Oblig TMA 4106
Eirik Vatland Johansen

PS:For simuleringen har jeg brukt C++ med en pakke vi bruker i TDT4102 så vet ikke om du kan bruke dette. Men jeg kan legge til et vedlegg om hvordan vi lastet ned dette, eller om du ikke gidder det og har lyst å se animasjonene kan du si i fra til meg så kan jeg sende en video av dem.

Opp1:
h=0.01
Gir 4.50417
h=0.0001
Gir 4.48191
Med minkende h ble feilen i svaret mindre og mindre helt til h= h=0.0000000000001 og for h=0.0000000000000001 ble svaret 0.0.

Opp2:
Selv om at denne metoden var mer nøyaktig hadde den samme problem med numerisk støy da h ble for liten. For h=0.0000000000000001 ble også hær 0.0.
Feil med taylorrekker:

Opp3:
Ble litt for gira på å komme meg videre så jag slo ikke på storommen.

Opp4:
Denne var gøy! For å løse lagde jeg en kode i C++.
Info om koden:
Koden starter med å først definere konstantene som skal bli brukt for å gjøre simuleringen, som k, h, n(antall ellmernter langs x aksen) og en dekor u som holder verdien til hvert element langs x aksen til et bestemt tidspunkt.
Videre har jeg lagd en funksjon som tar inn en vektor med forrige iterasjon verdier for temperatur langs x aksen og bruker eulers eksplisitt til å finne neste iterasjon sine verdier.
Etter det måtte jeg finne en måte og lage animasjon på, jeg viste ikke om det fantes noen innebyggede funksjoner for dette, det gode det sikkert, men jeg hadde lust å lage noe selv. For å gjøre dette bruker jeg en classe som heter AnimationWindow hentet fra TDT4102. til å lage linje funksjonen. Denne tar inn en vektor med verdier for tenpratur langs x-aksen og tenger en strekk mellom verdiene som kommer etterhverandre, slik vi får en graf av varmen langs x-aksen ved et spesifikt tidspunkt.
Så i selve main blir u verkotrvn insisalsert med insialbetingelsen.
Så opprettes et vindu som skal brukes til å animasjonen.
Animasjonen foregår inni en while løkke hvor en ny vektor blir opprettet. Veksten blir først gitt randkravene, så blir den fylt med verdiene for varme ved å løse Euler eksplisitt for alle verkende utenom ved endepunktene.
Tilslutt blir grafen tegnet i vinduet. Så gjentas denne løkken til en bestemt tid har gått.

Resultat: Det var tydelig at for å få en stabil løsning måtte k være mindre enn h, men for små verdier av h ble heller ikke løsningen stabil. Jeg kunne gjøre my som jeg ville med k dersom den var tilstrekkelig mindre en h, men da ble tiden mellom hver simulering veldig lang.

Jeg vett ikke hvordan jeg kan vise deg animasjonen, men her er koden så kan du kjøre den og se om du gidder.
KODE:

```
#include "AnimationWindow.h"
#include <fstream>
#include <random>
#include <iostream>
#include "std_lib_facilities.h"

double h = 0.1;
double k = 0.001;
int n = 3.14 / h;
std::vector<double> u(n);
double t = 0;


double eksplisitt(const std::vector<double>& v, int i) {
    return ((v.at(i+1) - 2*v.at(i) + v.at(i-1)) * k / (h*h)) +
v.at(i);
}



void linje(int n, const std::vector<double>& v,
TDT4102::AnimationWindow& window) {
    double x = h;
    for (int i = 1; i < n - 1; i++) {
        TDT4102::Point lineStart {
            static_cast<int>((x - h) * 300),
            static_cast<int>((1 - v.at(i-1)) * 250)
        };
        TDT4102::Point lineEnd {
            static_cast<int>(x * 300),
            static_cast<int>((1 - v.at(i)) * 250)
        };
        TDT4102::Color lineColor = TDT4102::Color::firebrick;
        window.draw_line(lineStart, lineEnd, lineColor);
        x += h;
    }
}

int main() {

    double x = 0;
    for (int i = 0; i < n; i++) {
        u.at(i) = std::sin(x);
        x += h;
    }
```

```
    TDT4102::AnimationWindow vindu(1000, 500);

    while (t < 1) {
    std::vector<double> ut(n);
    ut.at(0) = 0;
    ut.at(n-1) = 0;

    for (int p = 1; p < n - 1; p++) {
        ut.at(p) = eksplisitt(u,  p);
    }

    linje(n, ut, vindu);
    u = ut;
    t += k;

    vindu.next_frame();
}
    return 0;
}
```

