Creating Visual Effects by Solving Partial Differential Equations in Real-Time

Kasper Ornstein Mecklenburg

Bachelor's thesis
2014:K4

Lund University

Faculty of Engineering
Centre for Mathematical Sciences
Numerical Analysis

Abstract

The goal of this bachelor thesis is to create visual effects by solving partial differential equations (PDEs). The visual effects should be dependent on the input from music and the PDEs are solved in real-time. Creating visual effects can be done in many different ways and the reason for using PDEs is that they model natural phenomena which is appealing to the human eye. Focus will be on studying three different finite difference methods and one spectral method. Fast implementations are essential and by using multigrid techniques as well as the fast Fourier transform this can be achieved. In order to determine which method is suitable, we compare execution times as well as accuracy in time and space for different initial values. The final conclusion is that the spectral method is superior for the heat equation which was studied in the report.

Contents

1 Populärvetenskaplig sammanfattning

Klubbar och nattliv förändras med tiden och på senare dar har trenden gått mot att man använder sig mer och mer av visuella effekter för att förhöja klubbupplevelsen. Det vanligaste förekommande man ser på klubbar är discokulor, strobes, färgade lampor, lasrar, med mera. En del klubbar har tagit steget längre och projicerar olika färdiga filmklipp, mönster och liknande. Det är här idén om att skapa visuella effekter genom att lösa olika fysikaliska fenomen (partiella differential ekvationer) föddes.

När man gör beräkningar på till exempel flöde, värmeväxling eller vibrationer i realtid så uppkommer väldigt iögonfallande mönster och former. Anledning till att det är behagligt att titta på beror på att man modellerar naturliga rörelser som rör sig på ett tilltalade vis. För att skapa dessa effekter måste tunga beräkningar utföras och det är först på senare år som datorer varit tillräckligt kraftiga för att klara av detta inom rimliga tidsramar. Detta sätt att modellera på finns inte kommersiellt, dock så finns det liknande effekter.

Winamp är ett program som spelar media, framför allt musik, och i detta finns så man kan sätta igång ett fönster som visar visuella effekter. De mönster som uppkommer är direkt kopplade till frekvenserna i musiken och följer på så vis takten i låten som spelas. Det är här min idé skiljer sig från redan existerande visuella effekter. Effekterna kommer att följa musiken men inte vara vara kopplat de till de frekvenser som finns i ljudet. Beräkningarna kommer att börja i takt till musiken och sedan få pågå tills dess att användaren bestämmer sig för att sedan förändra utseendet på bilden. Det nya utseendet kommer att starta i takt till musiken.

För att kunna lösa och beräkna de fysikaliska problemen så behövs det specifika metoder och det finns ett antal olika metoder man kan implementera. Ekvationen som använts för styftet är värmeekvationen med periodiska randvillkor, den är relativt simpel och behändig. De metoder som använts och jämförts är multigridmetoder, explicit Euler och en spektralmetod. De har alla olika egenskaper och är bra på olika saker. Metoderna har även tidstegningmetoder vilka är nödvändiga för att se hur utvecklingen sker över tid. Det är denna utveckling över tid som är de visuella effekterna. För att kort sammanfatta metoderna:
Multigridmetoder approximerar en lösning till det rätta svaret genom att iterera flera gånger och i varje iteration så förbättras gissningen till dess att den kovergerar mot det korrekt svaret. Två tidsstegningsmetoder har använts, implicit Euler och Trapezoidal metoderna. Implicit Euler är lite snabbare medan Trapezoidal är nogrannare.
Explicit Euler är snabb och simpel. Den är dock väldigt begränsad på så vis att man måste ta små tidssteg vilket leder till att animationen blir väldigt långsam.
Spektralmetoder bygger på att transformerar från rumsplanet till frekvensplanet genom att approximera med trigonometriska funktioner. Teorin bakom är rätt omfattande och det finns specialvillkor man måste ta hänsyn till. För att snabba upp tidsåtgången används snabb fouriertransform. Metoden är exakt i tiden och är nogrann i rummet.

De egenskaper metoderna besitter har jämförts däribland nogrannhet, tidsåtgång och hur lätta de är att implementera. Spektralmetoden är rätt överlägsen i flera avseenden så som i nogrannhet och tidsåtgång. För syftet av arbeter har spektralmetoden implementerats i det slutgiltiga programmet. För att koppla ihop lösaren med takten skrevs ett program som genererade en takt och via en lokal server som skapades på datorn kunde lösaren och takten sykroniseras. Slutresultat är ett fungerade program.

Programmet i sin helhet har potential att utvecklas till att bli mer visuellt tilltalande och kopplas ihop till musik och inte till en taktgenerator. Syftet med kandidatarbetet har uppnåtts och det finns möjlighet att forsätta arbetet. Vägen innan man har ett användarvänligt program är lång men allting har en början.

2 Introduction

Solving partial differential equations (PDE) can be done with many different methods all with different properties and qualities. We will be investigating how to solve $u_t = u_{xx}$ commonly called the heat equation in one and two dimensions. The reason for choosing this PDE is that it is fairly simple and easy to implement. The methods we consider include explicit Euler, implicit Euler, the Trapezoidal method and a spectral method. Their accuracy and mainly their speed will be compared and discussed in order to find the most suitable method. So how come we need a fast method to solve PDEs?

More or less all clubs nowadays have a Disc-Jockey (DJ) who plays and mixes music live to the crowd. Lights, lasers, strobes and smoke machines are some of the visual effects in the club. Some larger clubs have huge screens where a Video-Jockey (VJ) shows visual effect; cool patterns, funky characters etc. However these visual effects are almost always playback, they have made them in their studio prior. This is where solving PDEs in real-time comes into the picture.

By solving PDEs in real-time a VJ could create visual effects interactively with the DJ and connect them to the beat to the music. The visual effects would not follow the beat entirely as the charm with PDEs is that the solution is often not known. This could of course be done without solving PDEs however as PDEs describe physical phenomenas they are natural-looking and therefore appealing to the eye.

The final step of the report is to have working program that will connect a beat with a PDE solver. The program will not have a GUI nor be user friendly however it will give a picture of how a potential product might look like.

Briefly the structure in this report will start with theory, then computations followed by implementations and finally the conclusions that have been drawn. In Section 3 PDEs will be explained and the heat equation will be introduced. Section 4 demonstrates how to discretize the problem using finite difference methods, both in spatial and temporal domain.

Section 5 is about multigrid methods using the Jacobi method and in Section 6 the discrete Fourier transform is derived from Fourier series. These two sections are the theory necessary to implement the PDE solvers in Python and the code for these is in Section 7.

When the code is written the different methods will be compared in Section 8. There will be a continuous discussion of the results, whether or not they follow the theory. The actual program which connects everything together will be presented and explained in Section 9. Finally, conclusions will be drawn and further work and improvements will be discussed in Section 10.

3 Partial differential equations

A partial differential equation (PDE) is a deterministic relationship between a multi-variable function and it's partial derivatives. It can describe different phenomena in physics, i.e. flow of heat, wave propagation, etc. The main focus in this thesis will be the heat equation, however the methods later presented in this report can be applied to other PDEs as well.

3.1 The heat equation

The heat equation is a second order parabolic PDE describing how heat is distributed over a domain as time progresses. The heat equation is defined by

$$\begin{aligned} \frac{\partial u(t,\mathbf{x})}{\partial t} &= \alpha\Delta u(t,\mathbf{x}),\ t>0,\ \mathbf{x}\in\Omega, \\ u(0,\mathbf{x}) &= g(\mathbf{x}). \end{aligned} \tag{1}$$

Here, α is a constant and $\alpha \geq 0$, Δ is the Laplace operator, $g(\mathbf{x}) \in C^2$ is the initial condition at $t = 0$ and Ω is the domain of interest. $u(t,\mathbf{x})$ will most often represent the temperature distribution in Ω. In one dimension $\mathbf{x} = (x)$ and in two dimensions $\mathbf{x} = (x, y)$. For simplicity, we consider only square domains, i.e. $\Omega = (0, c)$ in one dimension and $\Omega = (0, c)^2$ in two dimensions where c is the domain length.
Along the boundary, $\partial\Omega$, certain boundary conditions are required. Some common alternatives include Dirichlet, Neumann or mixes of these. We will be using periodic boundary conditions which in one dimension are defined as

$$u(t, x + c) = u(t, x),\ \forall x \in \Omega,\ c \in \mathbb{Z}. \tag{2}$$

In practice this can be interpreted as if the solution $u(t, x)$ wraps around the edges of the domain.

