Prosjekt 1

Live Wang Jensen

September 17, 2016

Abstract

I dette prosjektet skal vi se nærmere på en numerisk løsning av den velkjente Poisson-ligningen, hvor Dirichlets grensebetingelser er brukt. Den andrederiverte er blitt tilnærmet med en tre-punkts-formel, og selve problemet løses ved hjelp av et lineært ligningssett. Vi skal løse disse ligningene på to ulike måter; ved Gauss-eliminasjon og ved LU-faktorisering. De to løsningsmetodene skal så sammenlignes når det kommer til antall FLOPS og beregningstid.

1 Introduksjon

Vi starter med å se på den én-dimensjonale Poisson ligningen for en sfærisk symmetrisk funksjon. Denne ligningen løses numerisk ved å dele opp, eller disktretisere, intervallet $x \epsilon$ [0,1]. Løsningen f på ligningen er gitt. Det vil vise seg at det å finne en diskret løsning vil være det samme som å løse et lineært ligningssett, $A\mathbf{u} = \tilde{\mathbf{b}}$, hvor matrisen A er en såkalt tridiagonal matrise, noe som forenkler Gauss-eliminasjonen betraktelig. Selve metoden er implementert i et C++-program, hvor vi har variert antall grid-points n. Den numeriske løsningen sammenlignes så med den analytiske løsningen. Deretter beregnes den maksimale relative feilen i den numeriske løsningen. Helt til slutt skal vi bruke LU-faktorisering på matrisen A til å finne den numeriske løsningen på Poisson ligningen. Antall FLOPS og beregningstid sammenlignes så mellom de to løsningmetodene.

2 Teori

Mange differensialligninger innenfor fysikk kan skrives på formen

$$\frac{d^2y}{dx^2} + k^2(x)y = f(x) \tag{1}$$

hvor f er det inhomogene leddet i ligningen, og k^2 er en reell funksjon. Et typisk eksempel på en slik ligning finner vi i elektromagnetismen, nemlig Poisson-ligningen:

$$\triangle^2 \Phi = -4\pi\rho(\mathbf{r}) \tag{2}$$

Her er Φ det elektrostatiske potensialet, mens $\rho(\mathbf{r})$ er ladningsfordelingen. Dersom vi antar at Φ og $\rho(\mathbf{r})$ er sfærisk symmetriske, kan vi forenkle ligning (2) til en én-dimensjonal ligning,

$$\frac{1}{r^2}\frac{d}{dr}\left(r^2\frac{d\Phi}{dr}\right) = -4\pi\rho(r) \tag{3}$$

Hvis vi bruker at $\Phi(r) = \phi(r)/r$, kan ligningen skrives som

$$\frac{d^2\phi}{dr^2} = -4\pi r\rho(r) \tag{4}$$

Det inhomogene leddet f er nå gitt ved produktet $-4\pi\rho$. Dersom vi lar $\phi \to u$ og $r \to x$, ender vi opp med en generell, én-dimensjonal Poisson ligning på formen

$$-u''(x) = f(x) \tag{5}$$

Det er denne ligningen vi skal se nærmere på i dette prosjektet. Nærmere bestemt skal vi løse ligningen

$$-u''(x) = f(x), \quad x \in (0,1), \quad u(0) = u(1) = 0 \tag{6}$$

Vi skal altså holde oss innenfor intervallet $x \in (0,1)$, med Dirichlet-grensebetingelser gitt ved $u(0) = u(1) = 0$. Vi definerer den diskrete tilnærmingen til løsningen u med v_i, slik at $x_i = ih$, hvor $h = 1/(n+1)$ er steglengden. Vi får da at $x_0 = 0$ og $x_{n+1} = 1$. Den andrederiverte av u kan da tilnærmes med en tre-punkts formel

$$-\frac{v_{i+1} + v_{i-1} - 2v_i}{h^2} = f_i \quad \text{når} \quad i = 1, ..., n \tag{7}$$

hvor $f_i = f(x_i)$ og grensebetingelsene er gitt som $v_0 = v_{n+1} = 0$. Feilleddet her går som $O(h^2)$.

