att rita en graf av funktionen så att vi ser hur många nollställen vi har och ungefär var de ligger.
Vi beskriver funktionen och ritar grafen med

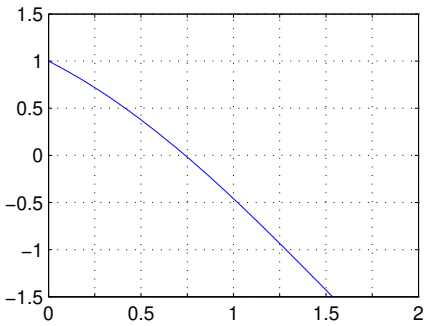
MATLAB

```
>> f=@(x)cos(x)-x;
>> x=linspace(-10,10);
>> plot(x,f(x))
```

PYTHON

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> def f(x):
        return np.cos(x)-x
>>> x=np.linspace(-10,10,100)
>>> plt.plot(x,f(x))
```





Från grafen till höger, som är över ett lite mindre intervall, ser vi att vi har ett nollställe nära $x_0 = 0.75$ som vi tar som startapproximation för Newtons metod.

MATLAB

```
>> Df=@(x)-sin(x)-1;
>> x=0.75;
>> kmax=10; tol=0.5e-8;
>> for k=1:kmax
       h=-f(x)/Df(x);
       x=x+h;
       disp([x h])
       if abs(h)<tol, break, end
   end
```

PYTHON

```
>>> def Df(x):
        return -np.sin(x)-1
>>> x=0.75
>>> kmax=10; tol=0.5e-8
>>> for k in range(kmax):
        h=-f(x)/Df(x)
        x=x+h
        print(x,h)
        if abs(h)<tol:
           break
```

```
0.739111138752579  -0.010888861247421
0.739085133364485  -0.000026005388094
0.739085133215161  -0.000000000149324
```

I kolumnen till vänster ser vi x_k-värdena och i den till höger ser vi motsvarande $f(x_k)$-värden. Vi ser att vi får snabb konvergens, dvs. vi får snabbt ett noggrant resultat. Iterationen avbryts eftersom vi har mer är åtta korrekta decimaler (felet mindre än $\frac{1}{2} \times 10^{-8}$).

Kommentarer till koden i PYTHON

Precis som i MATLAB kan man skriva kommandon i Console, men även skapa skript med programkod. I exemplen har jag markerat att vi skrivit i MATLAB's Console med >> och i PYTHON's Console med >>>.

Språket PYTHON är ganska litet och man jobbar med paket som man inkluderar (jämför med MATLABs toolboxar). I paketet `numpy` finns många av de matematiska funktioner vi använder i samband med numeriska beräkningar. T.ex. de trigonometriska funktionerna, approximationer till vanliga konstanter, t.ex. π, e. Funktionerna i `numpy` fungerar för vektorer och matriser och här finns även kommandon för att skapa vektorer och matriser. Om vi skriver

```
import numpy as np
```

får vi tillgång till funktonerna i `numpy`. Vi anropar de funktioner vi behöver genom att skriva `np.` framför funktionsnamnen. I programexemplen ovan hittar du bla. anropen `np.cos(x)`, `np.sin(x)` och `np.linspace(-10,10,100)`.

Paketet `numpy` har en egen hemsida där alla kommandon finns dokumenterade: `https://numpy.org`.

I paketet `matplotlib.pyplot` finns funktioner för att rita figurer. När man skapade paketet sneglade man på de funktioner som fanns i MATLAB för att skapa grafik. Många av kommandona i paketet har därför samma namn och fungerar på samma sätt som motsvarande kommando i MATLAB. För att få tillgång till funktionerna i `matplotlib.pyplot` är det vanligt att man skriver

```
import matplotlib.pyplot as plt
```

Man når sedan funktionerna i paketet genom att skriva `plt.` framför funktionsnamnen. I programexemplen ovan hittar vi anropet `plt.plot(x,f(x))`. Alla funktioner i `matplotlib.pyplot` finns dokumenterade hemsidan: `https://matplotlib.org`.

När man definerar egna funktioner i MATLAB kan man göra det på två olika sätt, man definerar en anonym funktion med hjälp av @, eller en extern funktion (`function`). I PYTHON definerar man funktioner genom att använda ordet `def` och får då en funktion som ungefär fungerar som en anonym funktion i MATLAB. För att beskriva vad funktionen returnerar använder man ordet `return` följt av värdet som ska returneras. I exemplen ovan hittar vi t.ex funktionsdefinitionen

```
def Df(x):
   return -np.sin(x)-1
```

Funktionen heter `Df`, den har en inparameter (`x`) och den returnerar (beräknar) $\sin(x) - 1$. Observera att första raden måste avslutas med kolon (`:`). Andra raden i funktionen (funktionskroppen) har dragits in några steg från vänstermarginalen. Det kallas för att indentera. I PYTHON avlsutas inte funktioner med `end`, utan man markerar var funktionen tar slut genom att sluta indentera i programmet. Vi hittar exemplet

```
def Df(x):
   return -np.sin(x)-1
x=0.75
```

i koden ovanför. Raden `x=0.75` tillhör alltså inte funktionen, utan är den rad som utförs efter att funktionen definerats[3].

I MATLAB-exemplen ovan hittar vi for-loopen

```
for k = 1:kmax
  ...
```

Variabeln `k` antar värdena $1, 2, \ldots,$ `kmax` allteftersom elementen i vektorn `1:kmax` genomlöps. I PYTHON skapar man en for-loop på ungefär samma sätt:

```
for k in range(kmax):
   ...
```

Anropet `range(kmax)` skapar vektorn[4] med värdena[5] $0, 1, \ldots$ `kmax` -1. Observera att satserna inne i for-loopen indenteras. Man markerar var loopen tar slut genom att sluta indentera satserna.

if-satser PYTHON fungerar och skrivs på ungefär samma sätt som i MATLAB. I koden ovanför hittar vi

