Project 1, September 9, 2019

Department of Physics, University of Oslo, Norway.
Philip Niane og Rohullah Akbari

Abstrakt
I denne rapporten er det vist hvordan vi kan løse Poissons likning ved bruk av følgende numeriske løsningsmetoder (metodens hurtighet): Thomas algoritmen på en tridiagonal matrise (Rask), Thomas algoritmen der a_n over diagonalen er lik c_n under diagonalen (Raskest) og LU dekomposisjon (Tregest).

Contents

1 Introduksjon **2**

2 Teori **2**
2.1 Tridiagonal matrise og Thomas algoritme(Generell) 2
2.2 Tridiagonal matrise og Thomas algoritme(Spesialisert) 4
2.3 LU dekomposisjon 4

3 Metode **5**
3.1 Omskriving til formen $\mathbf{Av} = \tilde{\mathbf{b}}$ (1a) 5
3.2 Generell Thomas algoritme 6
3.3 Spesialisert Thomas algoritme 6
3.4 LU dekomposisjons algoritme 7
3.5 Metode for å regne ut feilen 7

4 Resultater **8**
4.1 Plott og algoritmetid 8
4.2 Feilen i metodene 8

5 Diskusjon **8**

6 Konklusjon **9**

7 Appendiks **9**
7.1 Utledning av betingelser til den spesialiserte Thomas algoritmen 9

8 Bibliografi **10**

1 Introduksjon

Målet med dette prosjektet er å lære om matriseoperasjoner i tillegg til å lære hvordan man programmerer med hensyn til dynamisk minneallokering i programmeringsspråket C++. For å lære dette er det tatt utgangspunkt i Poissons likning:

$$-u''(x) = f(x) \tag{1}$$

som helt tydelig er en differensiallikning i 1 dimensjon. I dette prosjektet skal vi vise hvordan man kan løse denne differensiallikningen via forskjellige numeriske metoder. De numeriske metodene vi skal bruke er Thomas algoritmen, Thomas algoritmen når elementene rett over- og rett under diagonalen er like og LU dekomposisjon. Disse metodene brukes til å løse problemer innenfor lineær algebra ved bruk av matriser. Dermed må vi skrive differensiallikningen vår over til matrise form. Vi viser hvordan dette gjøres senere i rapporten, men matriselikningen vår blir seende slik ut:

$$\mathbf{Av} = \overline{\mathbf{b}} \tag{2}$$

Når de forskjellige løsningsmetodene er utført, sammenlikner vi resultatene og effektiviteten til metodene.

2 Teori

2.1 Tridiagonal matrise og Thomas algoritme(Generell)

Vi kan skrive differensiallikningen vår på formen $\mathbf{Av} = \overline{\mathbf{b}}$ som videre kan skrives på følgende matriseform.

$$\mathbf{Av} = \begin{bmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{bmatrix}.$$

En tridiagonal matrise er en matrise på formen til A. Altså en matrise der elementene langs diagonalen er like, samtidig som elementene over diagonalen er like og elementene under diagonalen er like.

For å løse en slik matriselikning kan vi bruke en såkalt thomas algortime. Thomas algorithmen er en form for gauss eliminasjon, der vi radreduserer matrisen vår ved bruk av en såkalt forward substitution, så kjører vi en backward substitution for å gi oss løsningen vår.

Forward substitution radreduserer matrisen vår ved å først ta den første raden og multiplisere med $\frac{a_n}{b_n}$. Dette gjør første element i rad 1 og rad 2 like.

$$\mathbf{Av} = \begin{bmatrix} a_1 & c_1\frac{a_1}{b_1} & 0 & \dots & \dots & \dots \\ a_1 & b_2 & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{r}_1\frac{a_1}{b_1} \\ \tilde{r}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{r}_n \end{bmatrix}.$$

Videre er det lett å subtrahere bort første element i rad 2.

