Varmeledende metallstang

Numerisk løsning av varmeledningsligningen
TFY4165 Termisk fysikk - Prosjekt

Innledning og noe teori

Varmeledningsligningen i én dimensjon er

$$\frac{\partial u}{\partial t} = D_T \frac{\partial^2 u}{\partial x^2}, \tag{1}$$

der det er antatt at termisk diffusivitet D_T er posisjonsuavhengig og $u = u(x,t)$. I dette prosjektet skal denne ligningen løses numerisk, ved hjelp av det som kalles for differansemetoder (finite-difference methods). Du skal se at ulike løsningsmetoder har ulike *stabilitetsområder*, slik at de egner seg til ulike situasjoner. Vi skal ikke gå dypt inn på hva man mener med begrepet *stabilitet*, men man kan si at en løsningsmetode er stabil for de steglengdene der den numeriske løsningen er "nærme nok" den analytiske løsningen. På samme måte kan man si at en numerisk løsningsmetode er numerisk ustabil dersom den numeriske løsningen blir veldig stor for ligninger der den analytiske løsningen ikke blir det. Dette er sagt på en relativt banal måte, men du kommer til å lære mer om dette begrepet senere. Du vil uansett klare å se tydelig forskjell på en ustabil og en stabil løsningsmetode i denne oppgaven. Videre vil det komme fram hvordan diskretisering av en partiell differensialligning (PDE), som varmeledningsligningen er et eksempel på, kan skape et lineært system av ligninger. Fordelen med dette er at PDEer kan løses ved hjelp av kjent lineær matrise-algebra. I forbindelse med dette kommer du til å se noe av styrken til Python-modulen numpy, da denne er veldig godt egnet til å jobbe med lineære systemer.

Systemet du skal se på i dette prosjektet består av en metallstang med lengde L orientert langsmed x-aksen. Stangen har konstant tverrsnitt og er laget av et homogent materiale. Dette betyr at stangen ikke har noen uregelmessigheter som du trenger å ta høyde for. I tillegg er den perfekt isolert i alle andre retninger, slik at varme kun kan flyte langsmed x-aksen. Dermed er dette et én-dimensjonalt problem og vi kan bruke varmeledningsligningen i én dimensjon til å modellere situasjonen. Stangen har en sinusoidal varmefordeling ved starttiden $t = 0$. Endene til staven er koblet til "kalde" varmereservoarer, slik at temperaturen holdes ved null grader. Etter hvert som tiden går overføres det varme fra

stangen til de kalde varmereservoarene.

Det fysiske systemet uttrykkes matematisk som

$$u_t = u_{xx} \qquad 0 < x < L,\, t > 0$$

$$u(x,0) = f(x) = 6\sin\left(\frac{\pi x}{L}\right).$$

$$u(0,t) = g_0(t) = g_1(t) = u(L,t) = 0, \qquad t > 0,$$

der $L = 1$, $f(x)$ representerer initialbetingelsen, og $g_0 = g_1 = 0$ representerer grensebetingelsene. Her er det i tillegg brukt en annen, kortere notasjon for de partiellderiverte $u_t = \frac{\partial u}{\partial t}$ og $u_{xx} = \frac{\partial^2 u}{\partial x^2}$. Du ser sikkert også at $D_T = 1$.

Siden dette er et relativt enkelt system har det en analytisk løsning, som kan være grei å bruke for å forsikre oss om at den numeriske løsningen er en god approksimasjon. Den analytiske løsningen til systemet er

$$u(x,t) = 6\sin\left(\frac{\pi x}{L}\right)\exp\left\{-\left(\frac{\pi}{L}\right)^2 t\right\}. \tag{2}$$

Først skal systemet løses ved hjelp av Eulers metode. Dette klassifiseres som en *eksplisitt* metode, siden den beregner systemets tilstand ved et senere tidspunkt ved hjelp av systemets nåværende tilstand, Dette gjør eksplisitte metoder relativt enkle å implementere, samt lite beregningskrevende for en datamaskin.

Deretter skal vi løse systemet ved hjelp av Crank-Nicolson-metoden. Dette klassifiseres som en *implisitt* metode, siden den beregner systemets tilstand ved et senere tidspunkt ved å beregne en ligning bestående av både den nåværende og den kommende tilstanden til systemet. Dette medfører at de implisitte metodene ofte er mer kompliserte å implementere, samt mer beregningskrevende enn de eksplisitte metodene. Fordelen med å bruke en implisitt metode er at stabilitetsområdet til løsningsmetoden ofte er større enn for de eksplisitte.