```
if abs(h)<tol:
   break
```

Man skriver `if` följt av ett villkor. Relationsoperatorerna är de samma som i MATLAB, utom $\neq$ som skrivs `!=` i PYTHON. Om man vill ha flervalsalternativ använder man `elif` som motsvarar `elseif` i MATLAB. Precis som i loopar indenterar man satserna i respektive villkorsdel. Man markerar att if-satsen tar slut genom att inte längre indentera. Observera kolonet (:) efter villkoret.

Uppgift 1. Låt $f(x) = x^3 - \cos(4x)$. Lös ekvationen $f(x) = 0$. Rita upp grafen till f för att se var ungefär lösningarna (skärningspunkterna) ligger. Hur många lösningar finns det? Läs av i grafiken en första approximation av en lösning för att sedan förbättra denna med Newtons metod. Rita ut lösningen med en liten ring. Upprepa tills du beräknat alla lösningar till ekvationen.

[3] Om funktionskroppen bara består av en rad kan man skriva: `def Df(x):  return -np.sin(x)-1`

[4] Eg. ett itererbart objekt

[5] Om vi istället skriver `for k in range(1,kmax+1):` får vi värdena $1, 2, \ldots,$ `kmax`.

Färdiga program i MATLAB/PYTHON

Verktygslådan OPTIMIZATION TOOLBOX i MATLAB har en funktion `fzero` som finner nollställe till en given funktion och används enligt något av alternativen

```
x=fzero(fun,x0)      x=fzero(fun,x0,opts)
```

där `fun` beskriver funktionen vi skall finna nollstället till, `x0` är en första approximation av nollstället eller ett intervall med teckenväxling som omsluter nollstället vi söker. I senare fallet kommer `fzero` garanterat finna en approximation av ett nollställe.

Alternativet med `opts` använder vi då vi t.ex. vill ange hur noggrant lösningen skall beräknas. Man skapar vektorn `opts` med funktionen `optimset`, se hjälptexten.

Om vi skulle använda `fzero` för att beräkna ett av nollställena i uppgift 1 skulle det kunna se ut så här

```
>> f=@(x)x.^3-cos(4*x);
>> x0=-1;
>> x=fzero(f,x0)
```

I PYTHON kan vi använda funktionen `fsolve` från paketet `scipy.optimize` som finner nollställe till en given funktion och kan användas enligt alternativen nedan[6]:

```
x=fsolve(fun,x0)      x=fsolve(fun,x0,xtol=tol)
```

där `fun` beskriver funktionen vi skall finna nollstället till, `x0` är en första approximation av nollstället. i

Alternativet med `xtol=tol` använder vi då vi t.ex. vill ange hur noggrant lösningen skall beräknas.

Om vi skulle använda `fsolve` för att beräkna ett av nollställena i uppgift 1 skulle det kunna se ut så här

```
>>> import numpy as np
>>> from scipy.optimize import fsolve
>>> def f(x):
        return x**3-np.cos(4*x)
>>> x0=-1
>>> x=fsolve(f,x0)
```

(Upphöjt till skrivs `**` i PYTHON, operatorn opererar elementvis. Punktnotation `.^`, `.*`, `./` finns inte i PYTHON).

Exempel 2. Kastbana utan luftmotstånd (”Mer om funktioner och grafik i MATLAB” exempel 4) beskrivs av

$$y(x) = y_0 - \frac{g}{2v_0^2\cos^2(\theta)}\left(x - \frac{v_0^2\sin(2\theta)}{2g}\right)^2 + \frac{v_0^2\sin^2(\theta)}{2g}$$

där y_0 är utkasthöjden, v_0 är utkastfarten och θ är utkastvinkeln.

Hur långt når kastet om vi tar $y_0 = 1.85$, $v_0 = 10$ m/s och $\theta = 45°$?

MATLAB: Vi beskriver kastbanan med en funktion

[6]Fler möjligheter finns, se dokumentationen för `fsolve`, `https://docs.scipy.org`.

```
function y=kast(x,y0,v,t)
    g=9.81;
    a=g/(2*v^2*cos(t)^2); b=v^2*sin(2*t)/(2*g); c=v^2*sin(t)^2/(2*g);
    y=y0-a*(x-b).^2+c;
```

ritar graf med

```
>> y0=1.85; v0=10; t=45*pi/180;
>> x=linspace(0,14);
>> plot(x,kast(x,y0,v0,t),[x(1) x(end)],[0 0],'g')
```

och löser $y(x) = 0$ med

```
>> x=fzero(@(x)kast(x,y0,v0,t),[11,12])
x =
   11.7928
```

Lägg märke till hur vi använder ett anonymt funktionshandtag för att få med parametervärden. Det är bara `x` som `fzero` skall arbeta med.

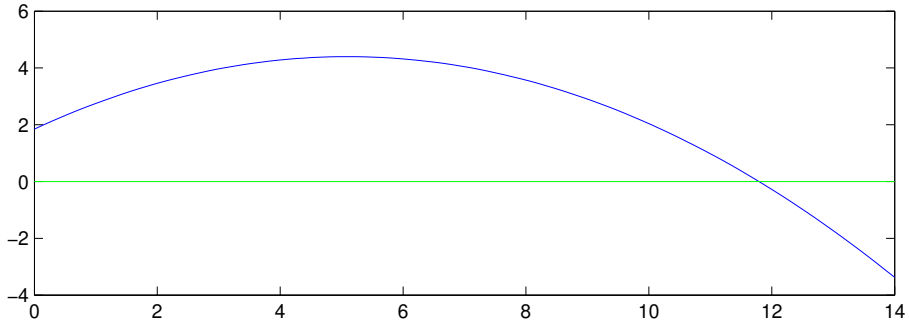
PYTHON: Vi beskriver kastbanan med en funktion