4 Finite difference methods

Finite difference methods are used to approximate PDEs numerically. They use finite difference quotients to approximate the derivatives. In order to approximate the solution $u(t,\mathbf{x})$ we wish to discretize the domain $(0,T)\times\Omega$ with a finite number of grid points $(t_n, \mathbf{x}_{i,j})$ given by

$$t_n = n\Delta t,\ x_i = i\Delta x,\ y_j = j\Delta y, \quad i, j = 0, 1, ..., N-1.$$

Here n is the number of time steps, Δt is the temporal step size and $\Delta x = \Delta y$ is the distance between two nodes defined by $\Delta x = \frac{1}{N-1}$ where N is the number of spatial nodes. We will denote the approximation to $u(t_n, \mathbf{x}_{i,j})$ by $u^n_{i,j}$.

4.1 Spatial discretization

A standard second-order central finite difference approximation to the Laplacian will be used. In one dimension, this is given by

$$\Delta u(t_n, x_i) \approx \frac{u^n_{i-1} - 2u^n_i + u^n_{i+1}}{\Delta x^2} \tag{3}$$

and in two dimensions,

$$\Delta u(t_,\mathbf{x}_{i,j}) \approx \frac{u^n_{i-1,j} + u^n_{i,j-1} - 4u^n_{i,j} + u^n_{i+1,j} + u^n_{i,j+1}}{\Delta x^2}.$$

The discretization can also be represented in matrix form. In one dimension the matrix is of size $(N \times N)$ and given by

$$T_{\Delta x}\mathbf{u} = \frac{1}{\Delta x^2}\begin{bmatrix} -2 & 1 & 0 & \dots & 0 & 1 \\ 1 & -2 & 1 & & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & 0 \\ 0 & & & 1 & -2 & 1 \\ 1 & 0 & \dots & 0 & 1 & -2 \end{bmatrix}\begin{bmatrix} u_0 \\ \\ \vdots \\ \\ \\ u_{n-1} \end{bmatrix}. \tag{4}$$

In two dimensions the matrix is $(N^2 \times N^2)$,

$$T_{\Delta xy}\mathbf{u} = \begin{bmatrix} T_{\Delta x} & I/\Delta x^2 & 0 & \dots & 0 & I/\Delta x^2 \\ I/\Delta x^2 & T_{\Delta x} & I/\Delta x^2 & & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & 0 \\ 0 & & & I/h^2 & T_{\Delta x} & I/\Delta x^2 \\ I/\Delta x^2 & 0 & \dots & 0 & I/\Delta x^2 & T_{\Delta x} \end{bmatrix}\begin{bmatrix} \mathbf{u_0} \\ \\ \vdots \\ \\ \\ \mathbf{u}_{n-1} \end{bmatrix}$$

Here I is the identity matrix and $\mathbf{u}_i = (u_{i,0}, ..., u_{i,n-1})^T$. These discretizations can also be expressed in a compact form as the computational stencils

$$\frac{1}{\Delta x^2}\begin{pmatrix} 1 & -2 & 1 \end{pmatrix} \text{ and } \frac{1}{\Delta x^2}\begin{pmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{pmatrix}. \tag{5}$$

These stencils are easily used to approximate the Laplacian by applying the stencil with a convolution over the domain. The boundary conditions (2) are discretized in one dimension by

$$u_0 = u_N \quad \text{and} \quad u_{-1} = u_{N-1}.$$

In two dimensions they are given by

$$u^n_{0,j} = u^n_{N,j} = u^n_{i,0} = u^n_{i,N} \quad \text{and} \quad u^n_{-1,j} = u^n_{N-1,j} = u^n_{i,-1} = u^n_{i,N-1}.$$

4.1.1 Error

When deriving the finite difference approximation a Taylor series has been used. In the approximation (3) some terms of higher order are left out and these terms give us the truncation error. Below $u(x_i \pm \Delta x)$ is expanded around x_i using a Taylor series.

$$u(x_i + \Delta x) = u(x_i) + \Delta x\left.\frac{\partial u}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2!}\left.\frac{\partial^2 u}{\partial x^2}\right|_{x_i} + \frac{\Delta x^3}{3!}\left.\frac{\partial^3 u}{\partial x^3}\right|_{x_i} + \frac{\Delta x^4}{4!}\left.\frac{\partial^4 u}{\partial x^4}\right|_{x_i} + ... \tag{6}$$

$$u(x_i - \Delta x) = u(x_i) - \Delta x\left.\frac{\partial u}{\partial x}\right|_{x_i} + \frac{\Delta x^2}{2!}\left.\frac{\partial^2 u}{\partial x^2}\right|_{x_i} - \frac{\Delta x^3}{3!}\left.\frac{\partial^3 u}{\partial x^3}\right|_{x_i} + \frac{\Delta x^4}{4!}\left.\frac{\partial^4 u}{\partial x^4}\right|_{x_i} + ... \tag{7}$$

Manipulation of (6) and (7) gives

$$\frac{u(x_i + \Delta x) - 2u(x_i) + u(x_i + \Delta x)}{\Delta x^2} - \left.\frac{\partial^2 u}{\partial x^2}\right|_{x_i} = \frac{2\Delta x^2}{4!}\left.\frac{\partial^4 u}{\partial x^4}\right|_{x_i} + \mathcal{O}(\Delta x^3).$$

As the terms on the right hand side are left out in our approximation (3), this is the truncation error and is $\mathcal{O}(\Delta x^2)$. In order to verify this we will compute

$$||T_{\Delta x}u - \Delta u||, \tag{8}$$

where the norm is the L^2-norm. This norm will be used here and in the following error calculations. In one dimension the L^2-norm is defined by

$$||u(x)||_{L^2}^2 = \int_\Omega u(x)^2 dx.$$

After discretization we get what is usually called the root-mean-square (RMS)-norm:

$$||\{u_i\}_0^{N-1}||_{RMS}^2 = \Delta x \sum_{i=0}^{N-1} u_i^2.$$

By combining (8) and (4.1.1) we finally get an expression which lets us compute the truncation error,

$$e_{RMS} = \sqrt{\frac{1}{N}\sum_{i=0}^{N-1} (T_{\Delta x}u - \Delta u)^2}. \tag{9}$$

The result of the error calculations for $u(x) = sin(2\pi x),\ \Omega = (0,1)$ can be seen in Figure 1. As we can see the relationship between the error and step size in the log-log graph has a slope of two proving that the error is of $\mathcal{O}(\Delta x^2)$. When the step size decreases the error will also decrease, however with a sufficiently small step size the error will start to increase. When this occurs it is due to the error being dominated by round-off errors and not by the truncation error [1, page 247].

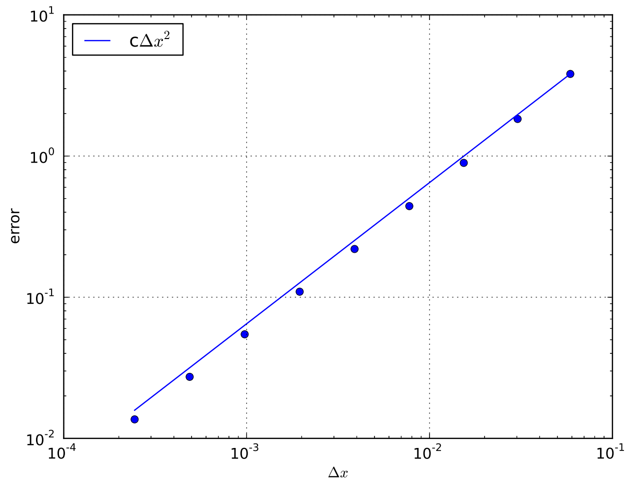


Figure 1: The log-log values of the error vs step size.

4.2 Temporal discretization

In order to see how the PDE evolves in time, time stepping methods are necessary. Three methods with different properties will be used to evaluate the PDE at certain times [2, page 120].

Explicit Euler,

$$u^{n+1} = u^n + \Delta t T_{\Delta x} u^n.$$

This is the simplest of the three methods and it is a a first order approximation, meaning that the accuracy is $\mathcal{O}(\Delta t)$.

Implicit Euler,

$$\frac{u^{n+1} - u^n}{\Delta t} = T_{\Delta x}(u^{n+1})$$

Just like the explicit Euler it is a first order approximation.
Trapezoidal method,

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{2}(T_{\Delta x}u^n + T_{\Delta x}u^{n+1}).$$

This method is more accurate and is $\mathcal{O}(\Delta t^2)$.

In the implicit Euler and the Trapezoidal method linear systems have to be solved. Solving linear systems with e.g. Gaussian elimination demands a lot of work and in order to cut down computational time a more efficient method is required. One method that can be used for this purpose is the multigrid method which will be presented in section 5.