Dersom vi multipliserer leddet h^2 i ligning (7) på hver side, kan vi definere leddet på høyre side av ligningen som $\tilde{b_i} = h^2 f_i$. Vi ender altså opp med

$$-v_{i+1} - v_{i-1} + 2v_i = \tilde{b_i} \tag{8}$$

Dersom vi setter inn ulike verdier av i i ligningen ovenfor, ender vi opp med ligninger på formen:

$$i = 1: \quad v_2 + v_0 - 2v_1 = \tilde{b_1}$$

$$i = 2: \quad v_3 + v_1 - 2v_2 = \tilde{b_2}$$

osv. Vi ender altså opp med et lineært ligningssett som kan settes opp som en matriseligning:

$$\begin{pmatrix} 2 & -1 & 0 & \dots & \dots & 0 \\ -1 & 2 & -1 & 0 & \dots & \dots \\ 0 & -1 & 2 & -1 & 0 & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & & -1 & 2 & -1 \\ 0 & \dots & & 0 & -1 & 2 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix} \tag{9}$$

hvor A er en tridiagonal nxn-matrise. Hvis vi kaller vektorene

$$\begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \mathbf{v} \quad \text{og} \quad \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix} = \tilde{\mathbf{b}} \tag{10}$$

kan vi på forkortet form skrive ligningen som

$$\mathbf{A}\mathbf{v} = \tilde{\mathbf{b}} \tag{11}$$

Her er $\mathbf{A}$ og $\tilde{\mathbf{b}}$ kjent, mens vektoren $\mathbf{v}$ er den ukjente.

I vårt tilfelle er funksjonen f gitt som $f(x) = 100e^{-10x}$. Dersom vi bruker dette i ligning (5) og integrerer denne ligningen analytisk, ender vi opp med en løsning på formen

$$u(x) = 1 - (1 - e^{-10})x - e^{-10x} \tag{12}$$

Det er denne analytiske løsningen vi skal sammenligne den numeriske løsningen med.

3 Løsningsmetoder

3.1 Gauss-eliminasjon

Generell tridiagonal matrise

Vi kan tenke oss at elementene langs diagonalen i matrisen vår, A, utgjør en vektor b, samtidig som over -og underdiagonalene utgjør vektorene a og c. Alle andre elementer i matrisen er null. Vi antar nå at elementene i hver vektor a, b og c *ikke* er identiske. Matriseligningen kan da skrives

$$\begin{pmatrix} b_1 & c_1 & 0 & \dots & \dots & \dots \\ a_2 & b_2 & c_2 & \dots & \dots & \dots \\ & a_3 & b_3 & c_3 & \dots & \dots \\ & \dots & \dots & \dots & \dots & \dots \\ & & & a_{n-2} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ \dots \\ \dots \\ \dots \\ v_n \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \dots \\ \dots \\ \dots \\ \tilde{b}_n \end{pmatrix} \tag{13}$$

hvor vektorene a, b og c har lengden n. Hvis vi nå tenker oss at n=4, forenkler denne matriseligningen seg noe:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 \\ 0 & a_3 & b_3 & c_3 \\ 0 & 0 & a_4 & b_4 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} \tilde{b}_1 \\ \tilde{b}_2 \\ \tilde{b}_3 \\ \tilde{b}_4 \end{pmatrix}. \tag{14}$$

Vi vil da få ligninger på formen

$$\begin{gathered} b_1 v_1 + c_1 v_2 = \tilde{b}_1 \\ a_2 v_1 + b_2 v_2 + c_2 v_3 = \tilde{b}_2 \\ a_3 v_2 + b_3 v_3 + c_3 v_4 = \tilde{b}_3 \\ a_4 v_3 + b_4 v_4 = \tilde{b}_4 \\ \vdots \\ a_i v_{i-1} + b_i v_i + c_i v_{i+1} = \tilde{b}_i \quad \text{når} \quad i = 1, 2, ..., n \end{gathered} \tag{15}$$

