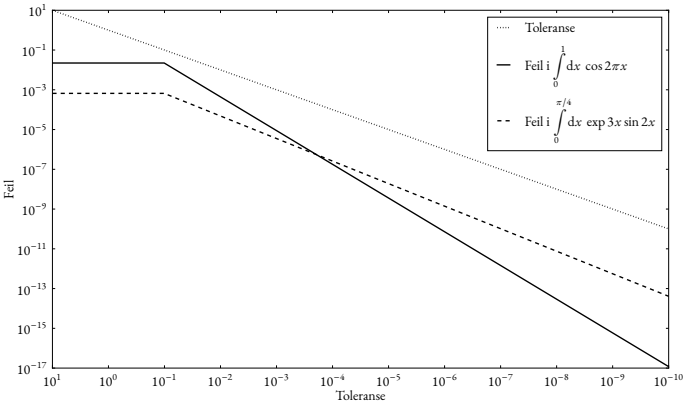
STEFFEN HAUG

Numeriske Metodar

Prosjekt 3

I. Kvadraturmetodar

Vi skal implementere to forskjellige kvadraturmetodar: Ein adaptiv kvadraturmetode som tilpassar steglengda slik at ein er meir nøye med tilnærmingar over intervall som er utsatt for større feil, og Rombergs metode som burkar ekstrapoleringsteknikkar for å konvergere raskare.



Figur 1: *Feil i adaptiv Simpson-kvadratur. Ein ser at feilen er grensa ovanfrå av toleransen.*

i. Adaptiv Simpson-kvadratur

Adaptive kvadraturmetodar baserer seg på å integrere over eit intervall med ein enkel kvadraturmetode, estimere feilen, for so å rekursivt dele intervallet i to og beregne eit meir nøyaktig estimat med halvparten av stelengda dersom feilen er uakseptabelt stor.

Algoritma (**appendix A**), med Simpsons metode over kvart del-intervall, vart nytta til å estimere integrala i **tabell 1**. Ei oversikt over samanhengen mellom feil og toleranse ser ein i **figur 1**.

Tabell 1: *Estimasjonar v. h. a. adaptiv Simpson-kvadratur med toleranse* $\varepsilon = 10^{-10}$.

Integral	Estimat
$\int_{x=0}^{x=1} \mathrm{d}x \cos 2\pi x$	$-2.7138785046 \times 10^{-17}$
$\int_{x=0}^{x=\pi/4} \mathrm{d}x \exp 3x \sin 2x$	2.58862863250

ii. Romberg-kvadratur

Romberg-algoritma (**appendix B**) organiserer (i prinsippet) dei utrekna verdiane i ei nedre-triangulær matrise, der kvar *kolonne* er estimat av integralet med aukande presisjon (kortare steglengde) nedover langs kolonna. Trikset til Romberg er at ein – ved hjelp av to estimat av ein gitt orden, med forskjellige steglengder h og h/t – kan ein konstruere eit estimat av *høgare orden*, ved hjelp av *Richardson-ekstrapolering*. Ekstrapoleringa *aksellererer* konvergensen til følgja, som ein ser i **figur 2**.

La $A(h)$ vere ein approksimasjon på ekvidistante nodar med steglengde h

$$A(h) \approx I := \int_a^b \mathrm{d}x\, f(x)$$

slik at $A(h) - I = \mathcal{O}(h^n)$, der restfunksjonen er eit polynom i h. I tilfelle for Rombergs metode er denne gitt ved Euler-Maclaurin-ekspansjonen. Generelt, ved å bruke tilnærmingar med to forskjellige steglengder, $A(h)$ og