```
import numpy as np
def kast(x,y0,v,t):
    g=9.81
    a=g/(2*v**2*np.cos(t)**2); b=v**2*np.sin(2*t)/(2*g); c=v**2*np.sin(t)**2/(2*g)
    return y0-a*(x-b)**2+c
```

ritar graf med

```
>>> import matplotlib.pyplot as plt
>>> y0=1.85; v0=10; t=45*np.pi/180
>>> x=np.linspace(0,14);
>>> plt.plot(x,kast(x,y0,v0,t),'b',[x[0],x[-1]],[0,0],'g')
```

I PYTHON indexeras fält från 0 och framåt (i MATLAB är första index 1). Dessutom används hakparenteser. Anropet `x[0]` avser första elementet i `x` och `x[-1]` är det sista elementet i `x`. I MATLAB skriver vi `x(1)` respektive `x(end)`.



och löser $y(x) = 0$ med

```
>>> from scipy.optimize import fsolve
>>> x=fsolve(kast,11,args=(1.85,10,45*np.pi/180))
>>> print(x)
11.7928
```

Lägg märke till hur vi använder inparametern `args` till `fsolve` för att få med parametervärden. Det är bara den första parametern, `x`, som `fsolve` skall arbeta med. Kommandot `fsolve` finns beskrivet på `scipy.optimize` hemsida, `https://docs.scipy.org`.

Uppgift 2. Rita den slutna kurva som i polära koordinater ges av

$$r(\theta) = \frac{2 + \sin(3\theta)}{\sqrt{1 + \exp(\cos(\theta))}}$$

För vilka vinklar skär kurvan enhetscirkeln?
Ledning: Använd polära koordinater. Rita cirkeln och kurvan med `plot`. Läs in startvärden på vinklar med `ginput`[7] och beräkna sedan vinklarna noggrant med `fsolve` i PYTHON.

3 Optimering

Vill vi minimera en funktion $f(x)$ som är deriverbar så söker vi lokala minpunkter, t.ex. genom att lösa ekvationen $g(x) = f'(x) = 0$. Sedan får vi kontrollera om dessa är lokala eller globala minimum. Vill vi maximera funktionen så finner vi maxpunkt genom att minimera $h(x) = -f(x)$.

Det finns också andra metoder som söker lokalt minimum till en funktionen. Vi beskriver inte dessa nu utan nöjer oss med att se vad MATLAB och PYTHON har för färdiga program.

Färdiga program i MATLAB/PYTHON

Verktygslådan OPTIMIZATION TOOLBOX i MATLAB har en funktion `fminbnd` som finner lokal minpunkt till en given funktion och används enligt något av alternativen

```
x=fminbnd(fun,x1,x2)      x=fminbnd(fun,x1,x2,opts)
```

där `fun` beskriver funktionen vi skall minimera, `x1` och `x2` ger ett intervall som omsluter minpunkten vi söker. Alternativet med `opts` använder vi då vi t.ex. vill ange hur noggrant lösningen skall beräknas. Man skapar vektorn `opts` med funktionen `optimset`, se hjälptexten.

I PYTHON kan vi använda funktionen `fminbound` från paketet `scipy.optimize`.

Exempel 3. Vi söker lokala min- och maxpunkter till funktionen

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} + \frac{1}{(x-0.9)^2 + 0.04} - 6$$

Vi börjar med att beskriva funktionen och rita dess graf med

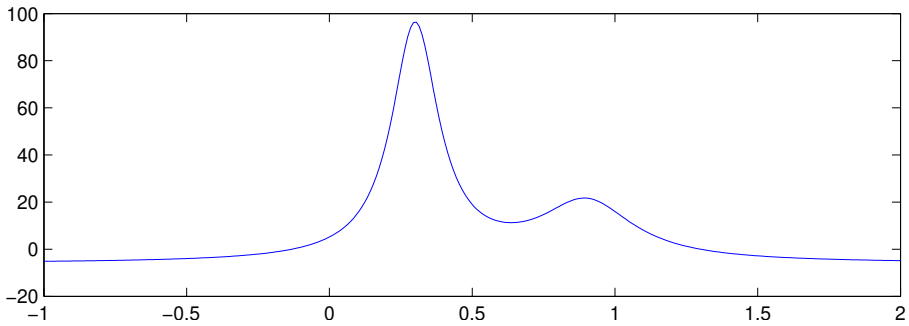
MATLAB

[7] Det finns en motsvarighet till MATLABS `ginput` i paketet `matplotlib.pyplot`, som heter `ginput`. Om kommandot kommer att fungera eller inte beror av vilken miljö du använder för att programmera PYTHON i, och hur parametrar i den miljön är inställda.

```
>> f=@(x)1./((x-0.3).^2+0.01)+1./((x-0.9).^2+0.04)-6;
>> x=linspace(-1,2,300);
>> plot(x,f(x))
```

PYTHON

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def f(x):
        return 1/((x-0.3)**2+0.01)+1/((x-0.9)**2+0.04)-6
>>> x=np.linspace(-1,2,300)
>>> plt.plot(x,f(x))
```



Vi bestämmer den lokala minpunkten enligt

MATLAB

```
>> x=fminbnd(f,0.5,0.8)
x =
    0.6370
```

PYTHON

```
>>> from scipy.optimize import fminbound
>>> x=fminbound(f,0.5,0.8)
>>> x
0.6370082119636189
```

Vi bestämmer sedan de lokala maxpunkterna, genom att minimera $h(x) = -f(x)$, med

MATLAB

```
>> h=@(x)-f(x);
>> x=fminbnd(h,0,0.5)
x =
    0.3004
```

PYTHON

```
>>> def h(x): return -f(x)
>>> x=fminbound(h,0,0.5)
>>> x
0.30037710643864535
```

På motsvarande sätt bestämmer vi den högra maximipunkten.

