Forord

Oppgavene i dette heftet er hentet direkte fra læreboka til INF1100. De er valgt ut, og sortert til hver dag av hensyn til hvilket pensum som er gjennomgått i løpet av denne dagen.

Oppgavene har tre typer merkinger. Skal gjøreser oppgaver som du helst bør bli ferdig med på datalabben. De bør gjøres i den rekkefølgen de står. "Hjemmeer oppgaver som, ihvertfall til en stor del kan gjøres for hånd. Jeg anbefaler at du gjør flesteparten av disse hjemme. Ekstraer oppgaver du kan gjøre, dersom du blir ferdig med Skal gjøresoppgavene på labben. (Ikke begynn på "Hjemmeoppgavene). I prinsippet kan du velge fritt hvilke ekstraoppgaver du vil gjøre og i hvilken rekkefølge, men endel av oppgavene er fortsettelsesoppgaversom bør gjøres i den rekkefølgen de står.

Det er mulig at jeg har gitt flere Skal gjøreoppgaver, enn det det er praktisk mulig å gjøre på fire timer. Ikke forrtvil, men gjør så godt du kan. Jo flere oppgaver du gjør, jo bedre blir du til å programmere, men ditt beste er uansett godt nok:-)

Du vil trenge noen formler til oppgavene. Disse følger her:

$$y(t) = v_0 t - \frac{1}{2} g t^2 \tag{1}$$

$$F = \frac{9}{5} C + 32 \tag{2}$$

$$\begin{aligned} S(t;n) &= \frac{4}{\pi} \sum_{i=1}^{n} \frac{1}{2i-1} \sin \frac{2(2i-1)\pi t}{T} && (3) \\ &= \frac{4}{\pi} \left(\sin \frac{2\pi t}{T} + \frac{1}{3} \sin \frac{6\pi t}{T} + \frac{1}{5} \sin \frac{10\pi t}{T} + \cdots \right) && (4) \end{aligned}$$

$$y = f(x) = x \tan \theta - \frac{1}{2v_0^2} \frac{g x^2}{\cos^2 \theta} + y_0 \tag{5}$$

Når du er ferdig med å skrive programmet til hver av oppgavene gjelder følgende:

- Kjør programmet. Vanligvis vil du få noen feilmeldinger første gang du gjør dette.
- Les feilmeldingen du får og prøv å rette den selv. Se avsnittet om feilmeldinger før du ber om hjelp, men for all del, be om hjelp når du står fast. Få rettet feilen.
- Kjør programmet igjen og rett feil inntil du ikke får flere feilmeldinger

- Det du har fått nå, er kjøreresultatet. Det står i terminalvinduet ditt etter programkallet. Hvis du ikke får noe ut, skyldes det sannsynligvis at du har glemt å lage en utskrift i programmet ditt.

- Gir kjøreresultatet ditt mening? Endre gjerne småting i programmet og sjekk om det fremdeles gir riktige resultater. Dersom programmet skal gjøre en utregning. gjør den for hånd og sjekk om du og programmet får det samme. Finn eventuelt et enklere eksempel der du kan finne svaret analytisk.

- Dersom kjøreresultatet ditt er feil, må du prøve å finne ut hva programmet gjør feil. Skriv ut mellomresultater i programmet. Spør om hjelp til å debugge programmet ditt, les gjerne vedlegget om debugging i INF1100-kompendiet eller se Sindre Froyns debuggingsfilm på http://folk.uio.no/sindrf/python/

- Når du er fornøyd med kjøreresultatet: Lim inn kjøreresultatet nederst i filen. (Du limer inn resultat fra kommandovinduet ved å markere teksten i kommandovinduet med musen. Deretter flytter du musepekeren nederst i programmet ditt og klikker på venstre musetast. Klikk tilslutt på midtre musetast, du har limt inn teksten.)

- Kommenter ut kjøreresultatet ditt med trippelt anførselstegn eller firkanttegnet (#). (I endel editorer finnes det også muligheter for å kommentere ut et markert område)

- Kjør programmet igjen og sjekk at det fortsatt fungerer, dersom det ikke fungerer, sjekk at du har kommentert ut kjøreresultatet på riktig måte.

- Begynn på neste oppgave.

Skriv gjerne komemtarer i programmene dine som forklarer hva de ulike variablene er, og hva som skjer gjennom programmet ditt. Tenk deg gjerne at andre skal kunne bruke programmet, og at du selv skal kunne åpne programmet igjen om et år, og fortsatt forstå hva det gjør.

1 Dag 1

1.1 Skal gjøres: Skriv et program som regner ut 1+1

Lag et program som regner ut 1+1. La programmet skrive ut resultatet. Gjør utregningen i ipython først. Skriv så en kort fil med programmet. Opprett gjerne en variabel. Kall programmet for eksempel: `1p1.py`. ◇

1.2 Skal gjøres: Regn om fra meter til de britiske lengdeenhetene

Lag et program der du setter en lengde i meter. Deretter regner du ut og skriver ut den tilsvarende lengden i inches, feet, yards, and miles. En inch er 2.54 cm, en fot er 12 inches, en yard er 3 feet, og en britisk mile er 1760 yards. (For å verifisere at programmet fungerere kan du bruke at 640 meter tilsvarer 25196.85 inches, 2099.74 feet, 699.91 yards eller 0.3977 miles.) Kall programmet for eksempel: `lengdeomregning.py`. ◇

1.3 Skal gjøres: Regn ut luftmotstanden på en fotball

Trekkraften F_{lm} luftmotstanden yter på et objekt kan uttrykkes ved:

$$F_{lm} = \frac{1}{2} C_T \varrho A V^2, \tag{6}$$

der ϱ er tettheten til luft, V er objektets fart, A er overflatearealet normalt på hastighetsretningen, og C_T er trekkoeffisienten, som avhenger av formen på objektet og hvor glatt overflaten på objektet er.

Gravitasjonskraften på et objekt med masse m er $F_g = mg$, der $g = 9.81\mathrm{m\,s^{-2}}$.