Oppgaver

Oppgave 1

Vi fikser et tilfeldig tidspunkt t og diskretiserer høyresiden av varmeledningsligningen (1). Det antas at steglengden i x-retning er $h = \frac{1}{M+1}$ for et heltall M. Deretter definerer vi noder

$$x_m = hm, 0 \leq m \leq M + 1. \tag{3}$$

Oppgaven din er å utføre approksimasjonen av høyresiden ved hjelp av diskretisering med en type differansemetode (*finite difference*) som heter *central difference*. Denne ser ut som

$$\delta_h f(x) = f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right), \tag{4}$$

der operatoren δ ofte brukes som kjennetegn på denne i *finite difference*-teorien. Dersom du deler denne på $h = \Delta x$ får du en *finite difference quotient*, som kan brukes til å approksimere den enkeltderiverte

$$f'(x) \approx \frac{\delta_h f(x)}{h} = \frac{f\left(x + \frac{h}{2}\right) - f\left(x - \frac{h}{2}\right)}{h}. \tag{5}$$

Bruk dette til å finne en approksimasjon til den annenderiverte. **Hint**: Prøv å bruke δ_h to ganger på funksjonen f for å finne et uttrykk for en omtrentlig dobbeltderivert.

Sammen med initialbetingelsen og grensebetingelsene har du nå et veldefinert system av ordinære differensialligninger (ODE) i én kontinuerlig variabel t. Denne fremgangsmåten kalles gjerne for *semi-discretization*, nettopp fordi vi kun har diskretisert den ene variabelen, og står igjen med én kontinuerlig variabel i ligningene.

Oppgave 2

Løs systemet av ODEer fra oppgave 1, ved å diskretisere t-aksen også, med eksplisitt Eulers metode. Husk at en førsteordens ODE kan skrives på den generelle formen

$$\dot{y}(t) = f(t, y(t)). \tag{6}$$

Husk videre at Eulers eksplisitte metode er gitt som

$$y_{n+1} = y_n + kf(t_n, y_n), \tag{7}$$

der $k = \Delta t = t_{n+1} - t_n$ og $f(t_n, y_n) = \frac{\mathrm{d}y(t)}{\mathrm{d}t} = \dot{y}(t)$. Til tross for at Eulers metode her er oppført for skalarer fungerer metoden på samme måte for vektorer.

t-aksen diskretiseres som

$$t_n = nk, n = 0, 1, 2, ..., N \tag{8}$$

Dersom du har funnet rett uttrykk i oppgave 1 skal Eulers metode gi følgende

$$U_m^{n+1} = U_m^n + r(U_{m+1}^n - 2U_m^n + U_{m-1}^n), m = 1, ..., M, \tag{9}$$

der $U_m^n \approx u(x_m, t_n)$ brukes som notasjon for den numeriske tilnærmingen til $u(x,t)$, og $r = \frac{k}{h^2}$. Vi løser altså ikke ligningen for $m = 0$ ($x_0 = 0$) og $m = M + 1$ ($x_{M+1} = L$), da løsningen i disse punktene er gitt av grensebetingelsene. Ønsker å gjenta at indeksen m viser til diskretiseringen i x-retningen, mens indeksen n viser til diskretiseringen i t-retningen. For å skille mellom disse indeksene er n superscript og m subscript når vi skriver den numeriske approksimasjonen U_m^n. Dette systemet skal altså løses for hver x_m i x-retningen og hver t_n i t-retningen.

Algoritmen dere skal følge blir som følger

$U_m^0 \leftarrow f(x_m), m = 1, ..., M + 1$
for $n = 0, 1, 2, ..., N$ **do**
  $U_0^{n+1} \leftarrow g_0(t_{n+1})$
  $U_{M+1}^{n+1} \leftarrow g_1(t_{n+1})$
  $U_m^{n+1} \leftarrow U_m^n + r(U_{m+1}^n - 2U_m^n + U_{m-1}^n),\ m = 1, ..., M$
end for

Tips til implementasjonen av algoritmen: Bruk numpy-arrays og slicing for alt det er verdt. Man kan ofte eliminere én eller flere for-løkker på denne måten.

Den numeriske løsningen du har kommet fram til må visualiseres. **Plott løsningen på tre ulike måter**

- 3D-plot med utstrekning på x-aksen, tid t på y-aksen og intensitet på z-aksen.
- Heatmap med colorbar ved siden av.
- 2D-subplot for 4 ulike tider, med utstrekning på x-aksen og intensitet på y-aksen.