Uppgift 3. I samband med studiet av diffraktion vill man söka lokala maximum till funktionen

$$y(u) = \frac{\sin^2(u)}{u^2}$$

Hur många lokala maximipunkter finns det? Bestäm den minsta (positiva) lokala maximipunkten och motsvarande lokala maximivärde.

4 Integraler

Ibland kan man inte bestämma integraler exakt utan man får nöja sig med att beräkna approximationer. T.ex. $\int_0^1 \exp(x^2)\,dx$ kan inte beräknas exakt, eftersom det inte finns någon användbar primitiv funktion. Det kan också vara så att integranden bara är känd i vissa punkter, t.ex. vi har en serie med mätdata.

Den geometriska tolkningen av integralen $\int_a^b f(x)\,dx$ är arean av ytan mellan grafen av integranden $y = f(x)$ och x-axeln, dvs. $y = 0$, mellan $x = a$ och $x = b$.

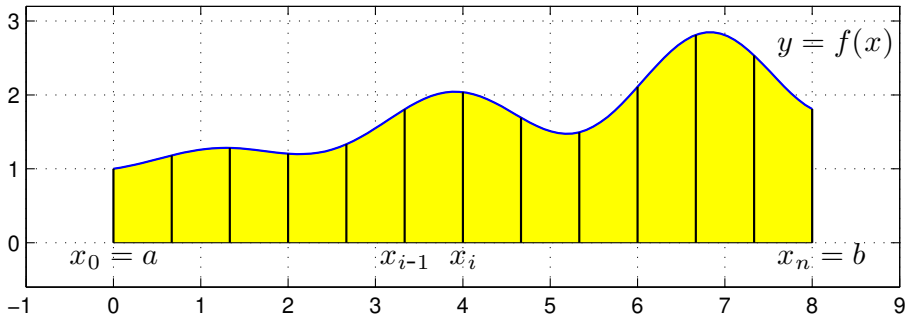
Vi gör en likformig indelning av intervallet $a \le x \le b$

$$a = x_0 < x_1 < x_2 < \cdots < x_{i-1} < x_i < \cdots < x_{n-1} < x_n = b$$

så att vi får n lika långa delintervall $x_{i-1} \le x \le x_i$ med bredden $h = \frac{b-a}{n}$.

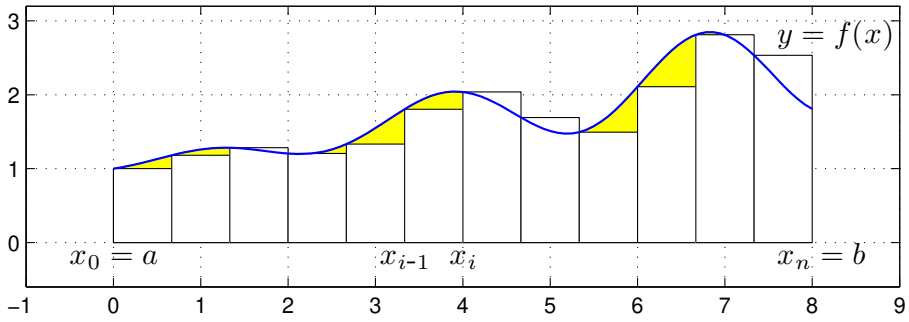
Sedan delar vi upp integralen i en summa av delintegraler över varje delintervall

$$\int_a^b f(x)\,dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)\,dx$$



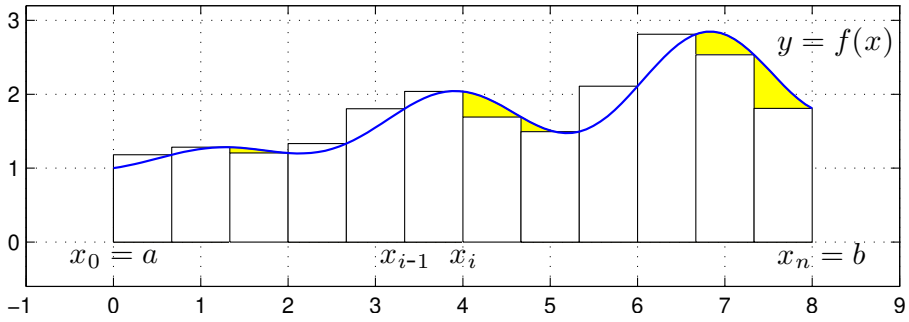
Om vi approximerar $f(x)$ med $f(x_{i-1})$ i intervallen $x_{i-1} \le x \le x_i$ får vi **vänster rektangelregel**

$$\int_a^b f(x)\,dx \approx L_n = \sum_{i=1}^{n} h\, f(x_{i-1})$$



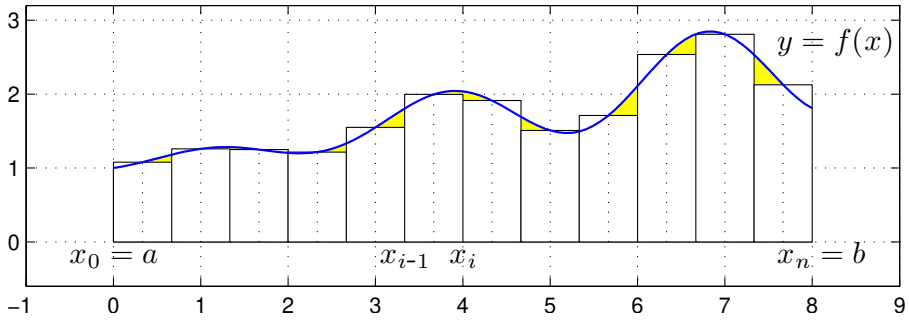
Om vi approximerar $f(x)$ med $f(x_i)$ i intervallen $x_{i-1} \le x \le x_i$ får vi **höger rektangelregel**

$$\int_a^b f(x)\,dx \approx R_n = \sum_{i=1}^{n} h\, f(x_i)$$



Om vi approximerar $f(x)$ med $f(m_i)$ i intervallen $x_{i-1} \leq x \leq x_i$, där m_i är mittpunkterna i intervallen, får vi **mittpunktsmetoden**