Lag et program som regner ut trekkraften og gravitasjonskraften på et objekt. Skriv ut kreftene med en desimal i enheter av N (Newton: $\mathrm{N} = \mathrm{kg\,m/s^2}$). Definer C_T, ϱ, A, V, m, g, F_{lm}, og F_g som variabel, og legg på en kommentar der du oppgir enheten på hver av dem.

Som et eksempel kan du bruke dataene for et hardt fotballspark. Tettheten til luft er $\varrho = 1.2\ \mathrm{kg\ m^{-3}}$. For en ball er overflatearealet mot hastighetsretningen $A = \pi a^2$, der a er ballens radius. For en fotball er radien a på 11 cm. Ballens masse er 0.43 kg. C_T kan du sette til 0.2. I et hardt spark er $V = 120$ km/h. (Hint: Regn om alt til SI-enheter, V må uttrykkes i m/s etc.)

Kjør programmet igjen med $V = 10$ km/h. Sammenlikn hvor viktig gravitasjonen og farten er i forhold til hverandre i de to tilfellene. Kall programmet f.eks.: `kick.py`. ◇

1.4 Skal gjøre: Generer oddetall

Skriv et program som genererer alle oddetall fra 1 til `n`. Fiksér `n` i begynnelsen av programmet og bruk en `while`-løkke til å regne ut tallene. (Dersom `n` er et partall, skal det største partallet bli `n-1`.) Kall programmet for eksempel: `odd.py`. ◇

1.5 Skal gjøre: Lagre oddetallene i en liste

Modifiser programmet fra forrige oppgave (1.4) slik at du lagrer oddetallene i en liste. Begynn med en tom liste og legg til et nytt oddetall for hver

omgang i `while`-blokken. Skriv til slutt ut elementene i listen til skjermen. Kall programmet for eksempel: `odd_list1.py`. ◇

1.6 Skal gjøres: Bruk `range`-funksjonen for å generere oddetall

Løs oppgave 1.5 ved å bruke en `range`-funksjon. Kall programmet for eksempel: `odd_list2.py`. ◇

1.7 Skal gjøres: "Feil" i en while-løkke

Skriv denne kodebiten og kjør den

```
while True:
    print '...inside loop...'
```

Forklar hvorfor programmet oppfører seg som det gjør (Tips: For å avslutte kjøring av et program kan du trykke Ctrl c. Hold inne (`Ctrl`)-knappen og trykk så inn bokstaven `c`.) ◇

1.8 Skal gjøres, prøv å fullføre oppgaven hjemme for hånd, dersom du ikke blir ferdig: Regn ut en matematisk sum

Koden under skal regne ut summen:

$$s = \sum_{k=1}^{M} \frac{1}{k}$$

```
s = 0;  k = 1;  M = 100
while k < M:
    s += 1/k
print s
```

Programmet virker ikke som det skal. Hvilke tre feil kan du finne i programmet? (Hvis du kjører programmet, vil ingenting bli skrevet ut. Se oppgave 1.7 for dersom du lurer på hvordan du skal stoppe programmet) Kall programmet for eksempel: : `find_errors_sum.py`. ◇

1.9 Bruk en løkke i oppgave 1.8

Skriv om den rettede versjonen i oppgave 1.8 slik at det bruker en `for`-løkke over `k`-verdier istedenfor en `while`-løkke. Kall programmet for eksempel: : `compute_sum_for.py`. ◇

1.10 Hjemme: Finn feilene i følgende formeler

Nedenfor ser du noen versjoner av programmer for å regne ut likning 2. Finn ut hvilke som virker, og hvilke som ikke virker og forklar hvorfor og hva som går galt. Gjør dette først for hånd hjemme, kjør deretter programmene i ipython, og sjekk om du hadde rett, en gang du har tid (; blir omtrent det samme som ny linje)

```
C = 21;    F =  9/5*C + 32;        print F
C = 21.0;  F =  (9/5)*C + 32;      print F
C = 21.0;  F =  9*C/5 + 32;        print F
C = 21.0;  F =  9.*(C/5.0) + 32;   print F
C = 21.0;  F =  9.0*C/5.0 + 32;    print F
C = 21;    F =  9*C/5 + 32;        print F
C = 21.0;  F =  (1/5)*9*C + 32;    print F
C = 21;    F =  (1./5)*9*C + 32;   print F
```

◇

1.11 Hjemme: Simuler et program for hånd

Programmet under regner med rentebeløper:

```
initial_amount = 100
p = 5.5  # interest rate
amount = initial_amount
years = 0
while amount <= 1.5*initial_amount:
    amount = amount + p/100*amount
    years = years + 1
print years
```

De siste to deloppgavene i denne oppgaven krever bruk av datamaskinen og tilgang på python. Gjør dette en dag du har tid på datalabben eller tenk det ut på egenhånd. Dersom du har tilgang til en Windows-maskin hjemme, kan du for eksempel skrive forslag til svar i notepad eller liknede. Ellers er det viktigste at du gjør oppgave a og b.

a Forklar med ord hva slags matematisk problem som løses av dette programmet. Sammenlikn den programmerte metoden med metoden du lærte i videregåendematematikken.

b Bruk en lommeregner (eller et interaktivt python-shell) og regn deg gjennom programmet for hånd. Skriv ned verdien av `amount` og `years` ved hver omgang i while-løkken.

c Endre verdien av `p` til 5. Hvorfor går løkken nå for alltid? (Se oppgave 1.7 hvis du kjører programmet og lurer på hvordan du skal få stoppet det.) Gjør programmet mer robust, slik at feil som denne ikke så lett oppstår.

d Bruk operatoren `+=` der det er mulig.

Putt inn teksten med svarene dine på (a) og (b) i en utkommentert tekststreng nederst i programmet. Kall programmet for eksempel: : `interest_rate_loop.py`. ◇

1.12 Hjemme: Indeksering i nestede lister

Vi definerer følgende nestede liste:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

Finn listeindeksen for å få ut:

1. bokstaven `a`
2. listen `['d', 'e', 'f']`;
3. det siste elementet `h`;
4. elementet `d`.