4.2.1 Stability conditions

The methods for time stepping have different conditions they must fulfill in order to maintain stability.
CFL-condition. When solving the Laplacian the explicit Euler method must satisfy $\frac{\Delta t}{\Delta x^2} < \frac{1}{2}$ in one dimension and $\frac{\Delta t}{\Delta x^2} < \frac{1}{4}$ in two dimensions. If the CFL-condition is violated the computation becomes unstable and the solution will "blow up".
A-stability. The Trapezoidal method is A-stable which implies that the absolute stability region is the left half-plane $(z \in \mathcal{C} : Re(z) \leq 0)$, i.e. there is no restriction on Δt in our case.
L-stability. In addition to having the same properties as A-stability, a L-stable method's region of stability can cover the right half-plane. The implicit Euler is L-stable [2, page 171].

4.2.2 Error

As mentioned earlier the explicit Euler, implicit Euler and Trapezoidal method are $\mathcal{O}(\Delta t)$, $\mathcal{O}(\Delta t)$ respectively $\mathcal{O}(\Delta t^2)$ approximations. The errors for these methods are derived in the same way as in Section 4.1, but by doing a Taylor expansion in the time variable instead.

5 Multigrid methods

As mentioned in Section 4.2, solving large linear systems demands lots of work and efficient methods are necessary. Some of these methods are multigrid methods and they exist in many different forms and variations. We will concentrate on a multigrid method which uses the Jacobi method for relaxation and a multigrid V-cycle to calculate the following approximation to the Laplacian in one dimension

$$T_{\Delta x}\mathbf{u} = \mathbf{f}. \tag{10}$$

The matrix $T_{\Delta x}$ is from (4). What the Jacobi method does is to approximate a difficult problem with an easier one and this is called relaxation. By relaxing iteratively on different grid sizes, called a V-cycle, a very good approximation to the problem (10) can be made. By using fewer grid points high frequencies are reduced and this reduces noise. The approximation to $\mathbf{u}$ is $\mathbf{v}$ and with this we can start formulating our multigrid method.

5.1 Jacobi method

In order to get a picture of how good our approximation to $\mathbf{u}$ is we need to define the error as the approximation subtracted from the exact solution,

$$\mathbf{e} = \mathbf{u} - \mathbf{v}. \tag{11}$$

However, as the exact solution which we wish to calculate is unknown, the error is unknown as well. We define the residual $\mathbf{r}$, by

$$\mathbf{r} = \mathbf{f} - T_{\Delta x}\mathbf{v}.$$

Then together with (11) another important relationship can be made:

$$T_{\Delta x}\mathbf{e} = \mathbf{r}.$$

With these definitions the Jacobi method can start to take shape. Lets start by rewriting (10) as

$$\frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} = f_j \iff u_j = \frac{1}{2}(u_{j+1} + u_{j-1} - h^2 f_j). \tag{12}$$

By introducing $T_{\Delta x} = D - L - U$ the notation can be kept simple. Here L and U are strictly the lower respectively the upper triangular elements and D are the diagonal elements. We can express (12) as

$$\begin{gathered} \mathbf{u} = D^{-1}(L+U)\mathbf{u} - D^{-1}\mathbf{f} \\ \text{with } R_J = D^{-1}(L+U). \end{gathered} \tag{13}$$

Here R_J is the Jacobi iteration matrix. Lets replace $\mathbf{u}$ with $\mathbf{v}$ in (12), then the Jacobi iteration can be formulated as

$$v_j^{(1)} = \frac{1}{2}(u_{j+1}^{(0)} + u_{j-1}^{(0)} - h^2 f_j).$$

Here $\mathbf{v}^{(0)}$ is the initial guess or current approximation and $\mathbf{v}^{(1)}$ is the new calculated value [3, page 8]. Expressed with the same notation as in (13)

$$\mathbf{v}^{(1)} = R_J\mathbf{v}^{(0)} - D^{-1}\mathbf{f}.$$

5.2 Recursive V-cycle

A way to get around solving (5.1) on a large grid is by reducing the number of grid points. This will reduce the number of equations that need to be solved and in order to do this we need an operator,

$$I_h^{2h}\mathbf{v}^h = \mathbf{v}^{2h}.$$

Here I_h^{2h} is a mapping operator which reduces the number of grid points by half so that $\Omega^h \to \Omega^{2h}$ with Ω^h being the finest grid and Ω^{2h} the coarsest of the two. The operator can also interpolate points which doubles the number of grid points $I_{2h}^h : \Omega^{2h} \to \Omega^h$. In order for the reduction/interpolation to work properly the number of grid points should be chosen as $N = 2^M + 1,\ M \in \mathbb{N}$. The algorithm for the recursive V-cycle follows as [3, page 40]

$$\mathbf{v}^h \leftarrow V^h(\mathbf{v}^h, \mathbf{f}^h)$$

1. Relax m times on $T_{\Delta x}^h\mathbf{u}^h = \mathbf{f}^h$ with a given initial guess $\mathbf{v}^h$.

2. If Ω^h is the coarest grid then go to step 4.
   Else
   - $f^{2h} \leftarrow I_h^{2h}(\mathbf{f}^h - T_{\Delta x}^h \mathbf{v}^h)$,
   - $\mathbf{v}^{2h} \leftarrow 0$,
   - $\mathbf{v}^{2h} \leftarrow V^{2h}(\mathbf{v}^{2h}, \mathbf{f}^{2h}$
3. Correct $\mathbf{v}^h \leftarrow \mathbf{v}^h + I_{2h}^h \mathbf{v}^{2h}$
4. Relax m times on $T_{\Delta x}^h \mathbf{u}^h = \mathbf{f}^h$ with a given initial guess $\mathbf{v}^h$.

The cost of each V-cycle is $\mathcal{O}(N^d)$ and for a second-order method the number of iterations costs $\mathcal{O}(\log_2(N))$. Combined the total cost for each cycle is $\mathcal{O}(N^d \log_2(N))$ [4, chap. 8].

5.2.1 Improvements

In order to give an even better approximation to (10) we can solve the error residual equation (5.1) exactly to correct our approximation $\mathbf{v}$. However this is much too costly to do in every iterative step. When the coarsest grid is reached we solve

$$\mathbf{e} = T_{\Delta x}^{-1} \mathbf{r}$$

and the cost of this depends on how rough the coarsest grid is. This error calculated is removed from our approximation $\mathbf{v}$. Another improvement which that can be made is to filter out any high pitch noise a lowpass filter, it is applied on even grid points [4, chap 9]. The computational stencils for a lowpass filter can have different appearances but we will use the following in one respectively two dimensions

$$\frac{1}{4}\begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \text{ and } \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

5.3 Error

For every time we do the Jacobi iteration a better result is given. In order to specify how fast the error converges we will introduce the spectral radius of a matrix as [3, page 16]

$$\rho(A) = \max|\lambda(A)|.$$

Here $\lambda(A)$ denotes the eigenvalues of the matrix A. If A is a symmetric matrix the spectral radius of A is its 2-norm. The error after m iteration sweeps can be written as

$$\mathbf{e}^{(m)} = R^m \mathbf{e}^{(0)}.$$

If we take the vector and matrix norm we can write

$$||\mathbf{e}^{(m)}|| = ||R||^m ||\mathbf{e}^{(0)}||$$

and this converges for all initial guesses for

$$\lim_{m\to\infty} R^m = 0 \quad \text{if and only if} \quad \rho(R) < 1.$$

This error from the V-cycle will be much smaller than the spatial error derived in Section 4.1 [4, chap. 8].

6 Spectral method

An additional way to solve PDEs in an efficient way other than multigrid methods is by using spectral methods which are based on expressing the solution as a Fourier series. We will start by deriving the discrete Fourier transform from the Fourier series and then a fast Fourier transform algorithm will be used to compute it.

6.1 Fourier series

The basic idea behind the Fourier series is to express a function as a sum of simpler functions. The Fourier series uses cosine and sine functions as its basis functions, often expressed with exponential functions. The A-periodic function f is associated with the Fourier series

$$f(x) \sim \sum_{k=-\infty}^{\infty} c_k e^{\frac{-i2\pi kx}{A}} \tag{14}$$

where c_k are the Fourier coefficients, k represents the frequency and A the spatial grid length. It is the Fourier coefficients that need to be calculated in order to satisfy the equality above. To find these coefficients we use the essential orthogonality properties of the exponential functions. Lets observe the following integral

$$\int_{\frac{-A}{2}}^{\frac{A}{2}} e^{\frac{i2\pi jx}{A}} e^{\frac{-i2\pi kx}{A}} dx = \left[A \frac{e^{i2\pi(j-k)x/A}}{i2\pi(j-k)} \right]_{\frac{-A}{2}}^{\frac{A}{2}} = A\delta_{j-k}$$

where δ is the ordinary Kronecker delta and j and k are any integers. Let us assume that we can write (14) as the stronger statement,

$$f(x) = \sum_{j=-\infty}^{\infty} c_j e^{\frac{-i2\pi jx}{A}}.$$

Multiply both sides by $\frac{1}{A} e^{\frac{-i2\pi kx}{A}}$ and integrate over $\frac{-A}{2} \leq x \leq \frac{A}{2}$:

$$\begin{aligned} \frac{1}{A} \int_{\frac{-A}{2}}^{\frac{A}{2}} f(x) e^{\frac{-i2\pi kx}{A}} dx &= \frac{1}{A} \int_{\frac{-A}{2}}^{\frac{A}{2}} \sum_{j=-\infty}^{\infty} c_j e^{\frac{-i2\pi(j-k)x}{A}} dx \\ &= \sum_{j=-\infty}^{\infty} c_j \frac{1}{A} \int_{\frac{-A}{2}}^{\frac{A}{2}} e^{\frac{-i2\pi(j-k)x}{A}} dx \end{aligned}$$

