Partial Differential Equations

Ramses van Zon

PHY1610, Winter 2026

Today's class

Today we will discuss the following topics:

- Basic approaches to solving PDEs numerically.
- How to approach the temporal part of the equations.
- How to approach the spatial part of the equations.

Classes of equations

Partial differential equations (PDEs) are differential equations with derivatives w.r.t more than 1 variable.

E.g., imagine a (scalar) field u two dimensions that obeys the following PDE:

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial x \partial y} + c\frac{\partial^2 u}{\partial y^2} = f\left(x, y, u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}\right)$$

For a, b, c constant, three classes of PDEs show up repeatedly in physical systems.

- $b^2 - 4ac < 0$ - *elliptic*
- $b^2 - 4ac = 0$ - *parabolic*
- $b^2 - 4ac > 0$ - *hyperbolic*

If there are packages that can solve PDEs in your field, then explore using them. However, writing your own is sometimes not a bad option.

Examples

Elliptic: Laplace, Poisson equations

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \rho$$

Hyperbolic: Wave equation

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0$$

Parabolic: Heat diffusion, Navier-Stokes

$$\kappa \frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}$$

Neither: continuity equation

$$\frac{\partial u}{\partial t} + \frac{\partial vu}{\partial x} = 0$$

How do we solve these problems?

Let's take a look at a parabolic equation, in particular the heat diffusion equation.

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2},$$

where u is the temperature field and κ is the thermal diffusivity.

How do we solve this equation? By discretizing in both space and time, and marching an initial condition forward in time.

This is similar to the initial value problem in the ODEs, but in this case we have another, spatial, derivative to deal with.

Discretizing the time dimension (parabolic case)

If we rewrite the right-hand side as a linear operator A:

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2} \quad \rightarrow \quad \frac{\partial u}{\partial t} = Au$$

Let $t_s = s\Delta t$. We have two basic approaches to dealing with the time part of the equation.

- Explicit methods, such as forward Euler:

$$\frac{u_{s+1} - u_s}{\Delta t} = Au_s \quad \rightarrow \quad u_{s+1} = (1 + \Delta t A)u_s$$

- Implicit methods, such as backward Euler:

$$\frac{u_{s+1} - u_s}{\Delta t} = Au_{s+1} \quad \rightarrow \quad (1 - \Delta t A)u_{s+1} = u_s$$

Explicit methods

Methods are called explicit when only u_s is on the right side of the equation. Explicit methods have some nice features:

- They are easy to implement.
- They are usually quick to calculate (no matrix inversions).
- Easier to parallelize, since the calculation is inherently local.
- There exist more-accurate explicit methods than forward Euler (Runge-Kutta, for example).

$$\frac{u_{s+1} - u_s}{\Delta t} = Au_s \quad \rightarrow \quad u_{s+1} = (1 + \Delta t A)u_s$$

But there are some serious downsides as well:

- They are not very accurate at low order ($\mathcal{O}(\Delta t)$ for forward Euler).
- They can be numerically unstable (though there are exceptions).

Implicit methods

Methods are called implicit when u_{s+1} is on both sides of the equation. E.g. backward Euler:

$$\frac{u_{s+1} - u_s}{\Delta t} = A u_{s+1} \qquad + \mathcal{O}(\Delta t)$$

and Crank-Nicolson:

$$\frac{u_{s+1} - u_s}{\Delta t} = (A u_{s+1} + A u_s)/2 + \mathcal{O}(\Delta t^2)$$

Implicit time stepping methods have some nice features:

- They are stable over a wide range of timestep sizes, sometimes unconditionally.
- Excellent for solving steady-state problems.

Downsides include being more difficult to program and parallelize.

Discretizing the spatial dimension(s)

How can we deal with the spatial aspects of the problem? How should set up our spatial discretization?

There are many different approaches one can take when dealing with fields.

- Finite difference methods.
- Finite element methods.
- Finite volume methods.
- Spectral methods.

And combinations thereof.

How you set up your discretization will determine the structure of the A operator.

Finite difference methods

Finite difference methods are the simplest way to deal with your equations.

- The simulation domain is discretized.
- Derivatives are approximated by linear combinations of function values at the grid points.

Let $x_j = j\Delta x$ and $u_j = u(x_j)$.

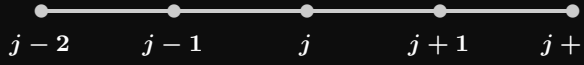
Then, one can e.g. use 'central differences' to approximate the second derivatives of u:

$$\frac{\partial^2 u_j}{\partial x^2} = \frac{u_{j+1} - 2u_j + u_{j-1}}{\Delta x^2}$$

But where does this come from?