$$\mathbf{Av} = \begin{bmatrix} a_1 & c_1\frac{a_1}{b_1} & 0 & \dots & \dots & \dots \\ 0 & b_2 - c_1\frac{a_1}{b_1} & c_2 & \dots & \dots & \dots \\ & a_2 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_{n-1} & b_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} \tilde{r}_1\frac{a_1}{b_1} \\ \tilde{r}_2 - \tilde{r}_1\frac{a_1}{b_1} \\ \dots \\ \dots \\ \dots \\ \tilde{r}_n \end{bmatrix}.$$

Så definerer vi $s_n = b_n - c_{n-1}\frac{a_{n-1}}{s_{n-1}}$ og $f_n = \tilde{r}_n - \tilde{r}_{n-1}\frac{a_{n-1}}{s_{n-1}}$. Dette er algoritmen for forward substitution.
Dette gir videre (når forward substitutionen fortsetter, og vi setter) følgende matrise:

$$\mathbf{Av} = \begin{bmatrix} s_1 & c_1 & 0 & \dots & \dots & \dots \\ 0 & s_2 & c_2 & \dots & \dots & \dots \\ & 0 & s_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & 0 & s_{n-1} & c_{n-1} \\ & & & & 0 & d_n \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \dots \\ \dots \\ \dots \\ f_n \end{bmatrix}.$$

Hvor vi antar at a_1=s_1 og $\tilde{r_1}$=f_1.

Videre kan vi utføre backward substitution for å finne vektor v. "Backward" kommer fra at vi starter nederst i matrisen og jobber oss oppover gjennom likningsettene. Vi finner v_n fra likningen $d_n v_n = f_n$. Siden radene over har to ukjente der en av de ukjente regnes ut fra raden under kan vi regne oss helt opp til toppen og stå igjen med hele vektor v. Her vil v_n og v_{n-1} være definert som

$$v_n = \frac{f_n}{s_n} \tag{3}$$

og

$$v_{n-1} = \frac{f_{n-1} - v_n c_{n-1}}{s_{n-1}} \tag{4}$$

2.2 Tridiagonal matrise og Thomas algoritme(Spesialisert)

I den spesialiserte versjonen av Thomas algoritmen antar vi at elementene langs diagonalen til a=-1, elementene til b diagonalen =2 og elementene til c=-1. Altså blir matrisen vår **A** slik:

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{bmatrix},$$

For å løse denne matrisen ved bruk av forward- og backward substitution må vi implementere de nye betingelsene våre i den generelle algoritmen. De nye betingelsene blir utledet i appendiks. Nye betingelser:

For forward substitution:

$$s_n = 2 - \frac{n+1}{n} \tag{5}$$

og

$$f_n = \tilde{r}_n + \frac{f_{n-1}}{s_{n-1}} \tag{6}$$

For backward substitution:

$$v_{n-1} = \frac{f_{n-1} - v_n}{s_{n-1}} \tag{7}$$

2.3 LU dekomposisjon

LU dekomposisjon er en metode som brukes til å løse likningsett. Dette er naturligvis også en metode som kan brukes til å løse matriselikningen vår.
LU dekomposisjon går ut på at matrisen vår A, deles opp i to nedre- og øvre triangulære matriser.

$$\mathbf{A} = \mathbf{LU} \tag{8}$$

hvor L er den nedre triangulære matrisen, mens U er den øvre triangulære matrisen. Siden likningsettet vårt er

$$\mathbf{Av} = \overline{\mathbf{b}} \tag{9}$$

Får vi følgende når vi implementerer LU istedenfor A:

$$\mathbf{Av} = \mathbf{LUv} = \mathbf{b} \tag{10}$$

Dette kan videre deles opp i 2 likninger:

$$\mathbf{Ly} = \mathbf{b} \;\; og \;\; \mathbf{Uv} = \mathbf{y} \tag{11}$$

I c++ kan matrisen A dekomponeres til L og U matriser ved funksjonen: **ludcmp**(**doublea**, **intn**, **intindx**, **doubled**). Så kan man bruke funksjonen: **lubksb**(**doublea**, **intn**, **intindx**, **doublew**) til å finne vektoren v ved bruk av backward substitution.(Funksjoner hentet fra lecture notes på semestersiden til fys3150)