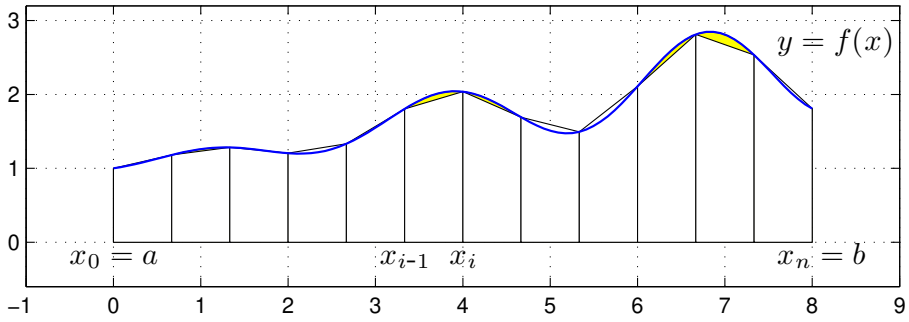
$$\int_a^b f(x)\,dx \approx M_n = \sum_{i=1}^{n} h\, f\left(\frac{x_{i-1} + x_i}{2}\right)$$



Dessa olika varianter är Riemannsummor (med $\xi_i = x_{i-1}$, $\xi_i = x_i$ respektive $\xi_i = m_i$).

Vi kan också approximera integralen med medelvärdet av vänster och höger rektangelregel och får då **trapetsmetoden**

$$\int_a^b f(x)\,dx \approx T_n = \sum_{i=1}^{n} \frac{h}{2}\,\left(f(x_{i-1}) + f(x_i)\right)$$



Namnet på denna metod kommer från de parallelltrapetser som approximerar i varje delintervall.

Antag att vi vill beräkna $\int_0^1 x \sin(x)\,dx$ med vänster rektangelregel med $n = 100$.

Vi skulle kunna göra så här

MATLAB

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

PYTHON

```
>>> import numpy as np
>>> def f(x): return x*np.sin(x)
>>> n=100; a=0; b=1
>>> h=(b-a)/n
>>> q=0
>>> for i in range(n):
       x=a+i*h
       q=q+h*f(x)
>>> q
```

Vi kan generera en vektor av alla funktionsvärden $f(x_i)$ och sedan summerar dessa enligt

MATLAB

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

PYTHON

```
>>> import numpy as np
>>> def f(x): return x*np.sin(x)
>>> n=100; a=0; b=1
>>> x=np.linspace(a,b,n+1)
>>> h=(b-a)/n
>>> q=sum(h*f(x[0:n]))
```

Detta sätt att organisera en beräkning kallas att vektorisera den, dvs. man genererar först en eller flera vektorer och utför sedan den önskade beräkningen på dem. De elementvisa operationerna `.*` `./` `.^` i MATLAB är exempel på vektoriserade operationer. I PYTHON är motsvarande operatorer `*`, `/` och `**` elementvisa på vektorer och matriser skapade med funktioner från `numpy`. (Vektorn `x` skapas ju med anropet `x=np.linspace(a,b,n+1)`).

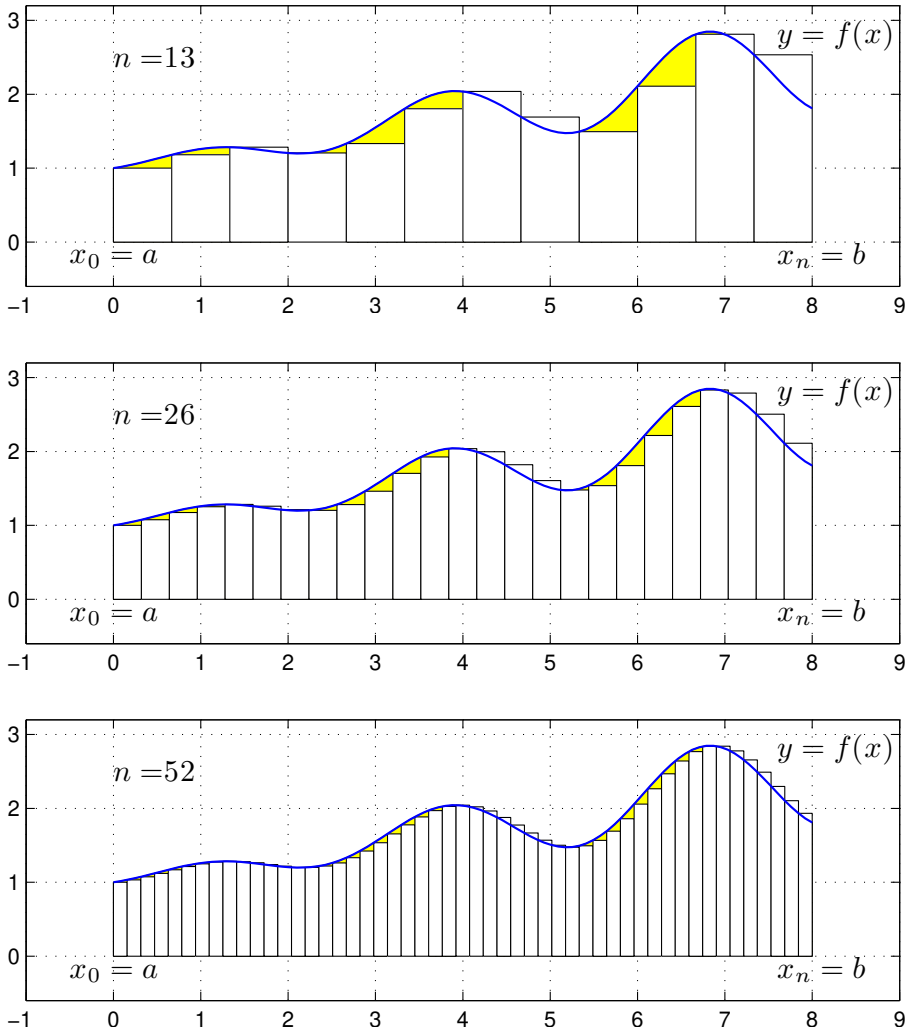
Med funktionen `sum` summeras elementen i vektorn. Det finns även en funktion `sum` definerad i `numpy`. Om vi istället vill använda den kan vi skriva `q=np.sum(h*f(x[0:n])` på sista raden i PYTHON-koden.