Calculating derivatives

We need to calculate the second spatial derivatives. To do that, we discretize the x domain, and examine the Taylor expansion of some function u, centered around three different points:



$$\begin{aligned} u(x_{j-1}) &= u(x_j) - (\Delta x)\frac{\partial u(x_j)}{\partial x} + \frac{(\Delta x)^2}{2!}\frac{\partial^2 u(x_j)}{\partial x^2} + \mathcal{O}(\Delta x^3) \\ u(x_j) &= u(x_j) \\ u(x_{j+1}) &= u(x_j) + (\Delta x)\frac{\partial u(x_j)}{\partial x} + \frac{(\Delta x)^2}{2!}\frac{\partial^2 u(x_j)}{\partial x^2} + \mathcal{O}(\Delta x^3) \end{aligned}$$

where $\Delta x = x_j - x_{j-1}$.

Calculating derivatives, continued

We can write this as a matrix operation:

$$\begin{bmatrix} u(x_{j-1}) \\ u(x_j) \\ u(x_{j+1}) \end{bmatrix} = \begin{bmatrix} 1 & -\Delta x & \frac{\Delta x^2}{2} \\ 1 & 0 & 0 \\ 1 & \Delta x & \frac{\Delta x^2}{2} \end{bmatrix} \begin{bmatrix} u(x_j) \\ u'(x_j) \\ u''(x_j) \end{bmatrix}$$

To get the answer we invert the matrix:

$$\begin{bmatrix} u(x_j) \\ u'(x_j) \\ u''(x_j) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ \frac{-1}{2\Delta x} & 0 & \frac{1}{2\Delta x} \\ \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} \end{bmatrix} \begin{bmatrix} u(x_{j-1}) \\ u(x_j) \\ u(x_{j+1}) \end{bmatrix}$$

$$u'(x_j) = \frac{u(x_{j+1}) - u(x_{j-1})}{2\Delta x}$$

$$u''(x_j) == \frac{u(x_{j+1}) - 2u(x_j) + u(x_{j-1})}{\Delta x^2}$$

Discretizing the time dimension (hyperbolic case)

For a hyperbolic PDE like the wave equation

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0$$

we need to discretize the 2nd derivative with respect to time.

$$\frac{\partial^2 u(x_j, t_s)}{\partial t^2} = \frac{u(x_j, t_{s+1}) - 2u(x_j, t_s) + u(x_j, t_{s-1})}{\Delta t^2}$$

$$c^2 \frac{\partial^2 u(x_j, t_s)}{\partial x^2} = \frac{u(x_j, t_{s+1}) - 2u(x_j, t_s) + u(x_j, t_{s-1})}{\Delta t^2}$$

$$\Rightarrow u(x_j, t_{s+1}) = 2u(x_j, t_s) - u(x_j, t_{s-1}) + c^2 \frac{\partial^2 u(x_j, t_s)}{\partial x^2}$$

Note:

- We need two previous time steps.
- Equivalent to initial values for u and $\frac{\partial u}{\partial t}$

Combining both discretizations (back to parabolic case)

Let us discretize the variable u in both time and space, with $u_{s,j} = u(t_s, x_j)$:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \kappa \frac{\partial^2 u}{\partial x^2} \\ &\downarrow \\ \frac{\partial u_{s,j}}{\partial t} \equiv \frac{u_{s+1,j} - u_{s,j}}{\Delta t} &= \kappa \left[\frac{u_{s,j+1} - 2u_{s,j} + u_{s,j-1}}{\Delta x^2} \right] \end{aligned}$$

Solve for $u_{s+1,j}$ and done.

Note: It's all linear algebra...

$$\frac{\partial u_{s,j}}{\partial t} = \kappa \left[\frac{u_{s,j+1} - 2u_{s,j} + u_{s,j-1}}{\Delta x^2} \right]$$

If we write $u_{s,j}$ as a vector of values, we can rewrite our equation as a matrix operation.

$$\frac{\partial}{\partial t} \begin{bmatrix} \vdots \\ u_{s,17} \\ u_{s,18} \\ u_{s,19} \\ u_{s,20} \\ u_{s,21} \\ \vdots \end{bmatrix} = \kappa \begin{bmatrix} & & \vdots & & & & \\ \cdots & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & 0 & 0 & \cdots \\ \cdots & 0 & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & 0 & \cdots \\ \cdots & 0 & 0 & \frac{1}{\Delta x^2} & \frac{-2}{\Delta x^2} & \frac{1}{\Delta x^2} & \cdots \\ & & & \vdots & & & \end{bmatrix} \begin{bmatrix} \vdots \\ u_{s,17} \\ u_{s,18} \\ u_{s,19} \\ u_{s,20} \\ u_{s,21} \\ \vdots \end{bmatrix}$$

Which we can write as $\frac{\partial \vec{u}_s}{\partial t} = A\vec{u}_s$, with A the matrix above.