$A(h/t)$, har vi

$$A(h/t) - A(h) = \left(I + k\left(\frac{h}{t}\right)^n + \mathcal{O}\left(h^{n+1}\right)\right) - \left(I + kh^n + \mathcal{O}\left(h^{n+1}\right)\right)$$

her har vi skrive ut det første leddet i rest-funksjonen; målet er å bli kvitt h^n-leddet. Dette oppnår vi dersom vi gongar $A(h/t)$ med t^n:

$$t^n A(h/t) - A(h) = t^n I - I + \mathcal{O}\left(h^{n+1}\right)$$

Her har vi innlemma t^n i det asymptotiske leddet. Del med $t^n - 1$ og trekk I frå begge sider:

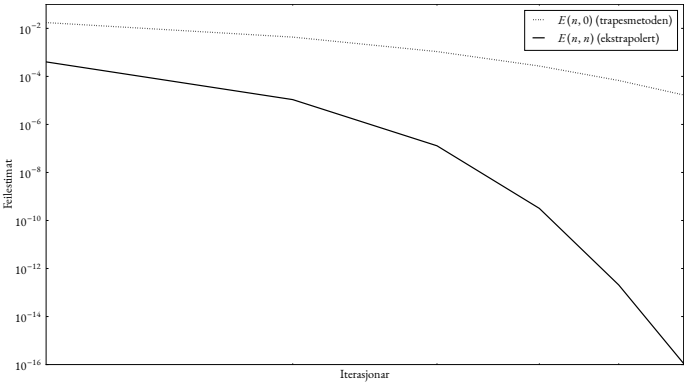
$$\frac{t^n A(h/t) - A(h)}{t^n - 1} - I = \mathcal{O}\left(h^{n+1}\right) \tag{1}$$

Altso har vi, ved å bruke *to* approksimasjonar med ulike steglengder, klart å hoste opp ein approksimasjon av *høgare orden*! Dersom vi startar med N estimat v. h. a. trapesmetoden kan vi ekstrapolere N gongar, og organisere approksimasjonane i ei nedretriangulær matrise. Organiserer vi utrekningane slik at kvar ekstrapolasjon $R(m, n)$ kun brukar dei to approksimasjonane $R(m-1, n)$ og $R(m-1, n-1)$ kan vi beregne approksimasjonane rad for rad, i staden for kolonne for kolonne. På denne måten oppnår vi ønska presisjon med ferrast mulig nodar i den opprinnelige trapesmetoden, og vi treng kun lagre to rader i gangen, så vi sparer minne. Denne metoden heiter *Rombergs metode*, og nokre estimat av integral er i **tabell 2.**

Tabell 2: *Estimasjonar v. h. a. Rombergs metode med toleranse* $\varepsilon = 10^{-10}$.

Integral	Estimat
$\int_{x=0}^{x=1} \mathrm{d}x \cos 2\pi x$	$3.30152344409 \times 10^{-17}$
$\int_{x=0}^{x=1} \mathrm{d}x \sqrt[3]{x}$	0.74998846514 ↓
$\mathrm{erf}(1) = \frac{2}{\sqrt{\pi}} \int_{x=0}^{x=1} \mathrm{d}x e^{-x^2}$	0.84270079294

Det ser ikkje ut til at toleransen ε er ei øvre skranke for feilen i Rombergs metode. Feilen i (↓) er om lag $\delta = 1.15 \times 10^{-5}$, som er større enn $\varepsilon = 10^{-10}$. For det same integralet, med den same toleransen, gjer den adaptive kvadraturmetoden svaret 0.75, som er eksakt, men tar om lag ti gongar so lang tid å kjøre.



Figur 2: *Feil i Rombergs metode. Det er tydeleg at elementa langs diagonalen kovergerer raskt samanlikna med elementa i første kolonne. Den akselererte følgja når presisjonen etter berre* 6 *iterasjonar.*

II. Simulasjon av fri, stiv lekam

Vi ønsker å simulere ein *fri, stiv lekam*, med andre ord ein lekam som ikkje lar seg deformere, som roterer fritt i rommet frå ein gitt start-tilstand.