For å kunne løse dette ligningssettet, bruker vi metoden *forlengs substitusjon*. Det vi ønsker i første omgang er å få element a_1 til å bli null. Vi starter med å multiplisere første rad i matrisen med a_2/b_1. Deretter trekker vi første rad fra andre rad. Skriver vi matrisen på utvidet form vil vi få:

$$\left[\begin{array}{cccc|c} 1 & c_1/b_1 & 0 & 0 & \tilde{b}_1/b_1 \\ 0 & b_2 - \frac{c_1}{b_1}a_2 & c_2 & 0 & \tilde{b_2} - \frac{\tilde{b_1}}{b_1}a_2 \\ 0 & a_3 & b_3 & c_3 & \tilde{b_3} \\ 0 & 0 & a_4 & b_4 & \tilde{b_4} \end{array}\right]$$

Nå som første element i andre rad har blitt redusert til 0, kan vi gå i gang med å redusere andre element i tredje rad. La oss først, for ordens skyld, definere $\tilde{d_1} = b_1$, $\tilde{d_2} = b_2 - \frac{c_1}{b_1}a_2$, $\tilde{e_1} = \tilde{b_1}/\tilde{d_1}$ og $\tilde{e_2} = (\tilde{b_2} - \frac{\tilde{b_1}}{b_1}a_2)/\tilde{d_2}$. Vi kan nå multiplisere rad nummer to med $1/\tilde{d_2}$ og døpe om variablene til noe mer oversiktelig:

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d_1} & 0 & 0 & \tilde{b}_1/\tilde{d_1} \\ 0 & 1 & c_2/\tilde{d_2} & 0 & (\tilde{b_2} - \frac{\tilde{b_1}}{b_1}a_2)/\tilde{d_2} \\ 0 & a_3 & b_3 & c_3 & \tilde{b_3} \\ 0 & 0 & a_4 & b_4 & \tilde{b_4} \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & c_1/b_1 & 0 & 0 & \tilde{e_1} \\ 0 & 1 & c_2/\tilde{d_2} & 0 & \tilde{e_2} \\ 0 & a_3 & b_3 & c_3 & \tilde{b_3} \\ 0 & 0 & a_4 & b_4 & \tilde{b_4} \end{array}\right]$$

Trekker nå a_3·(rad to) fra rad tre:

$$\left[\begin{array}{cccc|c} 1 & c_1/\tilde{d_1} & 0 & 0 & \tilde{e_1} \\ 0 & 1 & c_2/\tilde{d_2} & 0 & \tilde{e_2} \\ 0 & 0 & b_3 - \frac{c_2}{\tilde{d_2}}a_3 & c_3 & \tilde{b_3} - \tilde{e_2}a_3 \\ 0 & 0 & a_4 & b_4 & \tilde{b_4} \end{array}\right] \sim \left[\begin{array}{cccc|c} 1 & c_1/\tilde{d_1} & 0 & 0 & \tilde{e_1} \\ 0 & 1 & c_2/\tilde{d_2} & 0 & \tilde{e_2} \\ 0 & 0 & 1 & c_3/\tilde{d_3} & (\tilde{b_3} - \tilde{e_2}a_3)/\tilde{d_3} \\ 0 & 0 & a_4 & b_4 & \tilde{b_4} \end{array}\right]$$

I den siste overgangen har vi multiplisert tredje rad med $1/\tilde{d}_3$. Vi ser et gjentakende mønster hvor

$$\tilde{d}_i = b_i - \frac{c_{i-1}}{\tilde{d}_{i-1}} a_i \quad \text{og} \quad \tilde{e}_i = \frac{\tilde{b}_i - e_{i-1}\tilde{} a_i}{\tilde{d}_i} \quad \text{hvor} \quad i = 2, 3, ..., n \tag{16}$$

med initialbetingelser $\tilde{d}_1 = b_1$ og $\tilde{e}_1 = \tilde{b}_1/\tilde{d}_1$. Vi har nå utført en forlengs substitusjon. For å finne den endelinge (diskrete) løsningen $\mathbf{v}$, bruker vi det vi kaller *baklengs substitusjon*. Vi starter med å se på det endelinge resultatet etter at alle pivotelementene har blitt radredusert til ledende enere:

$$\begin{pmatrix} 1 & c_1/\tilde{d}_1 & 0 & 0 \\ 0 & 1 & c_2/\tilde{d}_2 & 0 \\ 0 & 0 & 1 & c_3/\tilde{d}_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{pmatrix} = \begin{pmatrix} \tilde{e}_1 \\ \tilde{e}_2 \\ \tilde{e}_3 \\ \tilde{e}_4 \end{pmatrix} \tag{17}$$