Due to the orthogonality properties, the only terms remaining are when $k = j$. This leaves us with the final expression for the Fourier coefficients

$$c_k = \frac{1}{A} \int_{\frac{-A}{2}}^{\frac{A}{2}} f(x) e^{\frac{i2\pi kx}{A}} dx.$$

6.2 Derivation of the DFT

Now that the Fourier series and its orthogonality properties have been expressed we can continue with the derivation of the discrete Fourier transform approximation of the Fourier coefficients. First we need to introduce a finite grid with $N+1$ equally spaced points similar to that of Section 4. However we choose the interval $[\frac{-A}{2}, \frac{A}{2}]$ which will make more sense as we wish to approximate (6.1). We approximate the Fourier coefficients by applying the trapezoidal rule with $g(x) = f(x)e^{\frac{-i2\pi kx}{A}}$ [5, page 39]

$$c_k = \frac{1}{A}\int_{\frac{-A}{2}}^{\frac{A}{2}} g(x)dx \approx \frac{1}{A}\frac{\Delta x}{2}\left\{ g\left(\frac{-A}{2}\right) + 2\sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}-1} g(x_n) + g\left(\frac{A}{2}\right)\right\}. \tag{15}$$

As we assumed in (2), it follows that $g\left(\frac{-A}{2}\right) = g\left(\frac{A}{2}\right)$. By observing that $2\pi k x_n/A = 2\pi nk/N$ and writing $f(x_n) = f_n$ we can rewrite (15) as our final approximation for the Fourier coefficients

$$c_k \approx \frac{1}{N}\sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} f_n \, e^{\frac{-i2\pi kn}{N}} =: F_k$$

and with $k = -N/2+1, ..., N/2$. Thus, we define the DFT of the sequence $\{f_n\}$ by $\mathcal{D}\{f_n\}_k = F_k$.
In order to be able to transform back from the frequency domain to the spatial domain we need to define the inverse discrete Fourier transform, i.e. find the mapping $\mathcal{D}^{-1}$ satisfying

$$\mathcal{D}^{-1}\{\mathcal{D}\{f_n\}_k\}_n = f_n.$$

But by a simple calculation, one confirms that

$$f_n = \sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} F_k \, e^{\frac{i2\pi kn}{N}}.$$

The DFT in two dimensions is derived similarly as in one dimension. With the notation $f_{m,n} = f(x_n, y_m)$ and $w_a^b = e^{i2\pi\frac{b}{a}}$ we can write the DFT in two dimensions as

$$\begin{aligned} F_{j,k} =& \frac{1}{MN}\sum_{m=-\frac{M}{2}+1}^{\frac{M}{2}}\sum_{n=-\frac{N}{2}+1}^{\frac{N}{2}} f_{m,n}\omega_M^{-mj}\omega_N^{-nk} \\ f_{m,n} =& \sum_{j=-\frac{M}{2}+1}^{\frac{M}{2}}\sum_{k=-\frac{N}{2}+1}^{\frac{N}{2}} F_{j,k}\omega_M^{mj}\omega_N^{nk} \end{aligned}$$

The derivation of the DFT in one and two dimensions is complete and we can now consider the error of the approximation.

6.3 Errors in the DFT

This topic is quite more abstract compared to the finite difference method in Section 4. The error when using the discrete Fourier transform depends on the sampling rate which is connected to the properties of the input function. The Fourier transform approximates functions as sines and cosines so if a function is built up of these the approximation can be exact. However if the function is unsmooth, has high frequencies or discontinuities the error becomes more complex to calculate. We will now consider two cases a bit more carefully.

6.3.1 Periodic, band-limited functions

According to the Nyquist sampling theorem [5, page 183] there must be at least two samples per period or else a phenomena called aliasing will occur [5, page 185]. What happens is that if a certain frequency is sampled at a unsatisfactory rate the frequency will be perceived as a lower frequency than it actually is. If the function fulfills the Nyquist sampling rate criterion the function is said to be band-limited and if the function is also periodic we can state that $F_k = c_k$, i.e. it is no longer an approximation as in (6.2). We have an exact solution with no error if and only if the sample rate is sufficient so that all frequencies are taken in account and the input is periodic.

6.3.2 Periodic, non-band-limited functions

If the sample rate isn't sufficient (aliasing) frequencies that are greater than $N/2$ will go unnoticed. We will introduce the discrete Poisson summation formula as

$$F_k = c_k + \sum_{j=1}^{\infty}(c_{k+jN} + c_{k-jN}), \quad k = -\frac{N}{2} + 1, ..., \frac{N}{2}. \tag{16}$$

These unsampled frequencies will be the DFT error and we will start to express the error by rewriting (16) as

$$F_k - c_k = \sum_{j=1}^{\infty}(c_{k+jN} + c_{k-jN}), \quad k = -\frac{N}{2} + 1 ... \frac{N}{2}. \tag{17}$$

In order to express the error will use the theorem of rate of decay of Fourier coefficients [5, page 188]. This states that if f has p continuous derivatives, then

$$|c_k| \leq \frac{C}{|k|^{p+1}} \tag{18}$$

where C is a constant independent of k. This is a way to measure the smoothness of a function, the smoother it is the smaller the error will be. By combining (17) and (18) we thus get

$$\begin{aligned} |F_k - c_k| \leq & \sum_{j=1}^{\infty} \frac{C}{|k+jN|^{p+1}} + \frac{C}{|k-jN|^{p+1}} \\ \leq & \frac{C}{N^{p+1}} \sum_{j=1}^{\infty} \frac{1}{|j+\frac{k}{N}|^{p+1}} + \frac{1}{|j-\frac{k}{N}|^{p+1}}, \\ & k = -\frac{N}{2} + 1, ..., \frac{N}{2}. \end{aligned}$$

This series converges for $p \geq 1$ so the error can be written as

$$|F_k - c_k| \leq C'/N^{p+1}. \tag{19}$$

The larger we choose N the less the error will be as higher frequencies are taken in account and if the function is smooth the error decreases exponentially.

In order to relate the error in the Fourier coefficients to the spatial domain Parseval's theorem will be used. For the DFT it states

$$\sum_{n=0}^{N-1} |x_n|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X_k|^2$$

with $\mathcal{D}\{x_n\}_k = X_k$ and together with (4.1.1) and (19) we can formulate

$$\begin{aligned} ||\mathcal{D}^{-1}\{\mathcal{D}\{x_n u\}\} - u||_{RMS}^2 &= \frac{1}{N} \sum_{k=0}^{N-1} |F_k - c_k|^2 \\ &\leq C/N^{2p+2}. \end{aligned}$$

6.4 FFT-algorithm

The DFT can be computed in a straightforward but expensive manner. However it can easily be speeded up by avoiding calculation of numbers more than once. This is what the fast Fourier transform does and it is important to understand that it only is an efficient algorithm to compute the DFT. It uses symmetries in the calculated terms and the efficiency is highest when N is a power of two, i.e. $N = 2^M,\ M \in \mathbb{N}$ [6]. The conventional way to solve the DFT is $\mathcal{O}(N^2)$ while the FFT is $\mathcal{O}(N\ log_2 N)$ [5, page 384], a vast improvement.
There a few different ways approach the FFT; the splitting method, index expansions, matrix factorizations, prime factor and convolution models. We will concentrate on the splitting method as this is the most straightforward approach. The method has two parts; the reordering stage and the combine stage. In the reordering stage the different frequencies are ordered in a certain way to avoid them being calculated twice. In (15) we had $n = -\frac{N}{2}+1, ..., \frac{N}{2}$ but for simplicity we will choose $n = 0, 1, ..., N-1$ and the notation from (6.2) which gives

$$X_k = \sum_{n=0}^{N-1} x_n \omega_N^{-nk}.$$

We now split the sequence x_n by letting $y_n = x_{2n}$ and $z_n = x_{2n+1}$ and get

$$\begin{aligned} X_k &= \sum_{n=0}^{\frac{N}{2}-1} y_n \omega_N^{-2nk} + z_n \omega_N^{-(2n+1)k} \\ &= \sum_{n=0}^{\frac{N}{2}-1} y_n \omega_N^{-2nk} + \omega_N^{-k} \sum_{n=0}^{\frac{N}{2}-1} z_n \omega_N^{-(2n+1)k}. \end{aligned} \tag{20}$$

By splitting X_n into two DFTs with the length of $N/2$ the symmetries can start to take shape. By introducing

$$Y_k = \sum_{n=0}^{\frac{N}{2}-1} y_n \omega_N^{-2nk} \quad \text{and} \quad Z_k = \sum_{n=0}^{\frac{N}{2}-1} z_n \omega_N^{-(2n+1)k}$$

we can write (20) as

$$X_k = Y_k + \omega_N^{-k} Z_k \quad \text{and} \quad X_{k+\frac{N}{2}} = Y_k - \omega_N^{-k} Z_k.$$

Computing Y_k and Z_k costs $2\left(\frac{N}{2}\right)^2 = \frac{N^2}{2}$ and evaluating the sums is only $\mathcal{O}(N)$. Y_k and Z_k are in turn splitted and their sub-sequences as well, and so forth until the sequence is of length 1, i.e. it is not a series any longer. When this happens the computational cost will become $\mathcal{O}(N\ log_2 N)$ per dimension. The FFT makes a huge difference especially when applied in multiple dimensions.