Vektorer i MATLAB indexeras med start från 1. När vi skriver `x(1:n)` i MATLAB menar vi elementen från och med 1 till och med n i vektorn `x`. Vektorer från `numpy` i PYTHON indexeras från 0, och när vi skriver `x[0:n]` i PYTHON menar vi elementen från och med 0 till n (till n inte till och med n). Lägg också märke till att man i PYTHON använder hakparenteser när man indexerar i vektorer.

Uppgift 4. Beräkna en approximation av integralen $\int_0^1 x\sin(x)\,dx$ med vänster och höger rektangelregel samt mittpunkts- och trapetsreglerna. Använd `sum`.

För metoderna ovan gäller att samtliga är konvergenta, dvs. låter vi antal delintervall n gå mot oändligheten så går approximationerna mot integralens värde.

Vi ser på några bilder för vänster rektangelregel där n blir allt större



Vi ser att vi allt bättre täcker upp ytan under grafen med allt fler och smalare staplar.

Nu räcker det i praktiken inte med konvergens. Vi måste få en bra approximation på en kort tid, dvs. inte behöva ta n alltför stort.

För vänster och höger rektangelregel gäller att om vi fördubblar antal delintervall så halveras felet i approximationen av integralen. För mittpunkts- och trapetsmetoderna gäller, vid samma fördubbling av antal delintervall, att felet delas med fyra, dvs. mycket bättre utdelning.

Färdiga program i MATLAB/PYTHON

Det finns funktioner i MATLAB för att beräkna bestämda integraler

$$\int_a^b f(x)\,dx$$

både effektivt och noggrant, t.ex. finner vi `integral` som används enligt

```
q=integral(fun,a,b)      q=integral(fun,a,b,name,value)
```

där `fun` är en funktion som beskriver integranden, `a` och `b` ger integrationsintervallet.

Om vi vill använda `integral` för att beräkna integralen $\int_0^1 x\sin(x)\,dx$ så skulle det se ut så här

```
>> a=0; b=1; f=@(x)x.*sin(x);
>> q=integral(f,a,b)
```

Genom att ge `name` som `'AbsTol'` eller `'RelTol'` kan vi ange önskad absolut respektive relativ noggrannhet i beräkningen och med `value` ger vi värdet på noggrannheten.

PYTHON är inte riktigt lika bra på att beräkna integraler som MATLAB. Men vi kan t.ex. använda kommandot `quad` från paktet `scipy.integrate`. Om vi vill använda `quad` för att beräkna integralen $\int_0^1 x\sin(x)\,dx$ i PYTHON så skulle det se ut så här

```
>>> import numpy as np
>>> from scipy.integrate import quad
>>> def f(x): return x*np.sin(x)
>>> a=0; b=1
>>> q=quad(f,a,b)
```

`quad` returnerar två värden

```
>>> print(q)
(0.3011686789397568, 3.3436440165485948e-15)
```

Det första värdet i tupeln är det beräknade värdet på integralen och det andra värdet är en skattning av hur stort felet i det beräknade värdet är. (`quad` finns även representerad i MATLAB, men är på väg bort. Rekommendationen är att man använder det aktuella kommandot `integral` om man vill beräkna integraler i MATLAB.)

Uppgift 5. Beräkna arean av det slutna området mellan graferna till funktionerna

$$g(x) = \exp(-\frac{x^2}{2}) \quad \text{och} \quad h(x) = x^2 - 3x + 2\,.$$

Rita en bild av området. Använd `fsolve` för att beräkna skärningspunkterna mellan graferna och `quad` för att beräkna en approximation till integralen.

5 Differentialekvationer

Låt $u(x)$ vara en funktion. En ekvation som beskriver ett samband mellan x, u och derivator av u kallas, som säkert bekant, en differentialekvation. Differentialekvationens s.k. ordning är ordningen på högsta derivatan som ingår.

Den allmänna differentialekvationen av 1:a, 2:a, $\cdots$ ordningen har formerna

$$u' = f(x,u), \quad u'' = f(x,u,u'), \quad \cdots$$

En differentialekvation av ordning n kan alltid skrivas om som ett system av n kopplade ekvationer av 1:a ordningen.

Vi skall se på s.k. begynnelsevärdesproblem för ekvationer av 1:a ordningen

$$\begin{cases} u' = f(t,u),\ 0 \le t \le T \\ u(0) = u_0 \end{cases}$$

Här känner vi värdet av u vid $t = 0$. Vi använder t som variabel, eftersom t är ofta tiden och u tillståndet i ett mekaniskt system eller dylikt.