Skriver vi ut ligningene får vi at

$$v_4 = \tilde{e_4}$$

$$v_3 + (c_3/\tilde{d}_3)v_4 = \tilde{e_3}$$

$$v_2 + (c_2/\tilde{d}_2)v_3 = \tilde{e_2}$$

$$v_1 + (c_1/\tilde{d}_1)v_2 = \tilde{e_1}$$

Den endelige algoritmen for å finne v_i blir derfor på formen

$$v_i = \tilde{e_i} - (c_i/\tilde{d}_i)v_{i+1} \quad \text{når} \quad i = n-1, n-2, ..., 1 \tag{18}$$

og $v_n = \tilde{e_n}$ når $i = n$. Figur 1 viser hvordan disse algoritmene kan implementeres inn i C++:

```
// Algorithm for finding v:
// a(i)*v(i-1) + b(i)*v(i) + c(i)*v(i+1) = b_twidd(i)
// Row reduction; forward substitution:
double b_temp = b[1];
v[1] = b_twidd[1]/b_temp;
for (int i=2;i<=n;i++) {
   // Temporary value needed also in next loop:
   diag_temp[i] = c[i-1]/b_temp;
   // Temporary diagonal element:
   b_temp = b[i] - a[i]*diag_temp[i];
   // Updating right hand side of matrix equation:
   v[i] = (b_twidd[i]-v[i-1]*a[i])/b_temp;
}

// Row reduction; backward substition:
for (int i=n-1;i>=1;i--) {
    v[i] -= diag_temp[i+1]*v[i+1];
}
```

Figure 1: Figuren viser hvordan forlengs -og baklengs substitusjon kan implementeres i C++.

Vi ser at forlengs substitusjons-algoritmen har 6 FLOPS, mens baklengs substitusjon har 2 FLOPS. Vi ender altså opp med at

$$N_{general} = O(8n) \tag{19}$$

Spesialtilfelle
I vårt tilfelle har elementene langs de tre diagonalene identisk verdi. Det vil si; alle elementene $a_i = c_i =$ -1, og alle elementene langs diagonalen $b_i = 2$. Setter vi disse tallene inn i algoritmen vår, får vi for forlengs substitusjon at

$$\tilde{d}_i = 2 - \frac{1}{\tilde{d}_i} = \frac{i+1}{i} \quad \text{og} \quad \tilde{e}_i = \frac{\tilde{b}_i + \tilde{e}_{i-1}}{\tilde{d}_{i-1}} \tag{20}$$

når $i = 2, 3, ..., n$, mens baklengs substitusjon forenkles til

$$v_i = \tilde{e}_i + \frac{v_{i+1}}{\tilde{d}_i} \tag{21}$$

når $i =$ n-1, n-2, ..., 1. Figur 2 viser hvordan denne forenklede algoritmen kan implementeres i C++.

```
//special case
double *b_tilde = new double [n+2];
double *f_tilde = new double [n+2];
b_tilde[1] = b[1]; //=2
f_tilde[1] = f_[1];

//precalculate updated diagonal elements
b_tilde[0] = b_tilde[n] = 2;
u_num[0] = u_num[n] = 0.0;
for (int i = 1; i < n; i++){
    b_tilde[i] = (i+1.0)/((double) i);
}

//forward substitution
for (int i=2; i<n; i++) {
    f_tilde[i] = f_[i] + f_tilde[i-1]/b_tilde[i-1];
}

//backward substitution
u_num[n-1] = f_tilde[n-1]/b_tilde[n-1];
for (int i = n-2; i > 0; i--) {
    u_num[i] = (f_tilde[i] + u_num[i+1])/b_tilde[i];
}
```

Figure 2: Forlengs -og baklengs substitusjon når matrisen har identiske elementer langs diagonalen, over diagonalen og under diagonalen.