There are libraries for linear algebra (BLAS, LAPACK)

What about the edges?

The edges are a problem. Why? Well, consider the first point, $j = 0$:

$$\frac{\partial u_{s,0}}{\partial t} = \kappa \left[\frac{u_{s,1} - 2u_{s,0} + u_{s,-1}}{\Delta x^2} \right]$$

There is no $j = -1$ point!

The solution is to not use the above equation to describe the edge points. Use a different equation instead for the *boundary conditions*. One distinghuishes three general classes:

- Dirichlet: the value of field is specified at the boundary.
- Neumann: the gradient of the field at the boundary (along the normal) is specified.
- Robin: a linear combination of the field and its gradient is specified.

Example problem

Suppose we have a rod, of length 1, whose temperature $u(t, x)$ is subject to the boundary conditions:

$$\begin{aligned} \sin(10t) \qquad & x = 0, \\ 0.0 \qquad & x = 1. \end{aligned}$$

The thermal diffusivity is $\kappa = 0.2$.

If initially $u(0, x) = 0$, how does the temperature field evolves in time?

How should we solve this problem? We could use explicit or implicit methods. Today we will just use explicit methods.

Using the explicit method

As mentioned previously, explicit methods can be unstable, but let's see how it performs here.

Starting from

$$\frac{\vec{u}_{s+1} - \vec{u}_s}{\Delta t} = A\vec{u}_s,$$

we get

$$\vec{u}_{s+1} = \vec{u}_s + \Delta t A\vec{u}_s,$$

and

$$\vec{u}_{s+1} = (1 + \Delta t A)\,\vec{u}_s,$$

where 1 is the identity matrix.

Implementing boundary conditions

The boundary conditions can be implemented by adjusting the matrix used in:

$$\vec{u}_{s+1} = (1 + \Delta t A)\,\vec{u}_s = G\vec{u}_s$$

The matrix adjustments to G for the left-most point are as follows:

$$\begin{bmatrix} u_{s+1,0} \\ u_{s+1,1} \\ u_{s+1,2} \\ \vdots \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & \dots \\ \alpha & 1-2\alpha & \alpha & 0 & 0 & \dots \\ 0 & \alpha & 1-2\alpha & \alpha & 0 & \dots \\ & & & \vdots & & \end{bmatrix}}_{G} \begin{bmatrix} \sin(10t_s) \\ u_{s,1} \\ u_{s,2} \\ \vdots \end{bmatrix}$$

where we have used the definition: $\alpha = \kappa\Delta t/(\Delta x^2)$.
The $u_{s+1,0}$ should after each step be set to $\sin(10t_{s+1})$.

Implementing the other boundary condition

$$\vec{u}_{s+1} = (1 + \Delta t A)\,\vec{u}_s = G\vec{u}_s$$

Assume that there are $n = 100$ points in x. The boundary condition equation at $x = 1$ ($u_{s+1,99} = 0.0$) is implemented similarly to that at $x = 0$:

$$\begin{bmatrix} \vdots \\ u_{s+1,96} \\ u_{s+1,97} \\ u_{s+1,98} \\ u_{s+1,99} \end{bmatrix} = \underbrace{\begin{bmatrix} & & & \vdots & & \\ \dots & \alpha & 1-2\alpha & \alpha & 0 & 0 \\ \dots & 0 & \alpha & 1-2\alpha & \alpha & 0 \\ \dots & 0 & 0 & \alpha & 1-2\alpha & \alpha \\ \dots & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}_{G} \begin{bmatrix} \vdots \\ u_{s,97} \\ u_{s,97} \\ u_{s,98} \\ 0 \end{bmatrix}$$