Opp5:
Denne var ikke så lett nei. Førs prøvde jeg ett par lure snarveger for å få en løsningen, som å løse for u(i+1,j+1) og si at de to første verdiene i vekotren skule være 0, men dette ga ikke noe stabilt svar. Så jeg prøvde å lese meg litt mer opp på hvordan man kan løse slike systemer og ble egentlig enda mer forvirret. Jeg kom fram til at jeg måtte sette opp et liningsystem Au(i+1)=u(i), og løse dette hver gang. Men jeg klaret ikke å komme på en måte å løse dette selv så jeg har tatt en måte og løse det på og tilpasset det min kode fra forigge oppgave og prøv å satt me godt inn i fremgangsmåten.
KODE:

```
#include "AnimationWindow.h"
#include "widgets/TextInput.h"
#include <vector>
#include <cmath>

// Thomas-algoritmen for tridiagonalt system
std::vector<double> solveTridiagonal(const std::vector<double>& a,
const std::vector<double>& b, const std::vector<double>& c, const
std::vector<double>& d) {
    int n = d.size();
    std::vector<double> cp(n), dp(n), x(n);
    cp[0] = c[0] / b[0];
    dp[0] = d[0] / b[0];

    for (int i = 1; i < n; ++i) {
        double denom = b[i] - a[i] * cp[i - 1];
        cp[i] = c[i] / denom;
        dp[i] = (d[i] - a[i] * dp[i - 1]) / denom;
    }

    x[n - 1] = dp[n - 1];
    for (int i = n - 2; i >= 0; --i) {
        x[i] = dp[i] - cp[i] * x[i + 1];
```

```
    }
    return x;
}

int main() {
    const int Nx = 100;
    const double L = 1.0;
    const double dx = L / (Nx - 1);
    const double dt = 0.005;

    const double alpha = dt / (dx * dx);
    const int numSteps = 5000;

    std::vector<double> u(Nx, 0.0);
    for (int i = 0; i < Nx; ++i) {
        double x = i * dx;
        u[i] = sin(M_PI * x);
    }

    std::vector<double> a(Nx, -alpha), b(Nx, 1 + 2 * alpha), c(Nx,
-alpha);
    a[0] = 0; c[Nx - 1] = 0;
    b[0] = 1; b[Nx - 1] = 1;
    a[Nx - 1] = 0; c[0] = 0;

    // Lag vindu
    TDT4102::AnimationWindow win{1000, 400};
    const int scale = 10;

    for (int step = 0; step < numSteps; ++step) {

        u = solveTridiagonal(a, b, c, u);


        for (int i = 1; i < Nx; ++i) {
            int x0 = (i - 1) * scale;
            int x1 = i * scale;
            int y0 = 200 - static_cast<int>(u[i - 1] * 100);
            int y1 = 200 - static_cast<int>(u[i] * 100);
            win.draw_line({x0, y0}, {x1, y1},
TDT4102::Color::red);
        }

        win.next_frame();
    }

    return 0;
}
```

Fremgangsmåte:
solveTridiagonal funksjonen kjøre Thomas-algoritmen. Dette er en algrittemp for å løse tidiagonale matriser alså at setntral diagonalen samrt den over og under den har verdier og alle andre er tomme. Denne brukes til å løse likningsettet ved å ta inn tre vektorer med kofisentene for likningsettet og en vektor u med de forrige verdiene for tempraur.
Så blir de nyeveridene regnet ut og plottet på samme vis som før.

Resultat:
Denne metoden er klart mer stabil en forrige, jeg kunne for det meste bruke de verdiene jeg ville og var mest begrenset av programmkjøretid.