Vi ser nå at antall FLOPS har blitt redusert til 6, slik at

$$N_{special} = O(6n) \tag{22}$$

3.2 LU-faktorisering

Det finnes en annen måte å løse det lineære ligningssettet vårt på. En matrise $A \in \Re^{nxn}$ kan LU-faktoriseres dersom dens determinant ikke er lik null. Dersom denne LU-faktoriseringen eksisterer, og A er ikke-singulær (det vi si; kvadratisk og med determinant $\neq 0$), så er dette en *unik* LU-faktorisering for A. Determinanten kan da gis ved

$$det\{\mathbf{A}\} = det\{\mathbf{LU}\} = det\{\mathbf{L}\}det\{\mathbf{U}\} = u_{11}u_{22}...u_{nn} \tag{23}$$

Siden disse kriteriene er oppfylt for vår matrise A, så eksisterer det en unik LU-dekomposisjon for A. Det betyr at matrisen A i ligningen $A\mathbf{v} = \tilde{\mathbf{b}}$ kan dekomponeres til et produkt av to matriser, nemlig L og U, hvor L er en nedre triangulær matrise og U er en øvre triangulær matrise: $A = LU$

$$A = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots & \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & & \vdots & \\ 0 & 0 & 0 & \dots & u_{nn} \end{pmatrix} \tag{24}$$

LU-faktorisering er en vanlig måte å løse ligningssett på numerisk. For eksempel kan ligningssettet

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4 = w_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = w_2$$

$$a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 = w_3$$

$$a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 = w_4$$

skrives som en matriseligning på formen

$$\mathbf{Ax = w}$$

hvor A og w er kjente størrelser mens x er den ukjente. Ved å bruke LU-faktorisering kan vi skrive

$$\mathbf{Ax \equiv LUx = w}$$

Denne ligningen kan splittes i to;

$$\mathbf{Ly = w}; \quad \text{og} \quad \mathbf{Ux = y}$$

Determinanten til L vil være lik 1, siden alle elementene på denne triangulære matrisen har verdien 1. Det betyr at L er en invertibel matrise, slik at vi kan bruke

$$\mathbf{Ux = L^{-1}w = y}$$

slik at

$$\mathbf{L^{-1}w = y}$$

Dersom vi finner $\mathbf{y}$ først, kan vi finne $\mathbf{x}$ ved ligningen $\mathbf{Ux = y}$. For en 4x4-matrise vil ligningene komme på formen

$$y_1 = w_1$$
$$l_{21}y_1 + y_2 = w_2$$
$$l_{31}y_1 + l_{32}y_2 + y_3 = w_3$$
$$l_{41}y_1 + l_{42}y_2 + l_{43}y_3 + y_4 = w_4$$

og

$$u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + u_{14}x_4 = y_1$$
$$u_{22}x_2 + u_{23}x_3 + u_{24}x_4 = y_2$$
$$u_{33}x_3 + u_{34}x_4 = y_3$$

$$u_{44}x_4 = y_4$$

Vi kan nå overføre dette til vår problemstilling. Dersom vi setter inn at $A = LU$ i ligningen $A\mathbf{v} = \tilde{\mathbf{b}}$, slik at $L(U\mathbf{v}) = \tilde{\mathbf{b}}$, ender vi opp med to ligningssett:

$$L\mathbf{y} = \tilde{\mathbf{b}} \quad \text{og} \quad U\mathbf{v} = \mathbf{y} \tag{25}$$

Her finner vi først $\mathbf{y}$, for så å bruke $\mathbf{y}$ til å finne løsningen $\mathbf{v}$. Bibliotekene `lib.h` og `lib.cpp` inneholder funksjonene vi trenger for å utføre LU-dekomposisjonen. Funksjonen `ludcmp` finner selve LU-dekomposisjonen, mens `lubksb` utfører en baklengs substitusjon for å finne løsningen $\mathbf{v}$. Firgur 3 viser hvordan dette kan implementeres i C++:

```
//LU-decomposition of matrix A
int *index;
double d;
index = new int[n]; //keeps track of the number of interchanges of rows.


ludcmp(A,n,index,&d); //LU-decomposition
lubksb(A,n,index,b_twidd); //Solve. The solution is now in b_twidd
```

Figure 3: LU-faktorisering ved bruk av funksjonene `lubdcmp` og `lubksb`, hentet fra biblioteket `lib.cpp`.