Fri rotasjon betyr at netto påført dreiemoment er null. At lekamen ikkje lar seg deformere medfører at treighetsmomentet er konstant. (massen flyttar seg ikkje relativt til rotasjons-aksen) Dette betyr at normen til dreieimpulsen, og rotasjonsenergien er bevart. [1] Ingenting hindrar dreieimpulsen i å endre retning.

i. Definisjon av problemet

Differensiallikninga [1, Namn på symbola er endra for å passe vår oppgåvetekst]

$$\mathrm{T}\frac{\partial \omega}{\partial t} + \omega \times \mathrm{T}\omega = \mathbf{M} \tag{2}$$

skildrar rotasjonen til eit stivt legeme. Her er $\mathbf{M}$ påført dreiemoment, T treighetsmoment, og ω vinkelfart. Per antagelse er $\mathbf{M} = 0$. Innfør substitusjonen

$$\mathbf{m} = \mathrm{T}\omega$$

$$\implies \frac{\partial \mathbf{m}}{\partial t} = \mathrm{T}\frac{\mathrm{d}\omega}{\mathrm{d}t} \quad \text{og} \quad \omega = \mathrm{T}^{-1}\mathbf{m}$$

i (2), trekk frå $\mathrm{d}\mathbf{m}/\mathrm{d}t$ på begge sider, og snu kryssproduktet for å få like forteikn. Samtidig innfører vi notasjonen $\dot{f} = \mathrm{d}f/\mathrm{d}t$.

$$\dot{\mathbf{m}} = \mathbf{m} \times \mathrm{T}^{-1}\mathbf{m} \tag{2*}$$

Bevaringslovene nevnt til å byrje med gir oss:

$$\gamma = \|\mathbf{m}\|^2 = \mathbf{m}(t) \cdot \mathbf{m}(t) \tag{3}$$

$$E = \frac{1}{2}\mathbf{m}(t) \cdot \mathrm{T}^{-1}\mathbf{m}(t) \tag{4}$$

Der (3) er ei sfære med radius $\|\mathbf{m}\|$, og (4) er ei ellipse. Sidan både γ og E er konstantar er løysingane $\mathbf{m}$ begrensa til å sitje på skjæringa mellom flatene.

Anta $\mathrm{T} = \mathrm{diag}(\mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_3)$. Ettersom T er diagonal er $\mathrm{T}^{-1} = \mathrm{diag}(1/\mathrm{I}_1, 1/\mathrm{I}_2, 1/\mathrm{I}_3)$. Skriv $\mathbf{m} = (x\ y\ z)$, og skriv ut (2*) på komponentform:

$$\underset{\dot{\mathbf{m}}}{\begin{pmatrix}\dot{x}(t)\\ \dot{y}(t)\\ \dot{z}(t)\end{pmatrix}} = \underset{\mathbf{m}}{\begin{pmatrix}x(t)\\ y(t)\\ z(t)\end{pmatrix}} \times \left[\underset{\mathrm{T}^{-1}}{\begin{pmatrix}1/\mathrm{I}_1 & 0 & 0\\ 0 & 1/\mathrm{I}_2 & 0\\ 0 & 0 & 1/\mathrm{I}_3\end{pmatrix}} \underset{\mathbf{m}}{\begin{pmatrix}x(t)\\ y(t)\\ z(t)\end{pmatrix}} \right]$$

Rekn ut kryssproduktet:

$$\underset{\dot{\mathbf{m}}}{\begin{pmatrix}\dot{x}(t)\\ \dot{y}(t)\\ \dot{z}(t)\end{pmatrix}} = \underset{f(t,\mathbf{m})}{\begin{pmatrix}Ay(t)z(t)\\ Bx(t)z(t)\\ Cx(t)y(t)\end{pmatrix}} \tag{5}$$

der A, B og C er konstantar

$$A = 1/I_3 - 1/I_2$$
$$B = 1/I_1 - 1/I_3$$
$$C = 1/I_2 - 1/I_1$$

ii. Implisitt Runge-Kutta midtpunkt-metode

Vi ønsker å løyse differensiallikningar av sorten

$$\dot{y} = f(t, y)$$

numerisk, ved hjelp av Runge-Kutta metoden

$$y_{n+1} = y_n + hf\left(t_n + \frac{h}{2}, \frac{1}{2}(y_n + y_{n+1})\right), \tag{6}$$