6.5 Temporal discretization

In Section 4 time stepping methods were introduced and we will now derive the time stepping method for the DFT. Lets assume that the Fourier coefficients are time dependent $\alpha(t)$ we can then formulate

$$u(x,t) = \sum^{N} \alpha(t)\ e^{i2\pi kx}.$$

By using this formulation in the heat equation (1) we can write

$$\frac{d}{dt}u(x,t) = \frac{d^2}{dx^2}\left(\sum^{N} \alpha(t)\ e^{i2\pi kx}\right),$$

which has the solution

$$u(x,t) = e^{-4\pi^2 k^2 t}\sum^{N} F_k\ e^{i2\pi kx}.$$

7 Implementation

We have now gone through all the theory necessary for explicit Euler, implicit Euler, Trapezoidal method and the DFT. Python will be our tool used to write the PDE-solvers and perform the necessary calculations. Python is a object-oriented, general-purpose, high-level programming language with a large standard library which include SciPy (Scientific Python) and NumPy (Numerical Python). These will be used to perform the necessary calculations. Python also allows us to visualize these calculations with the package matlibplot. In addition to this Python has some packages allowing the PDE solvers to interact with a potential Video-Jockey program (more on this in Section 9).

7.1 NumPy and SciPy

NumPy and SciPy [7] enables the user to perform calculations fast and efficiently, it is quite similar to the environment in MATLAB. Gregory von Winckel has a website on NumPy and SciPy where he shows how it is possible to implement solvers for different ODEs and PDEs [8]. Inspiration and ideas has been taken from his website. Below follows a short description on some of the methods from these packages used to implement the PDE solvers.

- **numpy.fft.fft**
  This function computes the n-point one-dimensional DFT using the fast Fourier transform. In two dimensions ones uses numpy.fft.fft2. The inverse DFTs are numpy.fft.ifft and numpy.fft.ifft2 respectively.

- **numpy.fft.fftshift**
  This is used after numpy.fft.fft has been applied and places the zero-frequency to the center of the spectrum. The reason why it is used is that the output from numpy.fft.fft does not arrange the frequencies as desired. Its inverse is numpy.fft.iffshift.

- **scipy.sparse**
  This is a two dimensional sparse matrix package which contains a few different formats for different purposes. As $T_{\Delta x}$ more or less is tridiagonal it will be created as a sparse matrix to speed up different arithmetic operations.

- **numpy.linalg.solve**
  Computes the exact solution of $Ax = b$. A must be of full rank, i.e. all rows must be linearly independent, since otherwise A is not invertible.

- **numpy.roll**
  This method shifts a vector $k = 0, \pm 1, \pm 2, ...$ steps, e.g. $[u_0, u_1, ..., u_{N-2}, u_{N-1}] \rightarrow [u_1, u_2, ..., u_{N-1}, u_0]$.

7.2 Explicit Euler

The code for the explicit Euler method is quite straightforward and the only change possible is how to implement $T_{\Delta x}$. Here the vector has been shifted using numpy.roll. Normal matrix multiplication with $T_{\Delta x}$ or convolution using computational stencils 5 are other alternatives. It also has a practical CFL function which allows the user to see whether or not the CFL condition is violated.

```
class ExEuHeat1D_test(object):
    """
    Solves the heat equation, dU/dt = d**2U/dx**2, using explicit Euler.
    Inputs:
    - initial condition, a twice differentiable periodic function
    - dt, size of the time step.
    """
    def __init__(self, initial, dt):
        self.u = initial
        self.dt = dt
        self.n = len(self.u)
        self.dx = 1/self.n

    def time_step(self):
        """ Takes a step dt in time by shifting
        the current vector one step up/down. """
        self.u = self.u + self.dt/self.dx**2*(1*np.roll(self.u,-1)
            - 2*self.u + 1*np.roll(self.u,1))

    def get_CFL(self):
        """ Returns the CFL condition. For stability dt/dx**2 <= 1/2."""
        return self.dt/self.dx**2
```

7.3 Multigrid methods

The only difference between the implicit Euler and Trapezoidal methods is how the time stepping is done. Their common class is instantiated by:

```
    class FMGVHeat1D_test(object):
        """
        Solves the heat equation, dU/dt = d**2U/dx**2, using implicit
        time stepping methods (implicit Euler or Trapezoidal).
        Inputs:
        - initial condition, a twice differentiable periodic function
        - dt, size of the time step.
        """
        def __init__(self, u, dt):
            self.dt = dt
            self.u = u
            self.v = np.zeros(self.u.shape)
            self.n = len(self.u)
            self.A = self.get_matrix(33)
```

The recursive vycle using Jacobi iteration.

```
    def vcycle(self, v, f):
        " recursively decreases the grid and solves exactly "
        n = len(f)
        dx = 1/n
        d = 2/dx**2
        if n < 34:
            # the bottom of the recursion
            return self.solver(f, n)
        else:
            rc = self.residual(v, f, n, dx)
            v = v + rc/d
            rc = self.residual(v, f, n, dx)
            # creates the coarse grid
            rc = self.restrict(rc)
            # recursive call
            ec = self.FMGV(np.zeros(np.shape(rc)), rc)
            # vector back to ordinary size
            ef = self.prolong(ec)
            # removes the error
            v = v - ef
            rf = self.residual(v, f, n, dx)
            v = v + rf/d
        return v
```

The vcycle calls upon the residual. For the implicit Euler method:

```
    def residual(self, v, f, n, dx):
        " Calculates the residual, r = Tv - f. "
        Tdx = self.get_matrix(n)
        T = np.eye(n) - self.dt*Tdx
        Tv = np.dot(T, v)
        return Tv - f
```

and for the Trapezoidal method:

```
    def residual(self, v, f, n):
        " Calculates the residual, r = Tv - f. "
        Tdx = self.get_matrix(n)
        T = np.eye(n)-self.dt/2*Tdx
        Tv = np.dot(T, v)
        return Tv - f
```

Restrict removes half the interior grid points leaving the boundaries untouched.

```
    def restrict(self, ef):
        " Halves number of grid points. "
        return ef[0::2]
```

Prolong interpolates and uses numpy.roll to approximate the new grid points.

```
    def prolong(self, ec):
        " Interpolates making: n --> 2*n - 1. "
        ef = np.zeros(len(ec)*2 - 1)
        ef[0::2] = ec
        efdown = np.roll(ef,1)
        efup = np.roll(ef,-1)
        return ef + (efdown + efup)/2
```

In the bottom of the recursion solve is called upon. The implicit Euler and Trapezoidal method differ and for the implicit Euler method the code is

```
    def solver(self, f, n):
        Tv = np.eye(n) - self.dt * self.A
        return np.linalg.solve(Tv, f)
```

and for the Trapezoidal it is

```
    def solver(self, f, n):
        T = np.eye(n)/2 - self.dt / 2 * self.A
        return np.linalg.solve(T, f)
```

The Trapezoidal method uses a different input f than the implicit Euler. So instead of sending the initial values the Trapezoidal method calculates f as:

```
    def time_step(self):
        " takes a trapeziodal timestep "
        f = self.u + (self.dt/self.dx**2 * (1*np.roll(self.u,-1)
            - 2*self.u + 1*np.roll(self.u,1)))/2
        self.u = self.vcycle(self.v, f)
```

The implicit Euler method only needs to call upon the vcycle to make a time step.