Vi vet at denne algoritmen krever

$$N_{LU} = \frac{2}{3}n^3 + 2n^2 = O(\frac{2}{3}n^3) \tag{26}$$

antall FLOPS. Sammenligner vi dette med algoritmene fra forrige seksjon, ser vi at LU-faktorisering faktisk er mer krevende for maskinen.

4 Resultater

Figur 4, 5 og 6 viser plot av den analytiske løsningen av differensialligningen sammen med den numeriske løsningen vi fant ved hjelp av Gauss-eliminasjon. Det er den generelle algoritmen som er brukt (skrevet i C++), mens et python program tar seg av plottingen. Resultatene er plottet for $n = 10$, $n = 100$ og $n = 1000$.

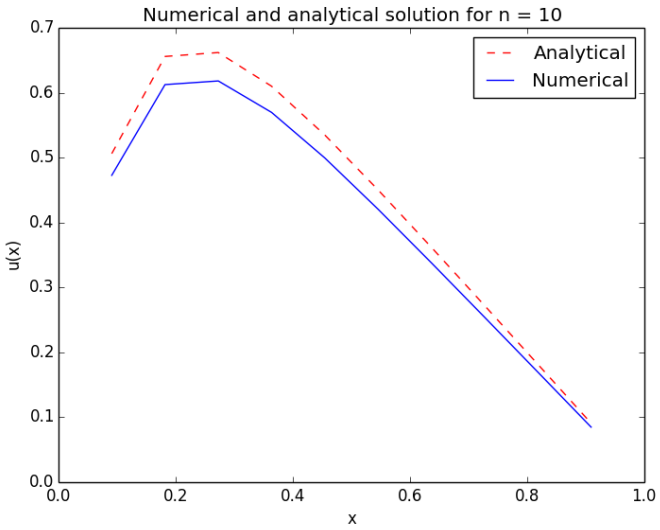

Figure 4: Analytisk og numerisk løsning for $n = 10$.

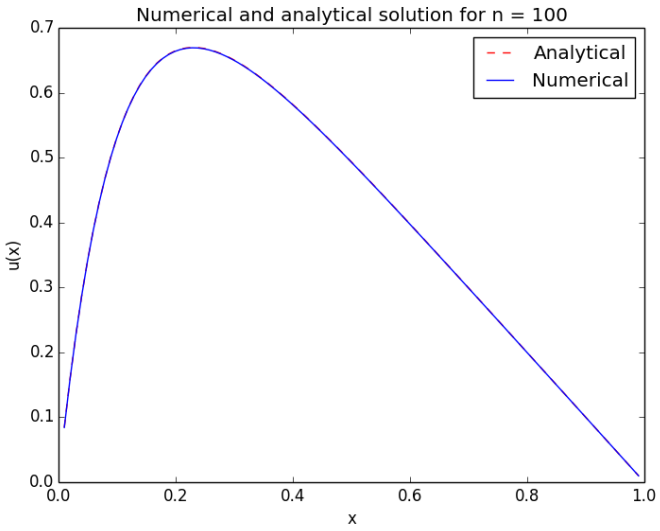

Figure 5: Analytisk og numerisk løsning for $n = 100$.

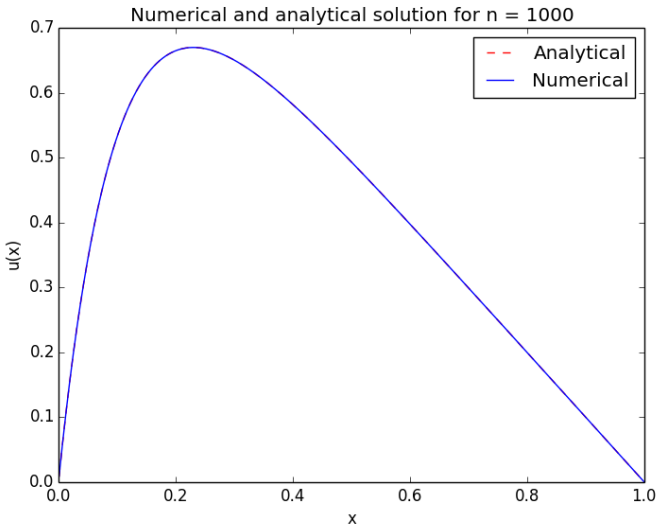

Figure 6: Analytisk og numerisk løsning for $n = 1000$.