kalt *implisitt midtpunktmetode*, fordi y_{n+1} avheng av eit estimat for $y_{n+1/2}$ (derav implisitt) for å estimere tangenten i midtpunktet mellom t_n og t_{n+1}. Substituer

$$u = \frac{1}{2}(y_n + y_{n+1}) \implies y_{n+1} = 2u - y_n$$

for å forenkle notasjonen litt. Dette gir likningssystemet

$$2u - y_n = y_n + hf\left(t + h/2, u\right)$$

som vi kan skrive

$$u = y_n + \frac{h}{2} f\left(t + h/2, u\right) \tag{7}$$

og dette må løysast med omsyn til u for kvart tidssteg. Systemet vi er interresert i har tre variablar; x, y og z. Med $\mathbf{u} = (x\ y\ z)$ kan vi definere

$$\mathbf{F}(\mathbf{u}) = \mathbf{u}_n + f(t_n + h/2, \mathbf{u}) - \mathbf{u} \tag{8}$$

der $\mathbf{u}_n$ er førre iterasjon, og søke etter nullpunkt ved hjelp av ein kjend løysar for ulineære likningssystem, til dømes Newtons metode. (som er kjend frå tidlegare øvingar, men eg gjentar det mest grunnleggande for kompletthets skyld)

Gitt Jacobian-matrisa til $\mathbf{F}$, $\mathcal{J}_\mathbf{F}$, kan vi bruke den iterative prosessen

$$\mathbf{u} \longleftarrow \mathbf{u} - \mathcal{J}_\mathbf{F}^{-1}\mathbf{F}(\mathbf{u})$$

med $\mathbf{u} = \mathbf{u}_n$ som startverdi. Iterasjonen er slutt når antal iterasjonar har nådd ei øvre grense, eller når $\|\mathbf{F}(\mathbf{u})\| < \varepsilon$, for ein gitt toleranse ε. Med $y_{n+1/2} = \mathbf{u}$ reknar vi ut y_{n+1}:

$$y_{n+1} = y_n + hf\left(t + h/2, y_{n+1/2}\right) \tag{9}$$

og dette fullfører éin iterasjon av Runge-Kutta-metoden.

iii. Implementasjon av RK-metoden

Vi brukar eit symbolsk verktøy for å rekne ut (9) i forkant, til dømes `sympy`. Den eine haka ved denne implementasjonen er at differensiallikninga vi vil løyse må vere definert berre ved hjelp av funksjonar som er kompatible med `sympy`, men det er enklare å ha eit eksplisitt uttrykk for Jacobian-matrisa ved kvart tidssteg.

Det er enklare å behandle uttrykk der dimensjonen er kjend. `sympy` *har* funksjonalitet for å lage vektorar med symbol, så det er (i prinsippet) mogleg å implementere metoden for N dimensjonar, men element i desse vektorane får obskure navn som gjer feilsøking til eit mareritt. Implementasjonen (**appendix C**) er derfor begrensa til tre dimensjonar.

iv. Modifisert Eulers metode

Vi implementerer òg den eksplisitte RK-metoden (**appendix D**)

$$y_{n+1/2} = y_n + \frac{h}{2} f\left(t_n, y_n\right)$$
$$y_{n+1} = y_{n+1/2} + hf\left(t_n + 1/2, y_{n+1/2}\right)$$

for å samanlikne den implisitte metoden med ein eksplisitt metode. Metoden treng ingen forklaring, sidan den er eksplisitt.

v. Numerisk Løysing av Likninga

I **listing 1** vert bruk av løysarane demonstrert.