7.4 FFT

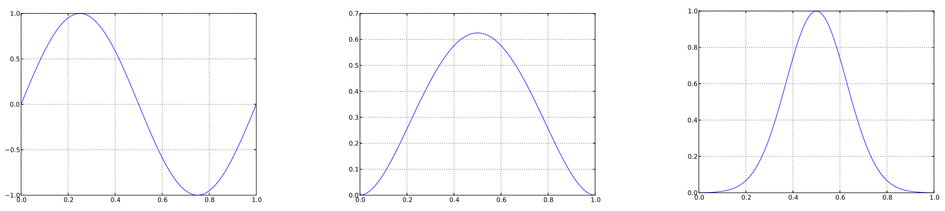
The solver first transforms the values from the spatial ux to the frequency domain uk and saves both vectors. uk is shifted to get the frequencies right place and shifted back before being transformed back to the spatial domain. For every time step uk is multiplied by the derivative constant from (6.5) and the vectors are updated.

```
class FFTHeat1D_test(object):
    """
    Solves the 1D heat equation, dU/dt = d**2U/dx**2, using FFT.
    Inputs:
    - initial condition, a twice differentiable periodic function
    - dt, size of the time step.
    """

    def __init__(self, u, dt):
        self.ux = u
        self.uk = self.compute_to_k(u)
        self.n = len(self.uk)
        self.dt = dt
        self.e = self.get_derivConst()

    def get_derivConst(self):
        kVec = np.arange(-self.n/2,self.n/2)
        derivConst = -(2*np.pi*kVec)**2*self.dt
        return np.exp(derivConst)

    def compute_to_k(self, u):
        " Converts vector from spatial to frequency domain. "
        u = np.fft.fft(u)
        return np.fft.fftshift(u)
    def compute_to_x(self, uk):
        " Converts vector from frequency to spatial domain. "
        uk = np.fft.ifftshift(uk)
        return np.fft.ifft(uk)

    def time_step(self):
        " Takes a step dt in time using FFT. "
        self.uk = self.e * self.uk
        self.ux = np.real(self.compute_to_x(self.uk))
```

8 Method comparison

We have now come to the point where it is time to actually test our theory and the implementation of our different methods. First the error in the time domain will be tested and later the execution time. It is the execution time which is most essential but not to forget stability. If the time it takes for one step in time exceeds $\frac{1}{25} = 0.04s$ the animation might look slow and choppy. If it is faster it will look smooth as the human eye cannot distinguish more than about 25 frames per second.

As our initial values for the heat equation we will use $g(x) = \sin(2\pi x)$, $g(x) = 10(x^4 - 2x^3 + x^2)$ and $g(x) = e^{-30(x-\frac{1}{2})^2}$. Plotted they look like:



(a) Trigonometric function, $\sin(2\pi x)$

(b) Polynomial function, $10(x^4 - 2x^3 + x^2)$

(c) Exponential function, $e^{-30(x-\frac{1}{2})^2}$

Figure 2: Initial values $g(x)$ used in our comparison. $\Omega \in (0, 1)$.

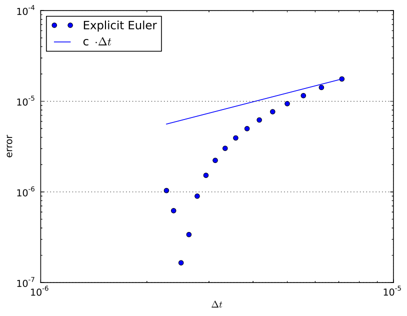
The reason for choosing these values is that they all fulfill periodic boundary conditions and have different properties. $\sin(2\pi x)$ is a trigonometric function and is $\mathcal{C}^2$. The polynomial function $10(x^4 - 2x^3 + x^2)$ is multiplied by 10 to increase the amplitude. It is periodic and $\mathcal{C}^2$. $e^{-30(x-\frac{1}{2})^2}$ also known as the Gaussian-spike contains all frequencies and satisfies $g(x) \in \mathcal{C}^2$.

8.1 Error in time domain

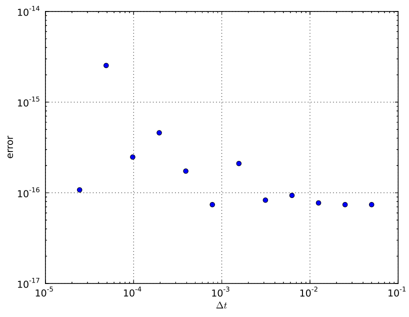
In order to check the accuracy of our solution we will use the RMS-norm (9). The exact solution to the values seen in Figure 2a is

$$u(x,t) = e^{-4\pi^2 t} \sin(2\pi x).$$

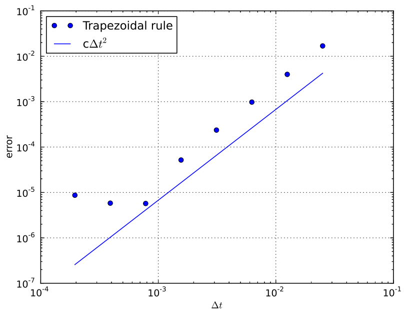
The error will be calculated by taking time steps until $t_{end} = 0.05$ has been reached. This will be done with varying sizes of Δt giving different errors depending on the method used. The two other initial values in Figure 2b & 2c lack exact solutions. Instead Δt will be chosen at least 2^3 times smaller than the smallest Δt evaluated. By doing this we can approximate it as the exact answer and use it to find an approximate error.



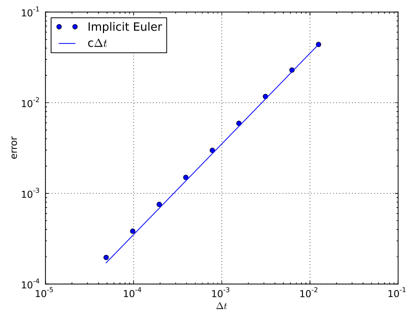
(a) As we can see the error in the Explicit Euler doesn't follow the intended line. This is due to a mix of errors in the temporal and spatial domain.



(b) The error from the FFT is $\sim 10^{-14}$ which is due to round-off errors and can be disregarded. The size of Δt is irrelevant and the solution is exact in both time and space.



(c) The first error-points in the Trapezoidal method follows the curve with slope of two and then the spatial error becomes larger, i.e. it does not matter if we make Δt smaller.



(d) The error from the Implicit Euler method correctly follows the curve with a slope of one.

Figure 3: The error is plotted against Δt until $t_{end} = 0.05$ for $g(x) = \sin(2\pi x)$. $N = 2^8 + 1$ for the multigrid methods and $N = 2^8$ for the FFT and Explicit Euler methods.

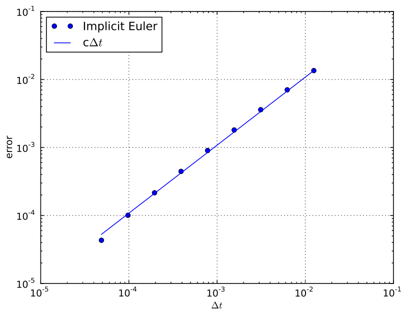
In Figure 3 the results from the calculations show the error plotted against Δt. As we can see in Figure 3b the FFT error can be disregarded and instead the number of grid points (sampling rate) will be investigated further in Section 8.3.

The Explicit Eulers CFL conditions are $\frac{1}{2} \leq \frac{\Delta t}{\Delta x^2}$. So as Δt decreases the error will become a mix between the spatial and temporal approximations which can be seen in Figure 3a. No further investigation will be made regarding the error in the time domain as we will get the same result.

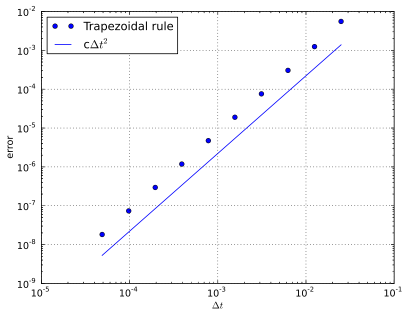
More interesting are Figures 3d & 3c. To recall the implicit Euler is $\mathcal{O}(\Delta t)$ and the Trapezoidal method is $\mathcal{O}(\Delta t^2)$. This difference is noticed as the curve in Figure 3c bends off at a much earlier stage, i.e. the temporal error becomes smaller than the domain error at a earlier stage.

Lets continue and plot $g(x) = 10(x^4 - 2x^3 + x^2)$ and $g(x) = e^{-30(x-1/2)^2}$ with the implicit Euler and Trapezoidal methods.

The plots in Figures 4 & 5 look like anticipated as they follow the curves slope. It is now clear that the implicit Euler is $\mathcal{O}(\Delta t)$ and Trapezoidal method is $\mathcal{O}(\Delta t^2)$.

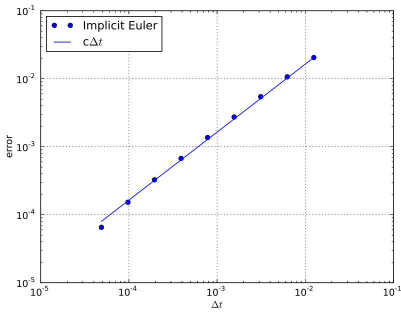


(a) The error from the implicit Euler follows the intended slope.