For å finne algoritmens CPU (Central Processing Unit), brukte vi koden gitt i figur 7.

```
//Start timer
clock_t start, finish;
start = clock();

//do something here

//stop timer
finish = clock();
( (double) (finish - start)/CLOCKS_PER_SEC );
```

Figure 7: Metode for å finne et programs CPU.

Siden det nå er utviklet to algrotimer for å løse samme problemstilling, kan det være interessant å sammenligne disse algoritmenes CPU tid. Vi har sammenlignet CPU tiden mellom den generelle og den spesielle algoritmen for matriser med størrelse opp til $n = 10^7$ grid points. Resultatet vises i tabell 1.

Table 1: Sammenligning av CPU-tid for de ulike algoritmene

n	**Generell CPU [s]**	**Spesiell CPU [s]**
10	2.0000000E-06	3.0000000E-06
10^2	8.0000000E-06	5.0000000E-06
10^3	4.6000000E-05	4.6000000E-05
10^4	0.00038200000	3.1000000E-05
10^5	0.0040140000	0.00030400000
10^6	0.034464000	0.0032440000
10^7	0.37009800	0.031006000

Vi kan også finne størrelsen på den relative feilen i beregningene vi nå har gjort, ved å implementere formelen

$$\epsilon_i = log_{10}\left(\left|\frac{v_i - u_i}{u_i}\right|\right) \tag{27}$$

hvor $i = 1, 2, ..., n$. ϵ_i er en funksjon av $log_{10}(h)$, siden verdiene v_i og u_i avhenger av steglengden h. Vi finner den maksimale relative feilen for ulike verdier av n ved å kjøre programmet *error.cpp*, med input-verdiene "n" = 10, "number of steps" = 7 og "step increase" = 10. På denne måten loopet programmet gjennom 7 ganger, og kalkulerte den relative feilen med en økning på x10 for hver gang, helt opp til $n = 10^7$. Tabell 2 viser resultatene.

Table 2: Relativ feil i den generelle algoritmen

n	$\mathbf{log_{10}}(h)$	$\mathbf{log_{10}}(\epsilon_{max})$
10	-1.0413927	-1.3589135
10^2	-2.0043214	-3.2621439
10^3	-3.0004341	-5.2541384
10^4	-4.0000434	-7.2533984
10^5	-5.0000043	-9.0436161
10^6	-6.0000004	-6.3508347
10^7	-7.0000000	-6.1988961

Til slutt sammenlignes CPU-tiden mellom en standard LU-faktorisering og den generelle tridiagonale algoritmen. Dette ble gjort for $n = 10$, $n = 100$ og $n = 1000$. Resultatene er vist i tabell 3.

Table 3: CPU-tiden til LU-faktoriseringen og den generelle tridiagonale løsningen.

n	**CPU [s]: LU**	**CPU [s]: Tridiagonal**
10	0.00013400000	1.0000000E-06
100	0.0018050000	5.0000000E-06
1000	2.8918500	3.4000000E-05

5 Diskusjon

Vi ser av figurene 4, 5 og 6 at den implementerte algortimen for en generell tridiagonal matrise fungerer. Som forventet vil den numeriske løsningen nærme seg den analytiske løsningen jo større n er. Dette ses særlig godt dersom man sammenligner figur 4 med de to andre figurene.

I tabell 2 ser vi at den maksimale relative feilen minker jo større n blir. Et sted i mellom $n = 10^5$ og $n = 10^6$ vil den maksimale relative feilen begynne å øke igjen, dette kommer av at steglengden h har blitt så liten at vi ender opp med avrundingsfeil. Man vil altså ikke få noen mer nøyaktig numerisk løsning enn ved $n = 10^5$.