I tillegg skal du plotte error mellom den oppgitte analytiske løsningen og den numeriske løsningen du har funnet i et 3D-plot.

Du skal løse og plotte systemet, slik som beskrevet ovenfor, for ulike verdier av M og N. Som en konsekvens av dette vil steglengdene i hver av retningene, k og h, samt verdien r, variere. Disse er avgjørende for stabiliteten til løsningsmetoden, noe du kommer til å se her. Verdiene du skal teste for er

- M = 19, N = 500
- M = 25, N = 700

- M = 35, N = 700

Hva ser du? Hva slags verdier har r i hvert av tilfellene? Du oppdager at løsningsmetoden er stabil i de to første tilfellene, men ustabil i det siste tilfellet. Dette er forventet, da det er mulig å vise at Eulers metode er stabil når

$$r = \frac{\Delta t}{\Delta x^2} = \frac{k}{h^2} \leq \frac{1}{2}, \tag{10}$$

noe som underbygger det du har sett når du har løst systemet for ulike verdier.

Oppgave 3

I forrige oppgave oppdaget du at Eulers metode blir ustabil for visse diskretiseringer, og kan i slike tilfeller ikke brukes til å gi oss en god numerisk approksimasjon til den analytiske løsningen. Ofte er det ikke tilstrekkelig å løse PDE-/ODEer med slike eksplisitte metoder. I denne oppgaven skal vi se på løsning av systemet fra oppgave 1 ved hjelp av en implisitt metode ved navn Crank-Nicolson. Her skal vi se at stabilitetsområdet er større enn for Eulers metode, slik at den kan brukes for flere typer diskretiseringer.

Vi skal betrakte systemet på en litt annen måte en tidligere. Systemet vi har er

$$\dot{v}_m(t) = \frac{v_{m+1}(t) - 2v_m(t) + v_{m-1}(t)}{h^2}, \quad i = 1, ..., M, \quad v_0(t) = v_{M+1}(t) = 0, \tag{11}$$

der $v_m(t)$ representerer approksimasjonen til $u(x_m, t)$ for kontinuerlig t. Dette er, som sagt, et lineært system av ODEer. Derfor kan det betraktes som en matriseligning

$$A_h \mathbf{V} = \mathbf{G}, \tag{12}$$

der A_h er en $M \times M$-matrise, $\mathbf{V} \in \mathbb{R}^M$ og $\mathbf{G} \in \mathbb{R}^M$ og

$$A_h := \frac{1}{h^2} \begin{bmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{bmatrix}, \quad \mathbf{V} := \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_M \end{bmatrix}, \quad \mathbf{G} := \dot{\mathbf{V}} - \frac{\mathbf{g}(t)}{h^2} := \begin{bmatrix} \dot{v}_1 \\ \dot{v}_2 \\ \vdots \\ \dot{v}_M \end{bmatrix},$$

som er et spesialtilfelle siden $\mathbf{g}(t) = \vec{0}$ i vårt system. En matrise på formen til A_h får ofte tilnavnet *tridiagonal* matrise. Denne spesielle typen av tridiagonale matriser, der elementene i hver av de tre diagonalene er like, kalles videre for en *Toeplitz* matrise.

Nå skal dette systemet løses med det som kalles trapesmetoden, som er en type Runge-Kutta-metode, for samme diskretisering av t-aksen som i oppgave 1. Denne metoden for

en ODE er en måte å utlede den velkjente Crank-Nicolson-metoden, som brukes for å løse PDEer.

Den generelle formen på trapesmetoden er

$$y_{n+1} = y_n + \frac{k}{2}\left(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right), \tag{13}$$

der $k = \Delta t = t_{n+1} - t_n$ og $f(t_n, y_n) = \frac{\mathrm{d}y(t)}{\mathrm{d}t} = \dot{y}(t)$. Denne ser vi at er implisitt, siden y_{n+1} inngår både på venstre og på høyre side av uttrykket. Akkurat som for Eulers metode ser denne metoden lik ut for skalarer som for vektorer.