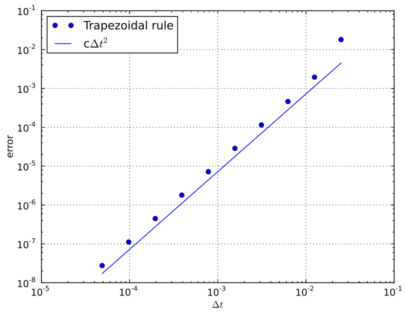


(b) The error from the Trapezoidal method behaves like expected.

Figure 4: The implicit Euler and Trapezoidal methods error plotted against Δt until $t_{end} = 0.05$ for $g(x) = 10(x^4 - 2x^3 + x^2)$ with $N = 2^8 + 1$. They behave like expected.



(a) The implicit Euler follows the curve with a slope of one.



(b) The Trapezoidal methods error behaves like predicted.

Figure 5: The implicit Euler and Trapezoidal methods error plotted against Δt until $t_{end} = 0.05$ for $g(x) = e^{-30(x-1/2)^2}$ with $N = 2^8 + 1$. The result is satisfying.

8.2 Time-error efficiency

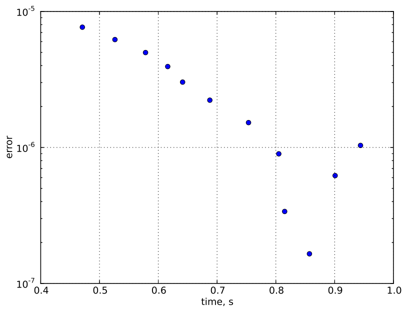
In order to compare the different methods and find an optimal Δt where the error and time elapsed are minimal an efficiency plot will be made. The error will be plotted against the time elapsed to reach $t_{end} = 0.05$ with varying Δt.

As the FFT is exact in time there is no reason to investigate this specific scenario. The Trapezoidal method is the best of the three finite difference methods and it can be compared to the FFT. If we look at the running time in Figure 9b and at $N = 2^8 + 1$ its $\leq 10^{-2}$ seconds. Besides, the number of steps until t_{end} is reached can be chosen freely.

8.3 Error spectral method

We shall now investigate how the number of grid points/frequencies taken in account affects the error. For the finite methods this is already well-known from Section 4.

In Figure 8 we see two plots with different slopes. We will connect this to the theorem in Section 6.3. The function $10(x^4 - 2x^3 + x^2)$ has two continuous derivatives and $e^{-30(x-\frac{1}{2})^2}$ has none. By using (19) and realizing that $p = 3$ respectively $p = 1$ the error can be approximated. For the polynomial the error will be $e_{RMS} = C/N^{p+1} = C/N^4$


(a) At circa 0.85 seconds the explicit Euler has its optimal ratio between time elapsed and error.

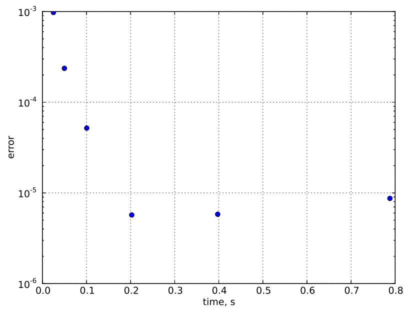

(b) Already at 0.2 seconds the Trapezoidal method reaches its optimal relation.

Figure 6: Time efficiency plots for the explicit Euler and Trapezoid method for $N = 2^8 + 1$. The Trapezoidal method reaches its maximum at a lower error four times faster than the explicit Euler.

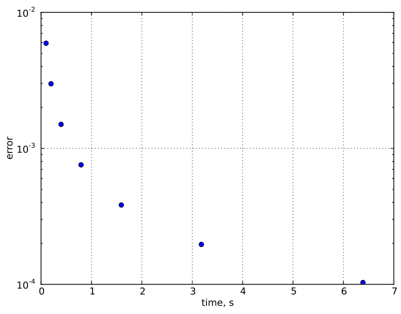


Figure 7: The implicit Euler is of $\mathcal{O}(\Delta t)$ and does not reaches any kind of maxima relation between time and error. $N = 2^8 + 1$.

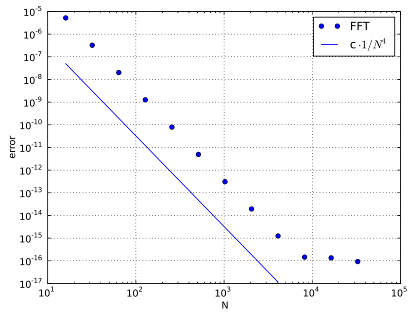
and for the Gaussian-spike $e_{RMS} = C/N^{p+1} = C/N^2$. As the slope is 4 respectively 2 this error approximation seems to be adequate.

8.4 Running time

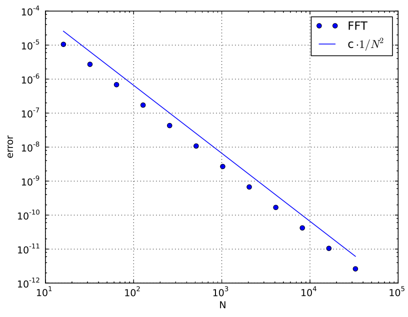
The final step of the investigation of the methods is their execution time. It is important to remember that the solvers are written as objects in Python an due to this they might act a bit unexpectedly, especially in Figure 11 it is noticeable.

All the methods in Figures 9 & 10 act like predicted. In Figure 9a the first points are a bit off however as N increases the remaining points follow the curve. The fastest method is the explicit Euler followed by the FFT, then the implicit Euler and finally the Trapezoidal method in Figure 10. This is also to expect as more work is done in the multigrid methods.

As we can see in Figure 11 the FFT and explicit Euler act like predicted, with the exception of the first few points. The multigrid methods in Figure 12 have a lot of methods calls in a recursive manner and this could be why they behave differently of what we expected. However as N increases the points seem to converge to the slope. The cause of this is perhaps that the method calls will matter less compared to the time

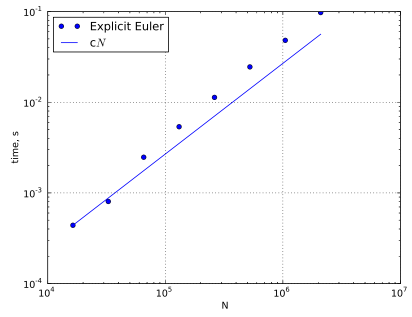


(a) The error decreases rapidly for the initial function $x^4 - 2x^3 + x^2$. It follows the curve's slope of 4 until only round-off errors start affecting.

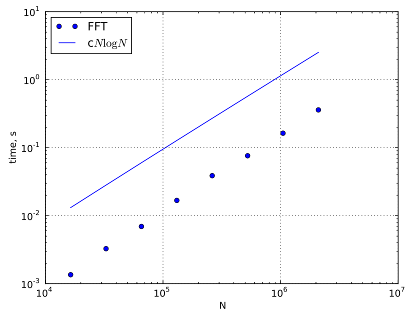


(b) The error with $g(x) = e^{-30(x-\frac{1}{2})^2}$ is plotted against N. It follows the curve which has a slope of 2.

Figure 8: The fast Fourier transform has been applied on $10(x^4-2x^3+x^2)$ and $e^{-30(x-\frac{1}{2})^2}$ for varying N.



(a) The explicit Euler is fast and follows the predicted slope with the exception of the first points.



(b) The FFT is quite fast and acts like predicted.

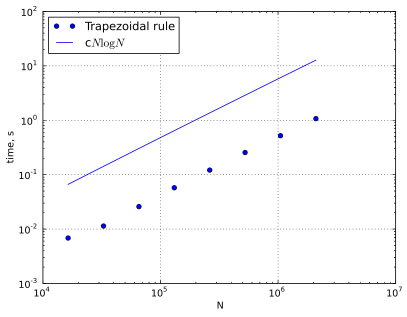
Figure 9: The explicit Euler and FFT methods showing the execution time for one step in time in one dimension.

the actual calculations take to perform. The explicit Euler is still the fastest method, about 10 times faster than the FFT. Its not visible in Figure 12 but the implicit Euler is slightly faster than the Trapezoidal.