Färdiga program i MATLAB/PYTHON

I MATLAB finns flera olika funktioner för lösning av begynnelsevärdesproblem av olika karaktär, t.ex. finner vi `ode45` som används enligt

```
[t,U]=ode45(fun,tspan,u0)      [t,U]=ode45(fun,tspan,u0,opts)
```

där `fun` är en funktion som beskriver differentialekvationens högerled, `tspan` är en vektor som anger tidsintervallet och `u0` ger begynnelsevärdet för lösningen vid tiden `tspan(1)`. Funktionen `ode45` producerar som utdata dels `t`, en vektor som innehåller tidpunkter och dels `U`, en vektor eller matris som innehåller lösningen för de olika tidpunkterna.

Alternativet med `opts` använder vi då vi t.ex. vill ange hur noggrant lösningen skall beräknas. Det finns också möjlighet till avbrottshantering, mer om detta i läsperiod 4. Man skapar vektorn `opts` med funktionen `odeset`, se hjälptexten.

I PYTHON finns också många olika differentialekvationslösare. Vi kan t.ex. använda `solve_ivp` från paketet `scipy.integrate`. Anropen skrivs då ofta

```
sol=solve_ivp(fun,tspan,u0,dense_output=True)
```

Precis som i MATLAB-fallet ovan är `fun` funktionen som beskriver differentialekvationens högerled, `tspan` är en vektor som anger tidsintervallet och `u0` ger begynnelsevärdet vid tiden `tspan[0]`. Man låter ofta inparametern `dense_output` få värdet `True` för att själv bestämma vilka funktionsvärden på intervallet `tspan` man vill ha. I anropet placeras lösningen i variabeln `sol`. Precis som i fallet `ode45` beräknar `solve_ivp` en vektor med tidpunkter och en vektor, eller matris, som innehåller lösningen för de olika tidpunkterna. Anropen nedan skapar en vektor `t` och en vektor eller matris med lösningarna i de olika t-värdena:

```
t=np.linspace(tspan)
U=sol.sol(t)
```

Det finns många fler möjligheter till anrop, och man kan läsa av lösningarna från `solve_ivp` på många andra sätt. Kommandot finns dokumenterat på hemsidan `https://docs.scipy.org`.

Exempel 4. Beräkna och rita lösningar till differentialekvationen

$$\begin{cases} u' = -u(t) + \sin(t) + \cos(t),\ 0 \le t \le 4 \\ u(0) = u_0 \end{cases}$$

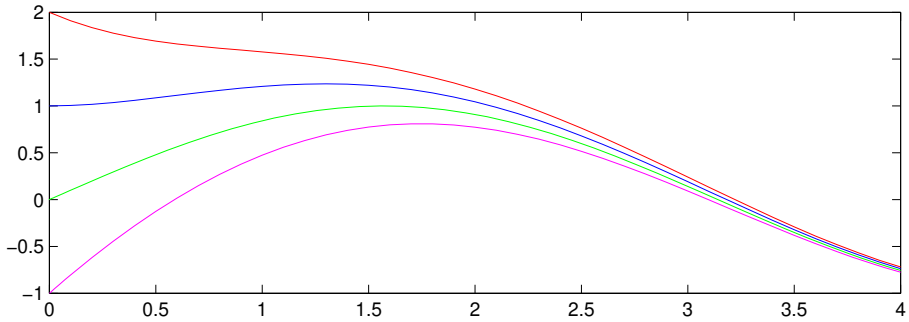
Vi beskriver högerledet i differentialekvationen beräknar och ritar lösningen för $u_0 = 1$ med

MATLAB

```
>> f=@(t,u)-u+sin(t)+cos(t);
>> tspan=[0 4]; u0=1;
>> [t,U]=ode45(f,tspan,u0);
>> plot(t,U)
```

PYTHON

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.integrate import solve_ivp
>>> def f(t,u): return -u+np.sin(t)+np.cos(t)
>>> tspan=[0,4]; u0=[1]
>>> sol=solve_ivp(f,tspan,u0,dense_output=True)
>>> t=np.linspace(0,4)
>>> U=sol.sol(t)
>>> plt.plot(t,U[0])
```



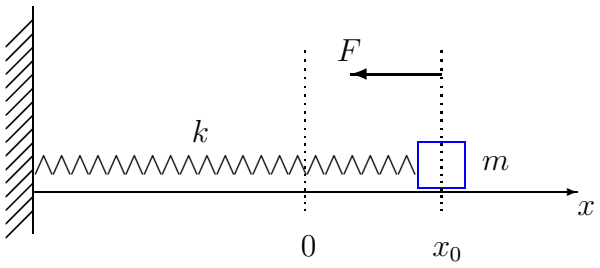
I bilden har vi även ritat ut lösningar för några andra begynnelsevärden.

Uppgift 6. Lös följande differentialekvation med begynnelsevillkor

$$\begin{cases} u' = \cos(3t) - \sin(5t)u, \ 0 \le t \le 15 \\ u(0) = 2 \end{cases}$$

Rita en graf av lösningen. Använd `solve_ivp`.

Exempel 5. Vi skall se på en harmonisk oscillator. Antag att vi har en partikel med massan m som är fastsatt i en fjäder med fjäderkonstanten k. Detta innebär att kraften då fjädern sträckts en sträcka x blir $-kx$. Fjäderns andra ända är fastsatt i en fix punkt och partikeln kan röra sig utan friktion längs en horisontell linje.



Vid tiden $t = 0$ släpps partikeln från vila, på avståndet x_0 från jämviktspunkten, och man vill beräkna den fortsatta rörelsen.

Inför ett koordinatsystem enligt figuren ovan med origo där partikeln har sitt jämviktsläge. Då x betecknar partikelns läge får vi rörelseekvationen $mx'' = -kx$ från Newtons andra lag ($F = ma$). Detta är en differentialekvation av 2:a ordningen.

Vidare har vi begynnelsevärdena $x(0) = x_0$, läget vid tiden $t = 0$, och $x'(0) = 0$, partikeln i vila vid tiden $t = 0$.

Vi har begynnelsevärdesproblemet

$$\begin{cases} x'' = -\frac{k}{m}x \\ x(0) = x_0, \quad x'(0) = 0 \end{cases}$$

Om vi låter $v = x'$, dvs. inför hastigheten, kan differentialekvationen med begynnelsevärden skrivas

$$\begin{cases} x' = v, & x(0) = x_0 \\ v' = -\frac{k}{m}x, & v(0) = 0 \end{cases}$$

dvs. som ett system av ekvationer av 1:a ordningen.

För att komma till standardform låter vi $u_1 = x$ och $u_2 = v$ och får

$$\begin{cases} u_1' = u_2, & u_1(0) = x_0 \\ u_2' = -\frac{k}{m}u_1, & u_2(0) = 0 \end{cases}$$

Med vektorbeteckningar kan detta skrivas

$$\begin{cases} \boldsymbol{u}' = \boldsymbol{f}(t, \boldsymbol{u}) \\ \boldsymbol{u}(0) = \boldsymbol{u}_0 \end{cases}, \quad \boldsymbol{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \quad \boldsymbol{f}(t, \boldsymbol{u}) = \begin{bmatrix} u_2 \\ -\frac{k}{m}u_1 \end{bmatrix}, \quad \boldsymbol{u}_0 = \begin{bmatrix} x_0 \\ 0 \end{bmatrix}$$

vilket är den form som numerisk programvara använder.

Vi beskriver differentialekvationen med funktionen

MATLAB