Fra systemet (12) har vi at

$$A_h \mathbf{V} = \mathbf{G} = \dot{\mathbf{V}}, \tag{14}$$

siden $\mathbf{g}(t) = \vec{0}$ i systemet. Dette medfører at funksjonen f i trapesmetoden er en vektorfunksjon $f : \mathbb{R}^M \to \mathbb{R}^M$, og $f(\mathbf{V}) = A_h\mathbf{V}$. Dermed gir trapesmetoden

$$\begin{aligned} \mathbf{V}_{n+1} &= \mathbf{V}_n + \frac{k}{2}\left(A_h\mathbf{V}_n + A_h\mathbf{V}_{n+1}\right) \\ &= \mathbf{V}_n + \frac{k}{2}A_h\mathbf{V}_n + \frac{k}{2}A_h\mathbf{V}_{n+1}. \end{aligned} \tag{15}$$

Det aller første du bør gjøre er å skrive om den siste linja i ligning (15) til et system på formen

$$K\mathbf{V_{n+1}} = H\mathbf{V_n}. \tag{16}$$

Du skal altså finne K og H. Dette vil gjøre det enklere for deg når du skal implementere løsningen i Python. Når du bytter ut K og H med sine respektive verdier har du kommet fram til uttykket som ofte kalles for Crank-Nicolsons metode.

Du må konstruere matrisen A_h i Python, slik at du kan bruke den når du skal løse systemet ved hjelp av Crank-Nicolson. Du kan bruke `scipy.sparse.spdiags` til å konstruere den korrekte matrisen. Et eksempel på bruk er vedlagt nedenfor, men finn gjerne dokumentasjonen dersom du lurer på hvordan den fungerer.

```
from scipy.sparse import spdiags # Make sparse matrices with scipy.

# Example using spdiags.
data = np.array([[1, 2, 3, 4, 5], [1, 2, 3, 4, 5], [1, 2, 3, 4, 5]])
diags = np.array([0, -2, 3])
sp = spdiags(data, diags, 5, 5)
print(sp) # Shows how the data is saved in memory. Saves only necessary data!
print(sp.toarray()) # Shows how the tuples are bundled to a sparse matrix.
```

Fordelen med å bruke `scipy.sparse.spdiags` er at den lagrer akkurat nok informasjon til å klare å reprodusere matrisen. Hvis du kjører koden ovenfor ser du av linje 7 at matrisen lagres som en liste med tupler, som representerer koordinatene til hver av verdiene som ikke er null, samt verdien som tilhører hver koordinat. Dersom vi jobber med veldig store systemer er dette enda viktigere, da det ikke er noe poeng med lagring av null-verdier. Lagring på denne måten gjør at vi unngår unødvendige addisjoner og multiplikasjoner som vi vet blir null uansett.

Den siste delen av denne oppgaven som du må utføre er å løse systemet med Crank-Nicolson-metoden du har funnet. Husk, slik som tidligere, at du må sette grenseverdiene for hver diskrete verdi av t-aksen. Det som gjør denne implementasjonen litt mer krevende enn Eulers metode er at du her må løse et lineært system for hver iterasjon på t-aksen. I denne oppgaven skal du bruke `np.linalg.solve` for å løse dette lineære systemet. For å gjøre dette litt lettere for deg selv er det anbefalt å skrive Crank-Nicolson-metoden din slik at du får en lineæar matriseligning på formen $A\mathbf{x} = \mathbf{b}$, slik du er vant til. Et tips underveis: `np.identity` kan brukes for å lage en identitetsmatrise. I tillegg kan du bruke '@' for å utføre en matrisemultiplikasjon. **Etter at du har løst systemet skal du plotte løsningen for de samme verdiene av M og N som i oppgave 1**. Ser du forskjell på løsningen med Eulers metode og Crank-Nicolsons metode?

Du ser forhåpentligvis at Crank-Nicolson-metoden er stabil for alle tre settene med verdier som du har plottet. Det kan faktisk vises at Crank-Nicolson-metoden er ubetinget stabil [2]

Oppgave 4 - Frivillig

Denne oppgaven kan du gjøre dersom du ønsker å se hvordan du kan løse et lineært system, med en numerisk metode, uten å bruke `np.linalg.solve`.

Her skal du modifisere Crank-Nicolson-metoden fra oppgave 3. I stedet for å bruke `np.linalg.solve` skal du nå i stedet bruke en iterativ metode til å løse matriseligningen i hvert tidssteg. Du har vært borti iterative metoder i tidligere kurs (FY1003 - Elektrisitet og magnetisme), men da fikk du presentert metodene på komponentform. Her skal en iterativ metode heller føres opp på matriseform. Dette gjør at vi kan bruke kjente operasjoner fra lineær algebra, noe som gjør det enklere å holde styr på alle komponenter. Som en liten repetisjon er tanken bak de iterative metodene å gjette en initiell løsning, for deretter å generere en følge med approksimasjoner som forhåpentligvis konvergerer mot en god approksimasjon av den korrekte løsningen til systemet.