Coding our example

```
// diffexample.cpp
#include <rarray>
#include <print>
void output(double t, double dx,
            const rvector<double>& a);
int main()
{
   int    n = 100;
   double k = 0.2;
   double runtime = 0.3;
   double dx = 1.0/(n-1);
   double dt = 0.0005;
   int nsteps = runtime / dt;
   double alpha = dt * k / (dx*dx);
   rmatrix<double> G(n,n);
   rvector<double> T(n);
   rvector<double> Told(n);
   G.fill(0.0);
   T.fill(0.0);
```

```
   for (int i = 0; i < n; i++) {
      G[i,i] = 1.0;
      if (i > 0 && i < (n-1)) {
         G[i,i-1] = G[i,i+1] = alpha;
         G[i,i] = 1.0 - 2.0 * alpha;
      }
   }
   for (int s = 0; s <= nsteps; s++) {
      if (s%(nsteps/100)==0)
         output(s*dt, dx, T);
      std::swap(T, Told);
      T.fill(0.0);
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            T[i] += G[i,j]*Told[j];
      T[0] = sin(10*(s+1)*dt);
      T[n-1] = 0.0;
   }
}
```

```
void output(double t, double dx, const rvector<double>& a) {
   for (const double& y: a)
      std::println("{} {} {}", t, a.index(y,0), y);
   std::println("");
}
```

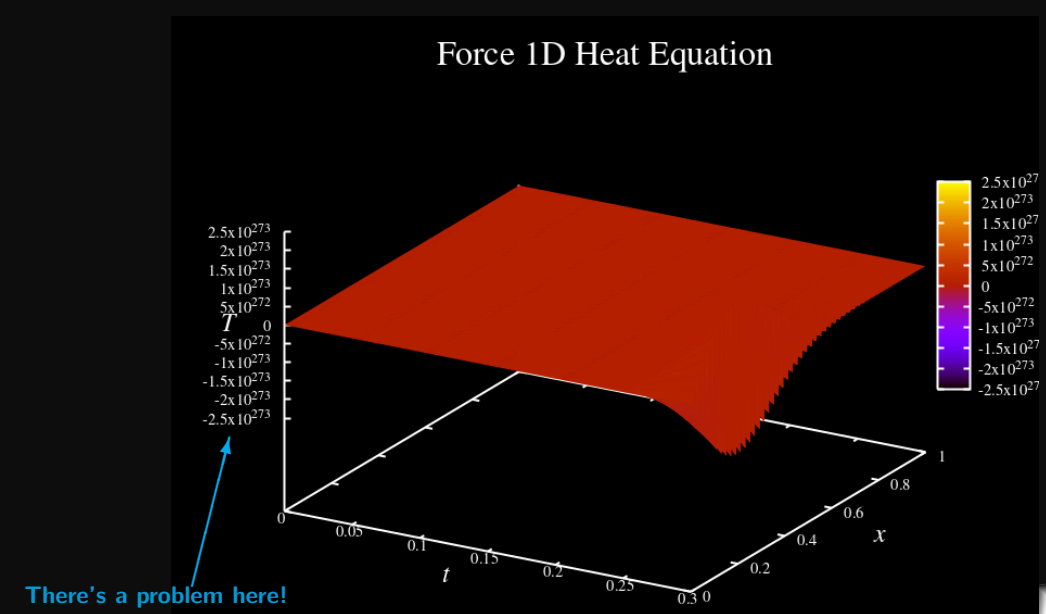
Note: Coding our example with a BLAS library

We'll see in next class that there is a library for linear algebra called BLAS.