```
function f=harmonisk(t,u,k,m)
    f=[u(2)
       -k*u(1)/m];
```

PYTHON

```
import numpy as np
def harmonisk(t,u,k,m):
   return np.array([u[1],-k*u[0]/m])
```

Låt oss ta $m = 0.1$ kg, $k = 0.12$ N/m och $x_0 = 0.1$ m. I MATLAB beräknar vi nu lösning med `ode45` och ritar en bild som visar läget $x(t)$ med

```
>> m=0.1; k=0.12; x0=0.1; v0=0; tspan=[0 10]; u0=[x0;v0];
>> [t,U]=ode45(@(t,u)harmonisk(t,u,k,m),tspan,u0);
>> plot(t,U(:,1),'b')
>> xlabel('t'), ylabel('x(t)')
```

och det s.k. fasporträttet $(x(t), v(t))$ med

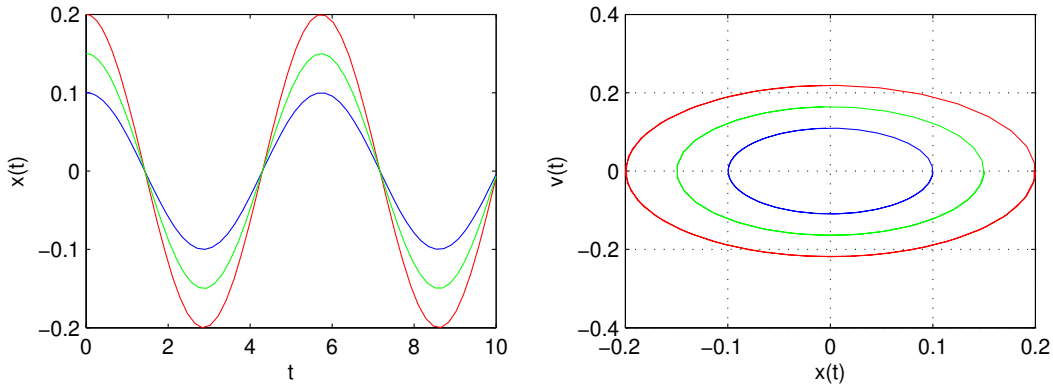
```
>> plot(U(:,1),U(:,2),'b')
>> xlabel('x(t)'), xlabel('v(t)')
```

I PYTHON kan vi skriva

```
>>> from scipy.integrate import solve_ivp
>>> import matplotlib.pyplot as plt
>>> x0=0.1; v0=0; tspan=[0,10]; u0=[x0,v0]
>>> sol=solve_ivp(harmonisk,tspan,u0,dense_output=True,args=(0.12,0.1))
>>> t=np.linspace(0,10)
>>> U=sol.sol(t)
>>> plt.plot(t,U[0],'b')
>>> plt.xlabel('t'); plt.ylabel('x(t)')
```

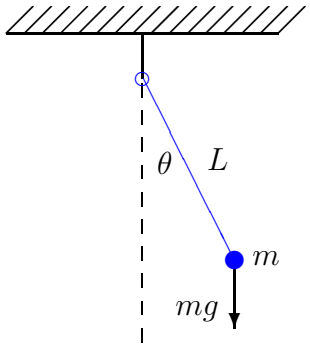
och det s.k. fasporträttet $(x(t), v(t))$ med

```
>>> plt.plot(U[0],U[1],'b')
>>> plt.label('x(t)'); plt.xlabel('v(t)')
```



I bilden har vi även ritat ut lösningar för några andra begynnelsevärden. Ekvationen för den harmoniska oscillatorn kommer vi kunna lösa analytiskt (räkna ut en formel för lösning) i läsperiod 2. Vi avslutar med att se på en uppgift med en differentialekvation som inte har någon analytisk lösning.

Uppgift 7. Den matematiska pendeln. En masspunkt med massan m hänger i en viktlös smal stav av längden L.



Med beteckningarna i figuren och Newtons andra lag får vi rörelseekvationen

$$mL\,\ddot{\theta}(t) = -mg\sin(\theta(t))$$

Vi vill bestämma lösningen för olika begynnelseutslag θ_0, dvs. $\theta(0) = \theta_0$, då vi släpper pendeln från vila, dvs. $\dot{\theta}(0) = 0$. Tag $L = 0.1$ m och begynnelseutslagen $\theta_0 = 30°, 45°, 60°$. Använd `solve_ivp`.