Vi endte opp med to ulike algortimer som utførte samme jobb; en generell og en spesiell. Den generelle algoritmen kan brukes på en vilkårlig tridiagonal matrise, mens den spesielle algoritmen er en (forenklet) versjon av denne hvor vi har identiske tall langs diagonalen og i diagonalen over og under hoveddiagonalen. Den generelle algoritmen endte opp med antall FLOPS på $O(8n)$, mens den spesielle hadde $O(6n)$ FLOPS. Det vil da være rimelig å anta at den spesielle algoritmen vil ha (noe) kortere CPU tid enn den generelle. Ser vi nærmere på tabell 7, ser vi at dette stemmer. Det skal likevel nevnes at dette ikke var noen banebrytende forskjell i CPU tid. Forskjellen blir mer markant for store n-verdier.

Når det kommer til LU-faktoriseringen, ser vi tydelig forskjell når det kommer til CPU (se tabell 3). Det har tidligere blitt nevnt at algoritmen for den generelle tridiagonale matrisen krever $O(8n)$ FLOPS, mens LU-fatoriseringen krever $O(\frac{2}{3}n^3)$ FLOPS. Når n blir stor kan vi derfor forvente at CPU tiden vil øke lineært som funksjon av n for den tridiagonale løsningsmetoden, mens LU-faktoriseringen vil øke kubisk.

Siden vi i vårt tilfelle har med en tridiagonal matrise å gjøre, vil en LU-faktorisering bruke unødig antall FLOPS, for ikke å snakke om minne i datamaskinen til å lagre alle matriseelementene. Når n blir betraktelig stor (størrelsesorden 10^4 og oppover), vil vi ende opp med en gigantisk matrise. Dersom vi hadde valgt $n = 10^5$, ville vi endt opp med en matrise med $10^5 \cdot 10^5 = 10^{10}$ elementer! Alle disse elementene bruker en viss menge lagringsplass. Dersom hvert element

er av typen "double" ville vi trengt 8byte$\cdot 10^{10} = 80GB$ med lagringsplass!

6 Konklusjon

Siden vi i vårt tilfelle hadde et ligningssett som førte til en tridiagonal matrise, sier det seg selv at det vil være unødig å anvende vanlig Gauss-elinimasjon eller standard LU-dekomposisjon, med tanke på antall FLOPS. Det egnet seg derimot å anvende en egen utviklet algoritme beregnet på tridiagonale matriser!

Vi har i dette prosjektet sett at vi kan løse en én dimensjonal Poisson-ligning numerisk på flere ulike metoder. Metodene varierer i antall FLOPS og beregningstid. Vi startet med å anvende Gauss-eliminasjon på en generell matrise, og fant ut at denne metoden kunne forenkles siden matrisen var tridiagonal. Vi kunne da se bort fra alle null-elementene og heller lagre de tre diagonalene i form av vektorer. Dette ga oss et antall FLOPS på formen $O(8n)$. Vi kalkulerte så den maksimale relative feilen i den numeriske løsningen ved å sammenligne med den gitte analytiske løsningen, og fant at vi hadde minst relativ feil når $n = 10^5$.

Den andre metoden vi brukte var LU-faktorisering. Dette er en standard metode å løse ligningssett på. Ulempen med denne metoden er at den allokerer plass i minnet til hvert eneste element i matrisen, slik at for store n-verdier vil dette være en tidkrevende metode. I vårt tilfelle var ikke dette den optimale metoden, siden matrisen vår var tridiagonal.

7 Vedlegg

Alle koder og resultater som er brukt i rapporten finnes på Github-adressen: https://github.com/livewj/Project-1

References

[1] Kursets offisielle Github-side *FYS3150 - Computational Physics* https://github.com/CompPhysics/ComputationalPhysics, 03.09.2016

[2] Slides fra kursets offisielle nettside: "Matrices and linear algebra" http://compphysics.github.io/ComputationalPhysics/doc/web/course, 14.09.16

[3] https://no.wikipedia.org/wiki/CPU, 16.09.16