3 Metode

3.1 Omskriving til formen $\mathbf{Av} = \tilde{\mathbf{b}}$ (1a)

Om det tas utgangspunkt i følgende tilnærming til den andre deriverte:

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{for } i = 1, \ldots, n, \tag{12}$$

Vi skriver om til:

$$-v_{i+1} - v_{i-1} + 2v_i = h^2 f_i \tag{13}$$

Så regner vi ut for i=1...3:

$$i = 1 \qquad -v_2 - v_0 + 2v_1 = h^2 f_1 \tag{14}$$

$$i = 2 \qquad -v_3 - v_1 + 2v_2 = h^2 f_2 \tag{15}$$

$$i = 3 \qquad -v_4 - v_2 + 2v_3 = h^2 f_3 \tag{16}$$

Om i fortsetter til n-1 ser vi at venstresiden av likningssettet kan skrives som en matrise **A** multiplisert med en vektor v.
Vi følger mønsteret og ser at om

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & \ldots & \ldots & 0 \\ -1 & 2 & -1 & 0 & \ldots & \ldots \\ 0 & -1 & 2 & -1 & 0 & \ldots \\ & \ldots & \ldots & \ldots & \ldots & \ldots \\ 0 & \ldots & & -1 & 2 & -1 \\ 0 & \ldots & & 0 & -1 & 2 \end{bmatrix},$$

og

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \ldots \\ \ldots \\ \ldots \\ v_n \end{bmatrix}$$

Vil dette stemme for likningssettet vårt og kan skrives på formen $\mathbf{Av} = \tilde{\mathbf{b}}$ når $\tilde{b}_i = h^2 f_i$. v_0=0 på grunn av grensebetingelser.

3.2 Generell Thomas algoritme

I koden for den generelle tridiagonale matrisen og i videre utskrifter av koden vil u og s i koden være lik v og f i rapporten hhv.

```
90  void gauss_generel(int n, double* a, double* b, double* c, double* u, double* s)
91  {
92          \\Forward substitution
93          for (int i = 2; i < n + 1; i++) {
94                  b[i] = b[i] - ((a[i - 1] * c[i - 1]) / b[i - 1]);
95                  s[i] = s[i] - ((s[i - 1] * a[i - 1]) / b[i - 1]);
96          }
97          u[n + 1] = s[n + 1] / b[n + 1];
98
99          \\Backward substitution
100         for (int i = n + 1; i > 1; i--)
101         {
102                 u[i - 1] = (s[i - 1] - c[i - 1] * u[i]) / b[i - 1];
103         }
104 }
```

Figure 1: Generell algoritme.

I algoritmen i figur 1 er det først utført en forward substitution der vi kjører en for-loop for å omgjøre kolonnene til pivotkolonner som forklart i teorien. Deretter er backward substitution brukt i en for-loop til å loope seg oppover frem til vektor v.

3.2.1 Floating point operations- Generel algoritme

For å regne ut antall floating point operations må vi se hvor mange operasjoner som blir utført i algoritmen vår. Algoritmen blir utført fra 1 til N, altså N-1 ganger. Vi ser at i forward substitutionen er det utført 3 operasjoner (subtraksjon, multiplikasjon og divisjon) over 2 linjer=6 for hver iterasjon. I backward substitution utføres det også 3 operasjoner(subtraksjon, multiplikasjon og divisjon). Altså står vi foreløpig igjen med 9(N-1) floating point operations. Itillegg må vi trekke fra 1 operasjon da vi regner ut 1 mindre verdi for vektor v enn N-1.
Dette gir 9N-8 operasjoner for den generelle algoritmen i vår kode.
Det er mulig å sette en variabel lik a[i-1]/b[i-1] for så å bruke denne variabelen i andre linje i forward substitutionen for å senke antall floating point operators med N-1.