Forklar hvorfor `q[-1][-2]` har verdien `g`. Lag et program der du skriver ut alle elementene fra deloppgavene ved å bruke indeksene du fant. Kall programmet for eksempel: : `index_nested_list.py`. ◇

1.13 Ekstra: Regn ut massen til en liter av ...

Tettheten ϱ til et stoff er definert ved $\varrho = m/V$, der m er massen til et volum V. Lag et program som regner ut massen til en liter av følgende stoffer med tetthet oppgitt i g/cm^3. Månen: 3.3; luft:0.0012; bensin: 0.67; is: 0.9; menneskekropp: 1.03; sølv: 10.5; platina = 21.4. La programmet skrive ut resultatene til slutt, bruk gjerne en formatert tekststreng Kall programmet for eksempel: `tettheter.py`. ◇

1.14 Ekstra: Definer objekter i IPython

Start `ipython` og gi følgende kommando,:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

Den vil lagre en log av økten din i `mysession.log` Definere så et heltall, et reelt talt og en tekststreng. Bruk `type`-funksjonen for å sjekke om objektene har riktig type. Skriv ut objektene. Skriv tilslutt `logoff` for å avslutte opptaket:

```
q = [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h']]
```

Gå ut av IPython og start opp igjen med `-logplay mysession.py` på kommandolinja. IPython kjører nå kommandoene i `mysession.log` på nytt, slik at alle variablene igjen er definert. Skriv ut variablene for å sjekke at dette stemmer. ◇

1.15 Ekstra: Regn ut en funksjon definert ved en sum

Den stykkevis konstante funksjonen:

$$f(t) = \begin{cases} 1, & 0 < t < T/2, \\ 0, & t = T/2, \\ -1, & T/2 < t < T \end{cases} \tag{7}$$

kan approksimeres ved summen:

$$S(t;n) = \frac{4}{\pi}\sum_{i=1}^{n}\frac{1}{2i-1}\sin\frac{2(2i-1)\pi t}{T} \tag{8}$$

$$= \frac{4}{\pi}\left(\sin\frac{2\pi t}{T} + \frac{1}{3}\sin\frac{6\pi t}{T} + \frac{1}{5}\sin\frac{10\pi t}{T} + \cdots\right) \tag{9}$$

Det kan vises at $S(t;n) \to f(t)$ når $n \to \infty$. Skriv et program som skriver ut verdien av $S(\alpha T; n)$ for $\alpha = 0.01$, $T = 2\pi$, og $n = 1, 2, 3, 4$. La `S` ($= S(t;n)$), `t`, `alpha`, og `T` være variable i programmet. En ny `S`, som svarer til en ny n skal regnes ut ved å legge et nytt ledd til den forrige verdien av `S`, d.v.s ved et utsagn som `S = S + term`. Kjør programmet for $\alpha = 1/4$ også. Er approksimasjonen for $S(\alpha T; 4)$ bedre for $\alpha = 1/4$ enn for$\alpha = 0.01$? Kall programmet for eksempel: `compare_func_sum.py`.

Bemerkning Dette programmet er tidkrevende å skrive for store n, men med en løkke kan vi bare kode det generiske leddet parametrisert ved i i i 8. Dette gir et kort program som kan finne $S(t;n)$ for alle n. ◇

1.16 Ekstra: Lag en liste og transverser den

Sett variablen `primtall` til å være en liste med tallene 1, 3, 5, 7, 11, 13. skriv ut hvert listeelement i en `for`-løkke. Kall programmet for eksempel: : `primes.py`. ◇

1.17 Ekstra: Lag en tabell av verdiene fra formel 1

Lag et program som skriver ut en tabell av t- og $y(t)$-verdier fra formel 1 til skjermen. Bruk 11 jevnt fordelte t-verdier i intervallet $[0, \frac{2v_0}{g}]$, og fiksér verdien på v_0. Kall programmet for eksempel: : `ball_table1.py`. ◇

1.18 Ekstra: Lagre verdiene fra 1 i lister

Lag et progam der du oppretter en liste `t` med 6 t-verdier $0.1, 0.2, \ldots, 0.6$. Regn ut den tilhørende listen `y` med $y(t)$-verdier som du finner fra formel 1. Skriv ut en pent formatert tabell med t- og y-verdier. Kall programmet for eksempel: `ball_table2.py`. ◇

1.19 Ekstra: Utforsk problemer med galt innrykk

Skriv inn følgende program, og vær nøye med at du har samme antall mellomrom først på hver linje.

```
C = -60; dC = 2
while C <= 60:
    F = (9.0/5)*C + 32
      print C, F
C = C + dC
```

Kjør programmet. Hva er det første problemet? Rett feilen. Hva er det neste problemet? Hva kan du gjøre for å rette på det? (Se oppgave 1.7 hvis du lurer på hvordan du stopper programmet)

Moralen er at man skal være nøye med innrykket når man skriver Python-programmer. Mange andre programmeringsspråk, som java og c lukker blokker inne i krøllete paraneteser. Dette gjelder både løkker, `if`-tester og funksjonsblokekr som vi skal lære om senere. Andre språk, som for eksempel tekstbehandlingsspråket latex har start- og slutt-merker, BEGIN-END-merker. I python er det bare innrykket som bestemmer hva som er inni og utenfor en blokk. ◇

1.20 Ekstra: Regn ut arealet av en trekant

En vilkårlig trekan kan beskrives ved koordinatene til de tre hjørnene: (x_1, y_1), (x_2, y_2), (x_3, y_3). Arealet A av trekanten er gitt ved:

$$A = \frac{1}{2}\left[x_2y_3 - x_3y_2 - x_1y_3 + x_3y_1 + x_1y_2 - x_2y_1\right] \tag{10}$$

Skriv en funksjon `areal(hjorner)` som returnerer arealet av en trekant med hjørnene spesifisert av argumentet `hjorner`.`hjorner` skal være en nestet liste av hjørnekoordinater. For eksempel kan `hjorner` være `[[0,0], [1,0], [0,2]]` hvis de tre hjørnene i trekanten har koordinatene $(0, 0)$, $(1, 0)$, and $(0, 2)$. Test `areal`-funksjonen på en trekant du kjenner arealet av. Kall programmet for eksempel: : `area_triangle.py`. ◇