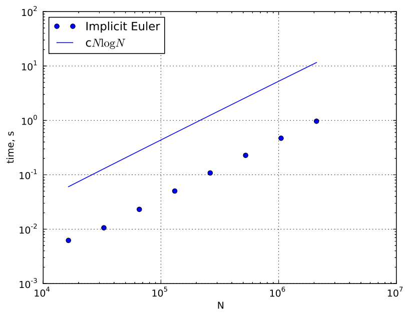
9 Application

The PDE solvers have been tested and we are ready to put them in use. There are some components needed in order to make everything work together as a prototype VJ software. First, a server is created and it opens up a server locally on the computer. A "beat-generator" connects to the server and generates a beat in a for-loop. The beat generator code is simple and therefore left out.

```
class Server(asyncore.dispatcher):
    def __init__(self, host, port):
        asyncore.dispatcher.__init__(self)
```

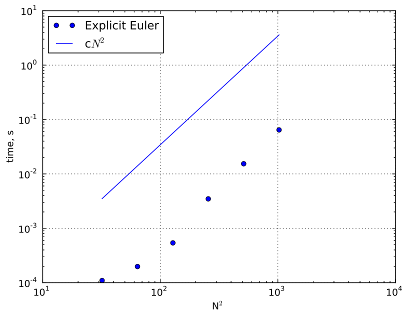


(a) The Trapezoidal method acts like expected.

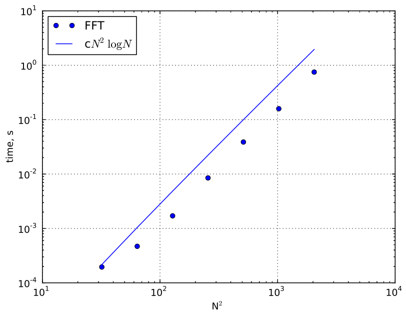


(b) The implicit Euler acts as predicted.

Figure 10: The multigrid methods showing the execution time for one step in time in one dimension.

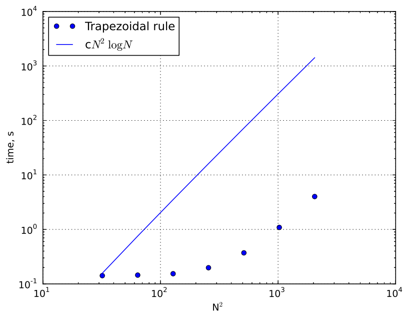


(a) The explicit Euler acts like predicted with the exception of the first points.

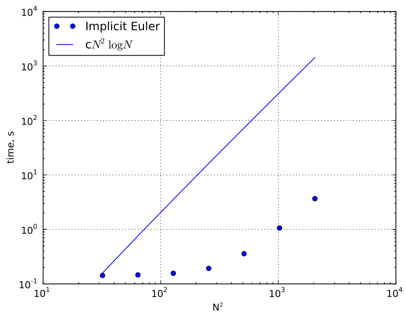


(b) The FFT follows the predicted slope.

Figure 11: The explicit Euler and FFT methods showing the execution time for one step in time in two dimensions.



(a) The Trapezoidal method does not act like predicted.



(b) The implicit Euler does not follow the intended slope.

Figure 12: The methods showing the execution time for one step in time in two dimensions.

```
        self.beatprotocolhandler = None
        self.create_socket(socket.AF_INET, socket.SOCK_STREAM)
        self.set_reuse_addr()
        self.bind((host, port))
        self.listen(10)

    def set_protocol_handler(self, beatprotocolhandler):
        self.beatprotocolhandler = beatprotocolhandler

    """ activates ConnectionHandler """
    def handle_accept(self):
        socket, address = self.accept()
        ConnectionHandler(socket, self.beatprotocolhandler)

    def run(self):
        asyncore.loop()
```

The beat recieved contains information with settings on which initial values to use and how often it is to be updated. The data sent to "Server" either has "beat" or both "beat" and "config" which updates the variables in "BeatProtocolHandler".

```
class ConnectionHandler(asynchat.async_chat):
    def __init__(self, socket, beatprotocolhandler):
        asynchat.async_chat.__init__(self, socket)
        self.beatprotocolhandler = beatprotocolhandler
        self.set_terminator('\n')
        self.data = []

    def collect_incoming_data(self, data):
        """ listens after beat and config """
        self.data.append(data)


    def found_terminator(self):
        msg = ''.join(self.data)
        self.data = []
        info = None
        try:
            info = json.loads(msg, encoding='UTF-8')
        except Exception, errtxt:
            print errtxt
            return

        if info.has_key("config"):
            self.beatprotocolhandler.update_config(info["config"])
        if info.has_key("beat"):
            self.beatprotocolhandler.update_beat(info["beat"])
```

The final large puzzle piece in the code is the "BeatProtocolHandler". It constantly runs the PDE solver's time stepping method and updates the beat count. When the beat count has reached the desired number of beats ("frequency") it updates the initial values. When doing this the thread is locked which prevents the program from trying to access the PDE solver simultaneously.

```
class BeatProtocolHandler(threading.Thread):
    def __init__(self):
        super(BeatProtocolHandler, self).__init__()
        self.count = 0
        self.mutex = threading.Lock()
        self.PDESolver = SolverFactory.create({})

    def update_config(self, config):
        with self.mutex:
            self.PDESolver = SolverFactory.create(config)

    def update_beat(self, beat):
        with self.mutex:
            self.count += 1

            if self.count % int(self.PDESolver.frequency) ==0:
                self.count = 0
                self.PDESolver.add_initial()

    def run(self):
        initial = self.PDESolver.get_u()
        xgrid = np.linspace(0, 1, len(initial))
        fig, ax = plt.subplots()
        points, = ax.plot(xgrid, initial, marker='o', linestyle='-')
        plt.axis([0, 1, -1, 1])

        while True:
            self.PDESolver.time_step()
            u = self.PDESolver.get_u()
            points.set_data(xgrid, np.real(u))
            plt.pause(0.05)
```

Some smaller simple programs have been left out but when all put together the program works. When running, the program shows the heat equation being solved in real-time.

10 Conclusions

We have investigated four different methods to compute PDEs and they all have different qualities. The optimal size of a grid would be $N = 1024 \times 1024$ which is close to standard resolution on a screen. The explicit Euler and FFT methods have an execution time of around 0.1 seconds at this resolution, Figures 11a & 11b. The implicit Euler and Trapezoidal are too slow at 1 second, Figures 12b & 12a. The maximum time should not exceed 0.04 seconds.

The FFT method is the most efficient solver for our purpose. It is easy to implement, fast and exact in time. However, as it is implemented now it requires periodic boundary conditions. There are other ways to implement it, i.e. for Dirichlet and Neumann boundary conditions [5, page 238] but we will not look further into this in this report.

There are some ways to increase the speed. One would be to perform the calculations at fewer grid points and instead interpolate to achieve the desired grid size when plotting. Another is to write more efficient code and though it is convenient to have the PDE solvers as objects for the purpose of creating a VJ-program it slows down the computations quite a lot.

With all consideration, the main goal has been reached in the report which was to show that it is possible to create visual effects dependent of music by solving PDEs in real-time.

10.1 Future work

A lot more work is needed to create a fully functioning VJ-program but this lies a bit outside of numerical analysis. A good way to visualize the calculations would be by using Open Graphics Library (OpenGL). There is a package to Python called PyOpenGL which would allow the user to create nice visualisations. Cython (an extension to C) could speed up the calculations as it lies closer to the C code that performs the calculations. Finally a Graphics User Interface (GUI) should be created so that the program would become user-friendly and accessible to the public. So, there is much more that could be done but everything starts somewhere.

References

[1] Timothy Sauer. *Numerical Analysis.* 75 Arlington Street, Suite 300, Boston, MA 02116: Pearson Education, 2006.

[2] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations - Steady-State and Time-Dependent Problems.* 3600 University City Science Center, Philadelphia, PA 19104-2688: Society for Industrial and Applied Mathematics, 2007.

[3] Steve F. McCormick William L. Briggs Van Emden Henson. *A Multigrid Tutorial second edition.* 3600 University City Science Center, Philadelphia, PA 19104-2688: Society for Industrial and Applied Mathematics, 2000.

[4] Gustaf Söderlind. *Introductions to Multigrid chap 7,8 & 9.* 2013. URL: `http://www.maths.lth.se/na/courses/FMNN15/`.

[5] Van Emden Henson William L. Briggs. *The DFT. An Owner's Manual for the Discrete Fourier Transform.* 3600 University City Science Center, Philadelphia, PA 19104-2688: Society for Industrial and Applied Mathematics, 1995.

[6] The Scipy community. Mar. 26, 2014. URL: `http://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.fft.html#numpy.fft.fft` (visited on 05/12/2014).

[7] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python.* 2001–. URL: `http://www.scipy.org/`.

[8] Gregory von Winckel. URL: `www.scientificpython.com` (visited on 10/2013).

Bachelor's Theses in Mathematical Sciences 2014:K4
ISSN 1654-6229

LUTFNA-4001-2014

Numerical Analysis
Centre for Mathematical Sciences
Lund University
Box 118, SE-221 00 Lund, Sweden

http://www.maths.lth.se/