3.3 Spesialisert Thomas algoritme

I koden for den spesialiserte tridiagonale matrisen og i videre utskrifter av koden vil u og s i koden være lik v og f i rapporten hhv.

```
106 void gauss_special(int n, double* b, double* u, double* s
107 {
108         \\Forward substitution
109         for (int i = 2; i < n + 1; i++) {
110                 b[i] = (i + 1.0) / (i);
111                 s[i] = s[i] + (s[i - 1] / b[i - 1]);
112         }
113         \\Backward substitution
114         for (int i = n + 1; i > 1; i--)
115         {
116                 u[i - 1] = (s[i - 1] + u[i]) / b[i - 1];
117         }
```

Figure 2: Spesiell algoritme.

I koden for den spesielle Thomas algoritmen er det tatt hensyn til at elementene langs diagonalen til a=-1, elementene til b diagonalen =2 og elementene til c=-1, som vi utledet i teoridelen.

3.3.1 Floating point operations- Spesialisert Thomas algoritme

Når det gjelder antall floating point operations ser vi at den første linjen i forward substitution kun avhenger av i og kan dermed regnes ut på forhånd. Operasjoner som kan regnes ut på forhånd regnes ikke med i floating point operations. I andre linje i forward substitution og i backward substitutionen har vi 2 operasjoner hver(addisjon og divisjon). Altså står vi foreløpig igjen med 4(N-1) floating point operations. Itillegg må vi trekke fra 1 operasjon da vi regner ut 1 mindre verdi for vektor v enn N-1.
Dette gir 4N-3 operasjoner for den spesielle algoritmen i koden vår.

3.4 LU dekomposisjons algoritme

Fyll inn om LU når koden er ferdig

3.5 Metode for å regne ut feilen

Metoden for å finne feilen til aproksimasjonene vår er å bruke følgende formel, hvor vi tar absoluttverdien av verdien vår og subtraherer med den eksakte verdien. Dette divideres på den eksakte verdien;

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right),$$

Deretter plottes maks verdien til feilen som funksjon av $log_{10}(h)$.
Plottene for feilene kan ses under i resultatdelen.

4 Resultater

4.1 Plott og algoritmetid

Plottet vårt til Thomas algoritmen kan ses i figuren under, her er er den eksakte løsningen plottet sammen med Thomas algoritmen forskjellige verdier av n.

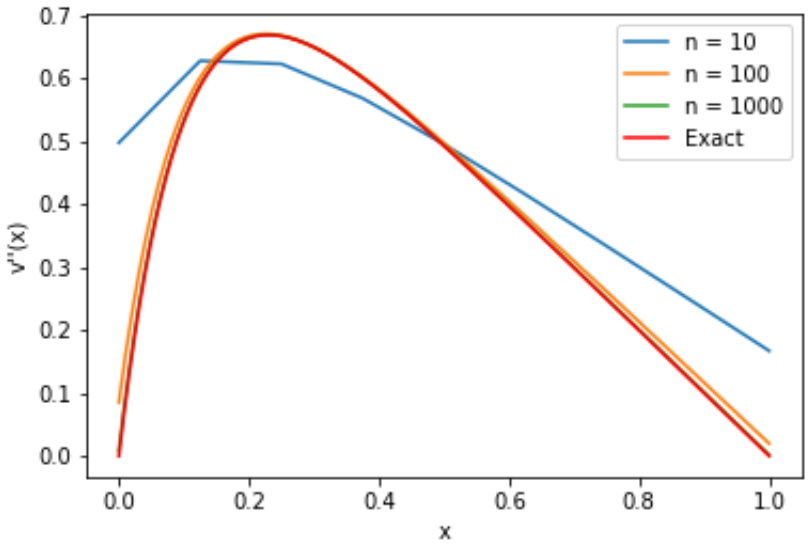

Figure 3: Eksakt løsning plottet sammen med metoden vår

4.2 Feilen i metodene

5 Diskusjon

I plottet med den eksakte løsningen sammen med vår løsning, er det lett å legge merke til at jo flere iterasjoner n vi har, jo bedre blir løsningen. n=10 gir en relativt dårlig løsning da disse grafene kunne overlappet mye bedre. Når n=100 derimot begynner det å nærme seg en bedre løsning. I plottet er det litt feil i henhold til den eksakte løsningen ved start og slutt, men overlappet er godt midt på. Når n=1000 ser det ut som vi har fult overlapp. Denne tilnærmingen er altså veldig god!