1.21 Ekstra: En dobbel for-løkke over en nestet liste

Vi ser på listen i oppgave 1.7. Vi kan gjennomløpe alle elementene i `q` ved å bruke en nestet `for`-løkke:

```
for i in q:
    for j in range(len(i)):
        print i[j]
```

Hva slags objekttyper er `i` og `j`? Kall programmet for eksempel: : `nested_list_iter.py`. ◇

2 Dag 2

2.1 Skal gjøres: Regn verdien av en Gaussisk klokkefunksjon

Den Gaussiske klokkefunksjonen:

$$f(x) = \frac{1}{\sqrt{2\pi}s} \exp\left[-\frac{1}{2}\left(\frac{x-m}{s}\right)^2\right], \tag{11}$$

er mye brukt i teknologi og (natur-)vitenskap. Parametrene m og s er reelle tall, der s må være større enn 0. Lag et program som finner verdien av funksjonen for $m = 0$, $s = 2$, og $x = 1$. Sjekk at resultatet stemmer ved å taste det inn på kalkulatoren Skriv om programmet slik at du bruker en funksjon som tar inn m, x and s. I hoveddelen av programmet tar du inn m, x og s som verdier fra bruker. Velg selv om du vil bruke sys.argv eller raw_input. Kall programmet for eksempel: : `bell_function.py`. ◇

2.2 Skal gjøres: Skriv en funksjon som gjør om Farenheit til Celsius

Formelen for å konvertere Fahrenheit til Celsius er:

$$C = \frac{5}{9}(F - 32) \tag{12}$$

Skriv en funksjon `C(F)` som implementerer denne formelen. Skriv også en funksjon `F(C)` som konverterer celsius tilbake til Farenheit. Formelen for dette blir:

$$F = \frac{9}{5}C + 32 \tag{13}$$

Sjekk implementasjonen din ved å konvertere en temperatur i Celsius til Fahrenheit og så tilbake. Se også opp for heltallsdivisjoner. Du sjekker altså om du har fått den samme temperaturen tilbake som du startet med. Bruk gjerne en if-test. Kall programmet for eksempel: : `c2f2c.py`. ◇

2.3 Skal gjøres: Skriv programmet i oppgave 1.8 som en funksjon

Definer en Python-funksjon `s(M)` som regner ut summen s som definert i opppgave 1.8. Kall programmet for eksempel: `compute_sum_func.py`. ◇

2.4 Program

```
g = 9.81;  v0 = 5
dt = 0.25

def y(t):
    return v0*t - 0.5*g*t**2

def table():
    data = []  # store [t, y] pairs in a nested list
    t = 0
    while y(t) >= 0:
        data.append([t, y(t)])
        t += dt
    return data

data = table()

for t, y in data:
    print '%5.1f  %5.1f' % (t, y)

# extract all y values from data:
y = [y for t, y in data]
print y
# find maximum y value:
ymax = 0
for yi in y:
    if yi > ymax:
        ymax = yi
print 'max y(t) =', ymax

data = table()  # this does not work now - why?
```

2.5 Skal gjøres: Bruk brukerinput fra kommandolinje i programmet 2.4 over

Modifiser programmet 2.4) over, slik at input-dataene initialiseres fra kommandolinjeargumenter. Utstyr programmet med en (`try-except`-forgreining som tar seg av eventuelle feil. Kall programmet for eksempel: : `ball_summary_cml_exception.py`. ◇

2.6 Skal gjøres: Spør brukeren om input til programmet 2.4

Modifiser programmet 2.4 slik at brukeren blir spurt om input-dataene, og skriver dem inn på tastaturet. Utstyr programmet med feilhåndtering (exeptions). Kall programmet for eksempel: : `ball_summary_qa_exception.py`. ◇

2.7 Hjemme: Hvorfor man skal teste for spesifikke feiltyper

Den enkleste måten å skrive en `try-except`-blokk på er å test for en hvilken som helst feil/exception. For eksempel:

```
try:
    C = float(sys.arg[1])
except:
    print 'C must be provided as command line argument'
    sys.exit(1)
```

Tenk over, hva er problemet med dette programmet?

At brukeren kan glemme å oppgi kommandolinjeargument, er den opprinnelige bakgrunnen for `try`-blokken over. Finn ut hva slags exception som er passende for denne feilhåndteringen, og skriv om programmet slik at den tester for denne spesifikke feilen. Hva er problemet med denne versjonen av programmet? Rett opp den nye feilen. Kall programmet for eksempel: : `cml_exception.py`. ◇

2.8 Ekstra: Skriv en funksjon for numerisk derivasjon

Formelen:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{14}$$

kan brukes til å finne en omtrentelig verdi for den deriverte til en matematisk funksjon $f(x)$ hvis h er liten. Skriv en funksjon `diff(f, x, h=1E-6)` som returnerer en approksimasjon til den deriverte til en matematisk funksjon, representert ved en annen Python-funksjon `f(x)`.

Sammenlikn den approksimasjonen for den deriverte med en nøyaktige verdien, for følgende matematiske funksjoner: $f(x) = e^x$ ved $x = 0$, $f(x) = e^{-2x^2}$ ved $x = 0$, $\cos x$ ved $x = 2\pi$, og $f(x) = \ln x$ ved $x = 1$. Bruk $h = 0.01$. Kall programmet for eksempel: : `diff_f.py`.