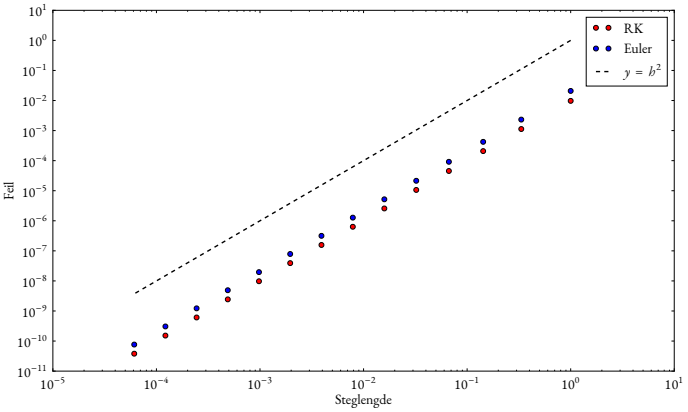
Listing 1: *Bruk av RK-løysarane*

```
1  def f(t, m):
2      x, y, z = m
3      return [ (1/I3 - 1/I2) * y * z ,
4               (1/I1 - 1/I3) * x * z ,
5               (1/I2 - 1/I1) * x * y ]
6
7  t = np.linspace(0, 150, 150)
8
9  y0 = np.array([2, 3, 4], dtype=float)
10 y0 *= 1/linalg.norm(y0)
11
12 x, y, z = RK(f, y0 = y0, t = t)
```

vi. Kvadratisk konvergens

Vi brukar ei svært nøyaktig algoritme frå `sympy`, med svært kort steglengde, for å estimere $\mathbf{m}(t = 1)$ so nøyaktig som mogleg. Dersom vi antar at denne verdien er om lag eksakt, samanlikna med verdiane vi oppnår med våre metodar med relativt lange steglengder kan vi undersøkje korleis feilen oppfører seg når steglengda minkar. Feilen kan ein sjå i **figur 3**, med steglengda som abscisse.

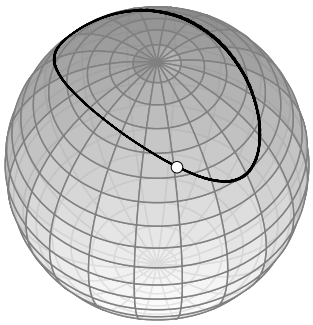
Ved sidan er der teikna ei linje $y = h^2$. I `loglog`-plottet har denne stigningstal $m = 2$. ($\log x^2 = 2 \log x$) Det er heilt klart at begge metodana har same stigningstal som linja, altso er begge kvadratisk konvergente i h, RK med litt lågare konstantfaktor enn Euler.



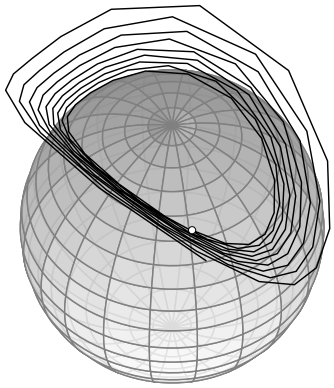
Figur 3: *Feilen i begge metodane er tydeleg kvadratiske.*

Merk at dette tidsrommet er svært kort. Vi er òg interresserte i å undersøkje kvar som skjer over lange tidsrom.

▽



(a) *Løysing v. h. a. implisitt midtpunkt-metode. Løysinga er presis, sjølv etter svært lang tid.*



(b) *Løysing v. h. a. eksplisitt modifisert Eulers metode. Løysinga vert ubrukeleg nesten med ein gang.*

Figur 4: *Løysingar av* (2*) *med initialverdiar* $\mathrm{T} = \mathrm{diag}(1, 2, 5)$ *og* $\mathbf{m}_0 = (2\ 3\ 4)$. *Tidsrommet er* $t = 150$, *med* $N = 150$ *steg. Initialverdien* $\mathbf{m}_0$ *er markert. Løysingane er plotta på ei sfære* $\gamma = \|\mathbf{m}_0\|^2$.

vii. Lange tidsrom

Løysingane produsert av løysarane er plotta i **figur 4a** (implisitt RK) og **figur 4b** (eksplisitt modifisert Euler). Det er tydeleg at den implisitte metoden produserer langt meir presise løysingar til dette initialverdiproblemet over lang tid: Den eksplisitte løysinga forlét sfæra nesten umiddelbart, og feilen veks raskt til å verte urimeleg stor.