Opp6:
Crank-Nicolson metoden bygger på samme proinsitt som Euler implisitt, men her blir likningsettet: Au(i+1)=Bu(i).
Så denne fikk jeg og hjelp til å implementere. Men fremgangsmåten er veldig lik Eulier implisitt.
Koden under simulerer alle de tre metodene i samme vidusammen med en analytisk løsning av varmelikningen.
Resultater:
Nå er det veldig tydelig hor mye mer stabile de andre metodene er enn Euler eksplisitt, men for verdier av h og k som Euler eksplisitt er stabil på blir det ikke noen tydelig foskjell mellom simuleringene, i tillegg til at de krever mer rekne tid og minne. Så dette tyder på at den ekstra jobben med å sette opp de andre løsningene ikke alltid er vert det. Jeg prøvde å endre på inisialbetigelsen for å se om det ble større foskjell da, men alle løsningene fulgte den analytiske løsningen bra. Gjerne prøv ut koden litt selv. PS om du ender på initialbetigelsen husk og endre utrykke til den analytiske løsningen.
KODE:

```cpp
#include "AnimationWindow.h"
#include <vector>
#include <cmath>

// Thomas-algoritmen
std::vector<double> solveTridiagonal(const std::vector<double>& a,
                                     const std::vector<double>& b,
                                     const std::vector<double>& c,
                                     const std::vector<double>& d)
{
    int n = d.size();
    std::vector<double> cp(n), dp(n), x(n);

    cp[0] = c[0] / b[0];
    dp[0] = d[0] / b[0];

    for (int i = 1; i < n; ++i) {
        double denom = b[i] - a[i] * cp[i - 1];
        cp[i] = c[i] / denom;
        dp[i] = (d[i] - a[i] * dp[i - 1]) / denom;
    }

    x[n - 1] = dp[n - 1];
    for (int i = n - 2; i >= 0; --i) {
        x[i] = dp[i] - cp[i] * x[i + 1];
    }
```

```
    return x;
}

std::vector<double> crankNicolson(const std::vector<double>& u,
double k, double h) {
    int n = u.size();
    double alpha = k / (h * h);

    std::vector<double> a(n - 2, -alpha / 2.0);
    std::vector<double> b(n - 2, 1 + alpha);
    std::vector<double> c(n - 2, -alpha / 2.0);
    std::vector<double> d(n - 2);

    for (int i = 1; i < n - 1; ++i) {
        d[i - 1] = (alpha / 2.0) * u[i - 1] + (1 - alpha) * u[i] +
(alpha / 2.0) * u[i + 1];
    }

    std::vector<double> u_inner = solveTridiagonal(a, b, c, d);
    std::vector<double> u_new(n, 0.0);
    for (int i = 1; i < n - 1; ++i) {
        u_new[i] = u_inner[i - 1];
    }
    return u_new;
}

std::vector<double> implisittEuler(const std::vector<double>& u,
double k, double h) {
    int n = u.size();
    double alpha = k / (h * h);

    std::vector<double> a(n - 2, -alpha);
    std::vector<double> b(n - 2, 1 + 2 * alpha);
    std::vector<double> c(n - 2, -alpha);
    std::vector<double> d(n - 2);

    for (int i = 1; i < n - 1; ++i) {
        d[i - 1] = u[i];
    }

    std::vector<double> u_inner = solveTridiagonal(a, b, c, d);
    std::vector<double> u_new(n, 0.0);
    for (int i = 1; i < n - 1; ++i) {
        u_new[i] = u_inner[i - 1];
    }
    return u_new;
}
```

```
void tegnKurve(const std::vector<double>& u,
TDT4102::AnimationWindow& win, TDT4102::Color farge, double h) {
    for (int i = 1; i < u.size(); ++i) {
        TDT4102::Point p1{static_cast<int>((i - 1) * 300 * h),
static_cast<int>((1 - u[i - 1]) * 250)};
        TDT4102::Point p2{static_cast<int>(i * 300 * h),
static_cast<int>((1 - u[i]) * 250)};
        win.draw_line(p1, p2, farge);
    }
}
```

```
std::vector<double> analytisk(double t, double h, int n) {
    std::vector<double> u(n);
    for (int i = 0; i < n; ++i) {
        double x = i * h;
        u[i] = std::exp(-4*t) * std::sin(2*x);
    }
    return u;
}
```

```
double eksplisitt(const std::vector<double>& v, int i,double k,
double h) {
    return ((v.at(i+1) - 2*v.at(i) + v.at(i-1)) * k / (h*h)) +
v.at(i);
}
void linje(int n, const std::vector<double>& v,
TDT4102::AnimationWindow& window,double h) {
    double x = h;
    for (int i = 1; i < n - 1; i++) {
        TDT4102::Point lineStart {
            static_cast<int>((x - h) * 300),
            static_cast<int>((1 - v.at(i-1)) * 250)
        };
        TDT4102::Point lineEnd {
            static_cast<int>(x * 300),
            static_cast<int>((1 - v.at(i)) * 250)
        };
        TDT4102::Color lineColor = TDT4102::Color::hot_pink;
        window.draw_line(lineStart, lineEnd, lineColor);
        x += h;
    }
}
void løsekesitt(std::vector<double>& u,int
n,TDT4102::AnimationWindow& vindu, double k, double h){
    std::vector<double> ut(n);
    ut.at(0) = 0;
    ut.at(n-1) = 0;
    for (int p = 1; p < n - 1; p++) {
        ut.at(p) = eksplisitt(u,p,k,h);
    }
```

```
    linje(n, ut, vindu,h);
    u = ut;
}


int main() {
    double L = 3.14;
    double h = 0.1;
    double k = 0.005;
    int n = static_cast<int>(L / h) + 1;
    double totalTid = 10.0;
    double t = 0;

    std::vector<double> u_CN(n), u_IE(n),u_E(n);
    for (int i = 0; i < n; ++i) {
        double x = i * h;
        u_CN[i] = std::sin(2*x);
        u_IE[i] = std::sin(2*x);
        u_E[i]=std::sin(2*x);
    }

    TDT4102::AnimationWindow vindu(1000, 500);

    while (t < totalTid) {
        // Oppdater
        u_CN = crankNicolson(u_CN, k, h);
        u_IE = implisittEuler(u_IE, k, h);
        std::vector<double> u_exact = analytisk(t, h, n);
        løsekesitt(u_E,n,vindu,k, h);

        // Tegn
        tegnKurve(u_CN, vindu, TDT4102::Color::green, h);
        tegnKurve(u_IE, vindu, TDT4102::Color::red, h);
        tegnKurve(u_exact, vindu, TDT4102::Color::blue, h);

        t += k;
        vindu.next_frame();
    }

    return 0;
}
```