Kommentar: `h=1E-6` i `diff(f, x, h=1E-6)` betyr at vi ikke trenger å oppgi h når vi kaller på funksjone. Dersom vi ikke sender med h, vil den per default settes til 1E-6. Vi kan oppgi default-verdier på vilkårlig mange av argumentene til en funksjon, men disse må alltid være de siste av argumentene. Argumentene som ikke har default-verdi må komme først. På samme måte kan vi endre rekkefølgen på variablene når vi sender dem med. For eksempel kan vi skrive diff(f,h=1E-8,x=4), men dersom vi har variable som ikke sendes med på denne måten, må, disse være de første i tuppelet. ◇

2.9 Ekstra: Sjekk om matematikkens lover gjelder på datamaskinen

På grunn av avrundingsfeil, kan det hende at en matematisk regel som $(ab)^q = a^q b^q$ ikke holder eksakt på datamaskinen. I denne oppgaven skal

du sjekke reglene for en stor mengde tilfeldige tall. Vi kan lage tilfeldige talll ved å bruke `random`-modulen som er standard i Python:

```
import random
a = random.uniform(A, B)
b = random.uniform(A, B)
```

Her vil `a` og `b` være tilfeldige tall, som alltid er større enn `A` og mindre enn `B`.

Lag et program som leser inn antall tester som skal gjøres på kommandolinja. Bestem verdiene for `A` og `B` i programmet, for eksempel -100 og 100. Utfør testene i en løkke; Inne i løkkka trekker du de tilfeldige tallene `a` og `b` og utfører tester for å se om de matematiske utrykkene er ekvivalente. Tell opp antall ganger reglene slår feil, og skriv ut hvor mange prosent av gangene det gikk feil i slutten av programmet.

Reglene du skal teste ut er de følgende:

1. $a - b$ og $-(b - a)$
2. a/b og $1/(b/a)$
3. $(ab)^2$ og b^2a^2

Kall programmet for eksempel: : `math_rules_failures.py`. ◇

2.10 Ekstra: Implementer en matematisk funksjon

Lag en Python-funksjon `g(t, a)` som regner ut

$$g(t) = e^{-at^2}$$

og $g'(t) = -2atg(t)$. Returner funksjonsverdiene for $g(t)$ og $g'(t)$. Bruk funksjonen til å skrive ut et resultat på følgende format:

```
g(1, a=0.5)=0.606531, g'(1, a=0.5)=-0.606531
```

Kall programmet for eksempel: : `exp_func.py`. ◇

2.11 Ekstra: Implementer fakultetsfunksjonen

n fakultet (eng. factorial), skrevet som $n!$, er definert ved

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1, \tag{15}$$

med spesialtilfellene:

$$1! = 1, \quad 0! = 1 \tag{16}$$

For eksempel er $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$, og $2! = 2 \cdot 1 = 2$. Skriv en funksjon `fact(n)` som returnerer $n!$. Returner 1 med en gang dersom n

er 1 eller 0, ellers lager du en løkke for å regne ut $n!$. Kall programmet for eksempel: `fact.py`.

Remark. Du kan importer en ferdiglaget fakultetsfunksjon ved å skrive:

```
from scitools.all import factorial
```

Denne `factorial`-funksjonen har mange ulike implementasjoner med ulik effektivitet, for å regne ut $x!$ (se kildekoden for detaljer). ◇

2.12 Ekstra dersom du har gjort oppgave 1.15: Implementer oppgave 1.15 med en for-løkke

Lag en funksjon som evaluerer $S(t; n)$ som definert i oppgave 1.15 på side 7, bruk en for-løkke til å summere de n leddene. Funksjonen skal ta inn verdiene t, T, og n som argumenter, og returnere $S(t; n)$ og feilen i approksiamsjonen av $f(t)$. Lag et main-program, altså hoveddelen av programmet som kjøres når vi kaller på programmet, der du setter $T = 2$ og skriver ut en tabell med de tilnærmede feilen for $n = 1, 5, 20, 50, 100, 200, 500, 1000$ og $t = 1.01, 1.1, 1.8$. Bruk en rad for hver n-verdi og en kolonne for hver t-verdi. Kall programmet for eksempel: : `compute_sum_S.py`. ◇

2.13 Ekstra: Skriv en Farenheit-Celsius-konverteringstabell

Gitt en temperatur F i Fahrenheit , finner man den tilsvarende verdien i Celsius ved å løse2 (på side 1) med hensyn på C. Dette gir formelen 13. Mange bruker en tilnærmet formel for å regne ut verdien i Celsius: trekk fra 30 på Fahrenheit-verdien, og del resultatet med to, d.v.s:

$$C = (F - 30)/2 \tag{17}$$

Vi vil lage en tabell som sammenlikner de to formelene 13 og 17 for Fahrenheit-grader mellom 0 og 100 (i skritt på for eksempel 10 grader). Skriv et program som lager og skriver ut denne tabellen. Kall programmet for eksempel: : `f2c_shortcut_table.py`. ◇

2.14 Ekstra: Les kommandolinjeinput for 1

Se på det enkle programmet for å evaluere formelen: 1:

```
v0 = 3; g = 9.81; t = 0.6
y = v0*t - 0.5*m*g*t**2
print y
```

Modifiser programmet slik at `t` og `v0` taes inn fra kommadolinjen. Kall programmet for eksempel: : `ball_cml.py`. ◇

2.15 Ekstra: Be brukeren om input til formelen 1

Modifiser programmet i oppgave 2.14, slik at programmet stiller brukeren spørsmålene `t=?` og `v0=?`, og får `t` og `v0` fra brukerens input. Kall programmet for eksempel: : `ball_qa.py`. ◇

2.16 Ekstra: Gjør programmet i oppgave 2.15 idiotsikkert

Test om `t`-verdien som leses inn i programmet i oppgave 2.15 ligger melllom 0 og $\frac{2v_0}{g}$. Hvis ikke, skriv en beskjed og avbryt kjøringen av programmet. Kall programmet for eksempel: : `ball_qa_errorcheck.py`. ◇