Referansar

[1] J. R. Lien og G. Løvhøiden, *Generell fysikk for universiteter og høgskoler* (2015).

A. Simpson-kvadratur

Listing 2: *Adaptiv Simpson-kvadratur*

```
def S(f, a, b):
    return (f(a) + 4 * f((a + b) / 2) + f(b))  \
         * (b - a) / 6

def asm_quad(f, a, b, tol=1e-5):
    I0 = S(f, a, b)
    c = (a + b) / 2

    I = S(f, a, c) + S(f, c, b)
    err = abs(I - I0) / 15

    if err < tol:
        return I + (I - I0) / 15
    else:
        return asm_quad(f, a, c, tol=tol/2)  \
             + asm_quad(f, c, b, tol=tol/2)
```

B. Romberg-kvadratur

Listing 3: *Romberg-kvadratur*

```
def romberg(f, a, b, MAX_ITER=100, tol=1e-5):
    R = np.zeros((2, MAX_ITER))
    Rp, Rn = 0, 1

    h = b - a
    R[Rp, 0] = 0.5 * h * (f(a) + f(b))

    for n in range(1, MAX_ITER):
        h = h * 0.5
        L = np.linspace(a + h, b - h, 1 << n - 1)
        R[Rn, 0] = R[Rp, 0]/2 + np.sum(f(L))*h

        for k in range(1, n + 1):
            E = (R[Rn, k - 1] - R[Rp, k - 1])  \
              / ((1 << 2 * k) - 1)
            R[Rn, k] = R[Rn, k - 1] + E

        Rp, Rn = Rn, Rp

        if abs(E) < tol: break

    return R[Rp, n]
```

C. Implisitt Runge-Kutta

`sympy`-koden for likningssystemet er ikkje spesielt vanskelig, men den er lang og stygg, så eg har klipt den ut. Algoritma kan finnast i jupyter-notebook: `rk3d.pynb`, og der er sjølvsagt `sympy`-koden inkludert.

Listing 4: *Implisitt Runge-Kutta midtpunktmetode.*

```
def RK(f, y0, t):

    # (Sympy-junk removed here...)

    y = np.empty((len(t), 3))
    y[0] = y0

    for n in range(len(t) - 1):
        h = t[n + 1] - t[n]

        u = newton(
            lambda u:  F(u, y[n], t[n] + h/2, h),
            lambda u: Ji(u, y[n], t[n] + h/2, h),
            y[n]
        )

        m = np.array(f(t[n] + h/2, u), dtype=float)

        y[n + 1] = y[n] + h * m

    return y.T
```

I metoden inngår Newtons metode; denne har vi implementert i ei tidlegare øving, men legg den ved for kompletthets skuld:

Listing 5: *Newtons metode frå tidlegare øving.*

```
def newton(F, Ji, u0, tol=1e-10, maxiter=10):
    u = u0
    for _ in range(maxiter):
        u = u - Ji(u).dot(F(u))

        if linalg.norm(F(u)) < tol:
            break

    return u
```

D. Modifisert Euler

Listing 6: *Modifisert Eulers metode (eksplisitt)*

```
1  def euler(f, y0, t):
2      y = np.empty((len(t), 3))
3      y[0] = y0
4
5      for n in range(len(t) - 1):
6          h = t[n + 1] - t[n]
7
8          u = y[n] + np.array(f(t[n], y[n])) * h / 2
9          m = np.array(f(t[n] + h/2, u), dtype=float)
10         y[n + 1] = y[n] + h * m
11
12     return y.T
```