De iterative metodene er gode hjelpemidler, blant annet, når man skal løse veldig store systemer, som kan medføre stor kostnad ved bruk av direkte løsningsmetoder, som for

eksempel Gauss-eliminasjon. Metoden vi skal benytte oss av her heter *successive over-relaxation* (SOR). Et generelt system $A\mathbf{x} = \mathbf{b}$ kan skrives om til $(L + D + U)\mathbf{x} = \mathbf{b}$, der L, D og U inneholder henholdsvis nedre tredel, diagonalen og øvre tredel av matrisen A. Resten av verdiene i de respektive matrisene er 0. For dette systemet kan SOR formuleres som

$$\mathbf{x}_{k+1} = (\omega L + D)^{-1}[(1-\omega)D\mathbf{x}_k - \omega U\mathbf{x}_k + \omega\mathbf{b}],\ k = 0, 1, 2, ..., \tag{17}$$

der ω kalles for en relaksasjonsparameter. Hensikten bak parameteren er å prøve å fremskynde konvergensen til metoden. $\omega > 1$ gir over-relaksasjon. Her skal vi bruke $\omega = 1.2$. **Din oppgave blir å fullføre implementasjonen av koden til SOR-metoden nedenfor**.

```
def SOR(A, b, omega,u0,tol,maxiter):
    '''SOR method.
    Return the computed solution u.
    omega: Value of the relaxation parameter in SOR.
    u0: The initial value for the iteration.
    tol: The tolerance to be used in the stopping criterion (err < tol).
    maxiter: The maximum number of iterations.
    '''
    k = 0
    err = 2*tol # Starts the loop.

    # Set L, D and U from A.
    L = # ?
    D = # ?
    U = # ?
    dividend = omega*L+D
    dividend_inv = np.linalg.inv(dividend) # Calculate inverse with numpy.
    while err > tol:
        k += 1

        u = # ?


        err = np.linalg.norm(u-u0)
        if k >= maxiter:
            break

        # Remember to move the iteration along.
    return u
```

Husk å bytte ut `np.linalg.solve` med SOR i Crank-Nicolson-metoden. Du kan, for eksempel, sette tol $= 10^{-8}$, maxiter $= 1000$ og u0 lik en vektor med null-verdier. **Skriv gjerne ut kjøretiden til koden som benytter seg av `np.linalg.solve` og sammenlign med kjøretiden når du bruker SOR til å løse det lineære systemet**.

Hvilken implementasjon tror du har kortest kjøretid?

Selv om noe av poenget med denne oppgaven var å ikke bruke `np.linalg` brukes modulen likevel i linje 17, da vi trenger å finne en invers matrise. Det finnes naturligvis numeriske metoder for dette også, men det skal vi ikke komme inn på her.

Avslutningsvis

I denne øvingen har du fått litt erfaring med å løse PDEer numerisk med noen av de mange forskjellige metodene som finnes der ute. Vi har benyttet oss av en metode som kalles for *semi-discretization*, som er veldig nyttig i mange tilfeller, da den reduserer PDEer til et system av ODEer i én kontinuerlig variabel. Du har videre sett at Crank-Nicolson-metoden er stabil i tilfeller der Eulers metode ikke er stabil. Dette er illustrativt for den generelle trenden blant metodene: at de implisitte metodene er stabile for flere ulike diskretiseringer enn de eksplisitte. Ulempen med dem er at de ofte er mer kompliserte å implementere og krever større regnekraft enn de eksplisitte. Poenget er å vise at det finnes mange ulike måter å finne numeriske løsninger til PDEer (og ODEer), der alle metodene er best til sin unike situasjon. Det er opp til oss, som skal løse de fysiske problemene, å finne ut hvilken løsning som er best egnet til akkurat vår situasjon.

Krav til godkjenning

Kravene under må være oppfylt for å få godkjent prosjektet.

1. Besvarelsen skal leveres i Jupyter Notebook (.ipynb).
2. Figurene fra alle oppgavene skal være i notebooken du leverer.
3. Alle figurer skal ha figurtittel, samt aksetitler.

Referanser

[1] B. Owren. *TMA4212 Numerical solution of partial differential equations with finite difference methods*. 31. januar 2017. Lest: 18. juli 2020.

[2] T. Sauer. *Numerical Analysis, Second Edition*. Pearson, 2012.