3 Dag 3

session:

```
>>> def f(x):
...     return x**3       # sample function
...
>>> n = 5                 # no of points along the x axis
>>> dx = 1.0/(n-1)        # spacing between x points
>>> xlist = [i*dx for i in range(n)]
>>> ylist = [f(x) for x in xlist]
>>> pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

3.1 Skal gjøres: Fyll lister med funksjonsverdier

En funksjon er definert som:

$$h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}, \quad -4 \leq x \leq 4 \tag{18}$$

Tilpass eksempelet øverst i avsnitt 3 til å representere funksjonen h og fylle listene `xlist`, `hlist`, og `pairs` med $h(x)$ funksjonsdata. Skriv ut listene for å teste resultatet Kall programmet for eksempel: : `fill_lists.py`. ◇

3.2 Skal gjøres: Fyll arrayer med funksjonsverdier, løkkeversjon

Løs oppgave 3.1, men lagre verdiene for x og $h(x)$ direkte i arrayer isteden for lister, ved hjelp av en løkke over arrayelementer. Kall programmet for eksempel: : `fill_arrays_loop.py`. ◇

3.3 Skal gjøres: Fyll arrayer med funksjonsverdier, vektorisert versjon

Løs oppgave 3.1, men lagre verdiene for x og $h(x)$ i arrayene `x` og `h` istedenfor lister. Bruk, til forskjell fra i oppgaven over, vektor- eller array-aritmetikk 3.2, ikke løkker. D.v.s finn `x`-arrayen med `linspace`-funksjonen og bruk vektoraritmetikk på denne for å finnne `h`. Kall programmet for eksempel: : `fill_arrays_vectorized.py`. ◇

3.4 Skal gjøres: Plott en bølgepakke

Funksjonen

$$f(x,t) = e^{-(x-3t)^2} \sin\left(3\pi(x-t)\right) \tag{19}$$

beskriver en bølge lokalisert i rommet for en bestemt verdi av t. Lag et program som visualiserer denne funksjonen som en funksjon av x i intervallet $[-4,4]$ når $t = 0$. Kall programmet for eksempel: : `plot_wavepacket.py`. ◇

3.5 Skal gjøres: Animer en bølgepakke

Vis en animasjon av funksjonen $f(x,t)$ i oppgave 3.4 ved å plotte f som en funksjon av x på $[-6,6]$ for en mengde av t i intervallet $[-1,1]$. Lag også en animert gif-fil. En passende oppløsning kan være 1000 intervaller (1001 punkter) langs x-aksen, 60 intervalller (61 punkter) i tid og 6 bilder per sekund (eng. frames per sekund, fps) i den animerte GIF-filen.

Filmen vil vise at $f(x,t)$ modellerer bølger som beveger seg mot høyre (der x er romkoordinaten og t er tiden). Farten på de individuelle bølgene og bølgepakken er forskjellig. Dette viser forskjellen på fasehastighet og gruppehastighet i fysikk. Kall programmet for eksempel: `plot_wavepacket_movie.py`. ◇

3.6 Hjemme: Simuler et vektorisert uttrykk for hånd

Anta at `x` og `t` er to arrayer av samme lengde, og at disse inngår i det vektoriserte uttrykket:

```
y = cos(sin(x)) + exp(1/t)
```

Anta at `x` inneholder de to elementene 0 og 2, og at `t` inneholder elementene 1 og 1.5. Regn ut hva `y`-arrayet blir for hånd (med hjelp av lommeregneren din). Skriv så et program som etteraper utregningene du gjor-

de for hånd. Bruk løkkker eksplisitt. Sjekk resultatene datamaskinen gir. Kall programmet for eksempel: `simulate_vector_computing.py`. ◇

3.7 Ekstra: Plott banen til en ball

Formelen for banen til en ball er gitt ved formel 5 på side 1. Lag et program der du først leser inn input-dataene y_0, θ, og v_0 fra kommandolinja. Regn så ut hvor ballen treffer bakken d.v.s, verdien av x_g der $f(x_g) = 0$. Plott banen $y = f(x)$ for $x \in [0, x_g]$, med samme skala på x- og y-aksen slik at vi får et visuelt korrekt inntrykk av banen. Kall programmet for eksempel: : `plot\_trajectory.py`. ◇

3.8 Ekstra: Bedøm et plott

Se på følgende program for å plotte en parabel:

```
x = linspace(0, 2, 20)
y = x*(2 - x)
plot(x, y)
```

Så bytter du til funksjonen $\cos(18\pi x)$ ved å endre utregningen av `y` til `y = cos(18*pi*x)`. Bedøm plottet du får ut. Er det riktig? Hvis funksjonen $\cos(18\pi x)$ med 1000 points i samme plott Kall programmet for eksempel: : `judge_plot.py`. ◇

3.9 Ekstra: Sammenlikn Taylor-serie-tilnærminger til $\sin x$

Sinusfunksjonen kan tilnærmes ved et polynom på den følgende formen:

$$\sin x \approx S(x; n) = \sum_{j=0}^{n} (-1)^j \frac{x^{2j+1}}{(2j+1)!} \tag{20}$$

Uttrykket $(2j + 1)!$ er fakultetsoperatoren (se oppgave 2.11. Dersom du ikke har gjort denne oppgaven allerede, kan du gjøre den nå, eller lese bemerkningen om hvordan du kan importere en fakultetsfunksjon). Feilen i tilnærmingen $S(x; n)$ minker når n øker og i grensen har vi $\sin x = \lim_{n\to\infty} S(x; n)$. Meningen med denne opppgaven er å visualisere ulike tilnærminger $S(x; n)$ når n øker.

Den første delen av oppgaven er å skrive en Python-funksjon `S(x, n)` som beregner $S(x; n)$. Bruk en enkel tilnærming, der du regner ut hvert ledd som det står i formelen, d.v.s. $(-1)^j x^{2j+1}$ delt med fakultetet $(2j + 1)!$.

Neste del av oppgaven består i å plotte $\sin x$ på intervallet $[0, 4\pi]$ sammen med aproksimasjonene $S(x; 1)$, $S(x; 2)$, $S(x; 3)$, $S(x; 6)$, og $S(x; 12)$. Kall programmet for eksempel: : `plot_Taylor_sin.py`. ◇

3.10 Ekstra: Lag en film av utviklingen av Taylor-tilnærmingene

En generell rekkutvikling av en funksjon kan skrives som $S(x) = \sum_{k=M}^{N} f_k(x)$. For eksempel er Taylorutviklingenstilnærmingen av e^x lik $S(x)$ med $f_k(x) = x^k/k!$. I denne oppgaven skal vi lage en film som viser hvordan rekkeutviklingenstilnærmingen utvikler seg når $k = M, M+1, \ldots, N$.

Lag en funksjon `animate_series` for å lage slike animasjoner. De første to argumentene er $f_k(x)$ og M, der $f_k(x)$ må være en Python-funksjon med argumentene x og k. De neste argumentene er antallet ledd i serien, miniums- og maksiumsverdiene for x i plottet, antall x-punkter i plottet, og minimums- og maksimumsverdiene for y values i plottet. Det siste argumentet skal være valgfritt (se oppgave 2.8 dersom du lurer på hva dette innebærer) og bestemme funksjonen $S(x)$ approksimerer. Hvis argumentet er oppgitt skal hvert bilde i filmen være et plott med både $S(x)$ og funksjonen den approksimerer. Lagre plottene i filer, og lag en hardcopy av filmen.

Test funksjonen på de to tilfellene:

1. Taylorrekka for $\sin x$-funksjonen, der $f_k(x) = (-1)^k x^{2k+1}/(2k+1)!$, og $x \in [0, 13\pi]$, $N = 40$, $y \in [-2, 2]$.
2. Taylorrekka for e^{-x}, der $f_k(x) = (-x)^k/k!$, og $x \in [0, 15]$, $N = 30$, $y \in [0, 1]$.

Kall programmet for eksempel: `animte_Taylor_series.py`. ◇

4 Dag 4 og 5, differensiallikninger

4.1 Skal gjøres: Løs en ODE for radioaktivt henfall

Likningen

$$f(u, t) = -au, \tag{21}$$

er en relevant modell på radioaktivt henfall. Den ukjente $u(t)$ representerer mengden av et radioaktivt stoff, og a er relatert til hvor fort det radioaktive stoffet henfaller. Mengden $u(t)$ er da bestemt ved differensiallikningen:

$$u'(t) = f(u, t) = -au, \tag{22}$$

I denne oppgaven skal vi skrive et program som løser denne differensiallikningen, og plotter løsningen. Vi skal løse differensiallikningen både med Eulers metode, og Runge-Kutta til fjerde orden. Vi vil plotte disse og den eksakte løsningen, og vurdere hvilken metode som er best.

a Lag en funksjon `f(u,t)` som representerer funksjonen i differensiallikningen. I hoveddelen av koden setter du $a = 0.01$ year^{-1},

og initialverdien $u(0) = 1$. Test funksjonen (kommenter ut funksjonstesten din når du har sett at funksjonen fungerer).

b Vi vil måle tiden i år og bruk et tidssteg på 100 år for både Eulers metode og fjerdeordens Runge Kutta. Begynn med å la tiden gå fra 0 til 1000 år. Opprett en verdi for steglengden `h`, og en array `tarray` med riktig lengde ved hjelp av zeros. Start med $t = 0$, og lag en `while`-løkke over skrittnummeret i, der du legger inn t i arrayen og øker t med h, til du har lagt inn alle verdiene. Skriv ut tarrayet til slutt for å sjekke om det har blitt riktig. Denne testen kan du fjerne senere. `while`-løkken skal du senere bruke til å løse differensiallikningen. Hver omgang i `while`-løkken brukes da til å finne det neste verdien for $u(t_{i+1})$.

c Lag en funksjon `EulerNeste(f,h,u,t)` som regner ut den neste verdien for u, fra den forrige, ved hjelp av Eulers metode. Metoden skal altså returnere den neste u-verdien. Lag en array for u-verdiene, med zeros. Sett den første verdien til u(0) = 1, og regn ut resten av verdiene en for en i `while`-løkken ved å kalle på EulerNeste-funksjonen.

d Plot verdiene av uarrayet mot tarrayet. Løs differensiallikningen analytisk og en array med funksjonsverdier som tilsvarer tarraet ved hjelp av vektoraritmetikk. Plott den analytiske løsningen sammen med den du fant fra løsning med Eulers metode. Sett på passende markeringer på aksene, en tittel, og bruk `legend` til å markere hvilken funksjon som er hvilken. Finn ut om du har valgt et passende tidsintervall, eller om du bør øke eller minke det.

e Lag en funksjon `RK4Neste(f,h,u,t)` som implementerer Runge-Kuttas-metode til fjerde orden. Lag et nytt uarray der du legger inn løsningen i `while`-løkken, på samme måte som for Eulers metode. Plott også denne løsningen i det samme plottet, og legg på en egen `legend`. (Du legger nye funksjoner i samme plott ved å skriv hold('on') mellom hver plottekommando).

f Studer plottet du har fått. Hvilken metode gir det beste resultatet? Hvorfor tror du det er slik? Tenk over; har medaljen en bakside, tror du at det kan være du taper noe på å bruke den beste metoden (hint, hvilken metode er enklest å forstå, hvilken fører til færrest beregninger? I datamaskinens ungdom var antall regneoperasjoner (eng. floating point operations, flops) mye viktigere enn det er nå. Da kunne du tydelig merke at en av metodene brukte flere operasjoner enn den andre. Selv nå kan det noen ganger lønne seg å bruke den mest effektive (det vil si raskeste, med