Når det gjelder LU dekomposisjonen ser vi at også her i likhet med Thomas algoritmen, vil lave verdier for n gi en relativ stor feil. Når n øker, minker naturligvis feilen itillegg, n=1000 gir så og si fullstendig overlapp mellom Thomas algoritmen og den eksakte løsningen.

Når det gjelder plottet over feilene våre ser vi at ... les om dette i kompendiet

I tabellen over tiden det tar å utføre algoritmene er det lett å se at den spesialiserte Thomas algoritmen er den raskeste, mens LU dekomposisjon tar lengst tid. Naturligvis øker også tidsforskjellen mellom algoritmene jo flere operasjone som må regnes ut.

6 Konklusjon

Vi har altså sett på hvordan vi kan løse Poissons likning i 1 dimensjon, ved å skrive differensiallikningen over på matriseform og angripe problemet ved å se på problemstillingen som et problem innenfor lineær algebra. Det har også blitt vist at den spesialiserte Thomas algoritmen som ble bestemt høyere opp i prosjektet var den mest effektive numeriske løsningsmetoden, etterfulgt av den generelle Thomas algoritmen og LU dekomposisjon til slutt. Selv om metodene varierte i effektivitet er fortsatt alle metodene fullt brukbare så lenge antall ittereringer n er høy nok.

7 Appendiks

7.1 Utledning av betingelser til den spesialiserte Thomas algoritmen

Siden vi tar utgangspunkt i algoritmen for å løse en generell tridiagonal matrise, tar vi utgangspunkt i $s_n = b_n - c_{n-1}\frac{a_{n-1}}{s_{n-1}}$ og $f_n = \tilde{r}_n - \tilde{r}_{n-1}\frac{a_{n-1}}{s_{n-1}}$. Som sagt antar vi at $a_1..a_n = c_1..c_n = -1$ og $c_1..c_n = 2$. Dette gir:

$$s_n = b_n - c_{n-1}\frac{a_{n-1}}{s_{n-1}} \tag{17}$$

$$s_n = 2 - \frac{1}{s_{n-1}} \tag{18}$$

$$s_1 = 2 \ (pga. s_1 = b_1 = 2) \tag{19}$$

$$s_2 = 2 - \frac{1}{2} = \frac{3}{2} \tag{20}$$

$$s_3 = 2 - \frac{1}{\frac{3}{2}} = \frac{4}{3} \tag{21}$$

$$s_4 = 2 - \frac{1}{\frac{4}{3}} = \frac{5}{4} \tag{22}$$

$$s_5 = 2 - \frac{1}{\frac{5}{4}} = \frac{6}{5} \tag{23}$$

Som gir:

$$s_n = 2 - \frac{n+1}{n} \tag{24}$$

For $f_n = \tilde{r}_n - \tilde{r}_{n-1}\frac{a_{n-1}}{s_{n-1}}$ bytter vi ut a_{n-1} med -1 og $\tilde{r}_{n-1}$ med f_{n-1}. Dette gir:

$$f_n = \tilde{r}_n + \frac{f_{n-1}}{s_{n-1}} \tag{25}$$

For backward substitution trenger vi bare å fylle inn -1 for c_{n-1} og få følgende:

$$v_{n-1} = \frac{f_{n-1} - v_n}{s_{n-1}} \tag{26}$$

Dermed har vi spesialisert algoritmen vi trenger til matriseproblemet vårt.

8 Bibliografi

Morten Hjorth-Jensen, august 2015, Computational Physics Lecture Notes,
(https://github.com/CompPhysics/ComputationalPhysics/blob/master/doc/Lectures/lectures2015.pdf)