færrest operasjoner) for å få et inntrykk, før vi gjør en tung beregning med den metoden som gir det beste resultatet. Det kan være en god måte å teste et program på, uten å måtte vente for lenge. Det kan også være lurt å kjøre et mindre testsett før vi prøver programmet på de virkelige dataene. Hvis det tar to dager å kjøre programmet, er det kjedelig om vi må kaste resultatene på grunn av en feil i koden. Forøvrig gir RK4 også bedre resultater en Euler når vi bruker så få skritt i denne at antall flops blir det samme som i Euler. Den eneste fordelen med Eulers metode er da at den er lettere å forstå. Den brukes derfor mest i pedagogisk øyemed.)

g Endre programmet slik at du kan oppgi skrittlengde på kommandolinje, la den være som tidligere dersom ingen skrittlengde er gitt. Endre skrittlengde både til kortere og lengre skritt i det samme intervallet. Er vurderingen din av hvilken metode som er best, den samme som før?

Slike tester for metoders følsomhet ovenfor endringer av parametre, er viktige når vi vurderer hvor god en metode er. Selvom vi i dette tilfellet kommer til den samme konklusjonen, er det ikke alltid slik. Det er viktig å teste om en metode er den beste også når vi endrer forutsetningene den kjøres under. Kall programmet for eksempel: `radioactive\_decay.py`. ◇

4.2 Skal leses: Hva nå?

Etter at du har gjort Skal gjøresoppgaven for dagene fire og fem kan du velge hva du vil gjøre videre. Du kan gjøre repetisjonsoppgavene under 4.5, eller du kan jobbe med virrevandiring og tilfeldige tall under 4.6. Dersom du vil jobbe mer med differensiallikninger, kan du gjøre resten av oppgavene i dette avsnittet.

4.3 Ekstra: Implementer en annenordens Runge-Kutta-metode

Implementer Runge-Kutta-metoden til annen orden gitt ved den generiske algoritmen:

$$k_1 = hf(t_i, y_i), \tag{23}$$

$$k_2 = hf(t_{i+1/2}, y_i + k_1/2), \tag{24}$$

Og verdien blir tilslutt

$$y_{i+i} \approx y_i + k_2 + O(h^3). \tag{25}$$

på samme måte som funksjonen du konstuerte for Euler og Runge-Kutta til fjerde orden i oppgave 4.1 Konstruer et enkelt testproblem der du kjenner den analytiske løsningen, og plott forskjellen mellom den numeriske og den analytiske løsningen og/eller plott den numeriske og den analytiske løsningen i ett og samme plott. Kall programmet for eksempel: : `RungeKutta2_func.py`. ◇

4.4 Implementer midtpunktsmetoden

I denne oppgaven skal du gjøre det samme som i oppgave 4.3, men denne gangen skal du gjøre det for å implementere midtpunktsmetoden gitt ved den generiske algoritmen:

$$y_{k+1} = y_{k-1} + 2hf(y_k, t_k), \tag{26}$$

Kall programmet for eksempel: `Midpoint.py`. ◇

4.5 Ekstra: Repetisjon

Under er noen punkter for hva du kan gjøre når du er ferdig med oppgave 4.1. Det er ment å være en grei måte å repetere det du har gjort denne uken på. Oppgave (b) kan du også godt gjøre hjemme på et senere tidspunkt.

a Gjør noen av ekstraoppgavene du ikke har gjort så langt.

b I dette kurset har vi i komprimert form gjennomgått pensum fra kapitlene 1, 2, 3, og 4 fra forelesningsheftet i INF1100. På nettsiden til INF1100-kurset

http://www.uio.no/studier/emner/matnat/ifi/INF1100/h07/

ligger det en quiz til hvert kapittel. Ta kapittel-quizene for kapitlene 1-4. Klarer du å svare riktig på alt?

c Gjør flere oppgaver fra kapitlene 1-4 i INF1100-kompendiet

d Gjør oppgavene merket "Optional"i kompendiet til MAT-INF1100. Dette er programmeringsoppgaver, som du nå skal være i stand til å gjøre

4.6 Ekstra: Bonusstoff - tilfeldige tall

Les bonusstoffet om tilfeldige tall og gjør følgende oppgaver. Programmene det refereres til i oppgavene er skrevet for INF1100, og kan lastes ned fra følgende lenke http://vefur.simula.no/intro-programming/src/random/. Mesteparten av dem er nesten identiske med eksempelet vi så i avsnittet om random walks, med unntak av mine oversettelser av variabelnavn.

4.7 Regn ut en sannsynlighet

Hva er sannsynligheten for å få et tall mellom 0.5 og 0.6 når du trekker uniformt distribuerte, tilfeldige tall i intervallet $[0, 1)$? La oss finne svaret på spørsmålet ved hjelp av empiri: Lag et program som trekker N slike tilfeldige tall ved hjelp av Pythons standard `random`-modul. Tell hvor mange, M, av disse tallene som ligger i intervallet $(0.5, 0.6)$, og regn ut sannsynligheten som M/N. Kjør programmet med verdiene $N = 10^i$ for $i = 1, 2, 3, 6$. Kall programmet for eksempel: `compute_prob.py`. ◇

4.8 1D-virrevandring

Modifiser programmet `walk1D.py`, slik at sannsynligheten for forflytning mot høyre er r og sannsynligheten for å gå mot venstre er $1 - r$ (trekk reelle tall i $[0, 1)$ heller enn i $[1, 2]$). Regn ut den gjennomsnittelige posisjonen til n_p partikler etter 100 skritt, der n_p leses fra kommandolinjen.

Man kan vise matematisk at den gjennomsnittelige posisjonen går mot $rn_s - (1 - r)n_s$ når $n_p \to \infty$. Skriv ut dette teoretiske resultatet sammen med den gjennomsnittelige posisjonen du fant fra simuleringen med et endelig antall partikler. Kall programmet for eksempel: `walk1D_drift.py`. ◇

4.9 1D-virrevandring til et gitt punkt

Sett `np=1` i programmet `walk1Dv.py` og modifiser det slik at det måler hvor mange skritt som må tas før en partikkel når et punkt $x = x_p$. Oppgi x_p på kommandolinjen. Test programmet for $x_p = 5, 50, 5000, 50000$. Kall programmet for eksempel: `walk1Dv_hit_point.py`. ◇

4.10 2D-virrevandring

Modifiser programmet `walk1D.py`, slik at det modellerer virrevandring i to dimensjoner på følgende måte. La det ved vært skritt være like sannsynlig at partikkelne beveger seg mot høyre, vestre, opp eller ned. Bruk fortsatt 4 partikkler og 100 skritt.

Plott posisjonene til partiklene for hver omgang i løkka slik at det blir en film, (som beskrevet i forelesningsheftet). Finn også gjennomsnittetsposisjonen og standardavviket i begge retninger hver gang. Kan du legge inn gjennomsnittsposisjonen som en prikk i plottet? Kan du lage linjer som representerer maksimalt standardavvik i hver retning. (fire linjer i plottet)? Kall programmet for eksempel: `walk2D.py`. ◇