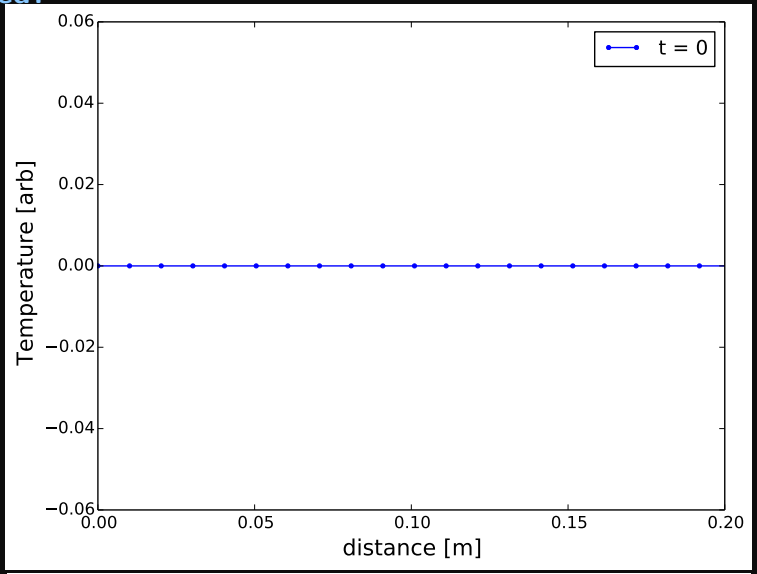
As a sneak preview, this is how we'd use it:

```
// diffexample.cpp
#include <rarray>
#include <cblas.h>
void output(double t, double dx,
            const rvector<double>& a);
int main()
{
   int    n = 100;
   double k = 0.2;
   double runtime = 2.0;
   double dx = 1.0/(n-1);
   double dt = 0.0005;
   int nsteps = runtime / dt;
   double alpha = dt * k / (dx*dx);
   rmatrix<double> G(n,n);
   rvector<double> T(n);
   rvector<double> Told(n);
   G.fill(0.0);
   T.fill(0.0);
   for (int i = 0; i < n; i++) {
      G[i,i] = 1.0;
      if (i > 0 && i < (n-1)) {
         G[i,i-1] = G[i,i+1] = alpha;
         G[i,i] = 1.0 - 2.0 * alpha;
      }
   }
   for (int s = 0; s <= nsteps; s++) {
      if (s%(nsteps/100)==0)
         output(s*dt, dx, T);
      std::swap(T, Told);
      cblas_dgemv(CblasRowMajor,CblasNoTrans,
                  G.extent(0),G.extent(1),1.0,
                  &G[0,0],G.extent(0),
                  &Told[0],1,0.0,&T[0],1);
      T[0] = sin(10*(s+1)*dt);
      T[n-1] = 0.0;
   }
}
```

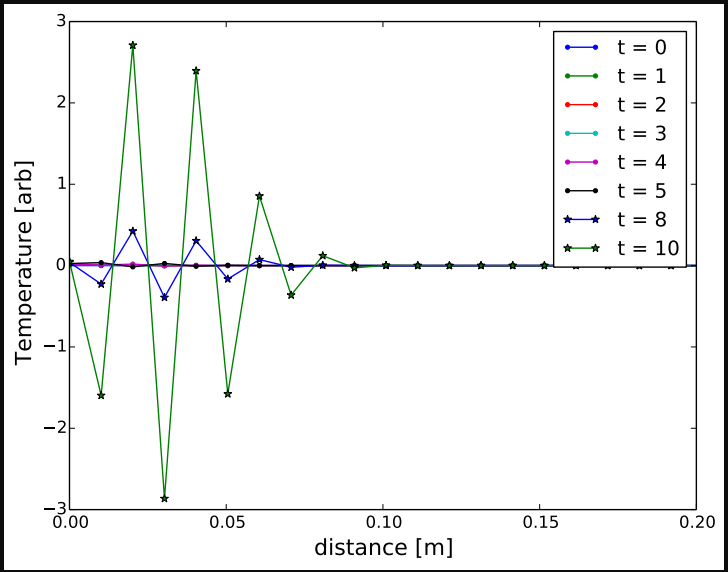
What is the result of running this?


Force 1D Heat Equation
T
t
x
2.5x10^273
2x10^273
1.5x10^273
1x10^273
5x10^272
0
-5x10^272
-1x10^273
-1.5x10^273
-2x10^273
-2.5x10^273
0
0.05
0.1
0.15
0.2
0.25
0.3
0
0.2
0.4
0.6
0.8
1
2.5x10^27
2x10^273
1.5x10^27
1x10^273
5x10^272
0
-5x10^272
-1x10^273
-1.5x10^27
-2x10^273
-2.5x10^27
There's a problem here!

What happened?



What happened?



What went wrong?

The algorithm is unstable, but why? Consider the algorithm in the interior of the domain

$$\begin{bmatrix} \vdots \\ u_{s+1,17} \\ u_{s+1,18} \\ u_{s+1,19} \\ u_{s+1,20} \\ u_{s+1,21} \\ \vdots \end{bmatrix} = \begin{bmatrix} & & & \vdots & & & \\ \dots & \alpha & 1-2\alpha & \alpha & 0 & 0 & \dots \\ \dots & 0 & \alpha & 1-2\alpha & \alpha & 0 & \dots \\ \dots & 0 & 0 & \alpha & 1-2\alpha & \alpha & \dots \\ & & & \vdots & & & \end{bmatrix} \begin{bmatrix} \vdots \\ u_{s,17} \\ u_{s,18} \\ u_{s,19} \\ u_{s,20} \\ u_{s,21} \\ \vdots \end{bmatrix}$$

For this to be stable, all eigenvalues of the matrix $|\lambda| \leq 1$, which requires $\alpha \leq 1/2$, i.e.,

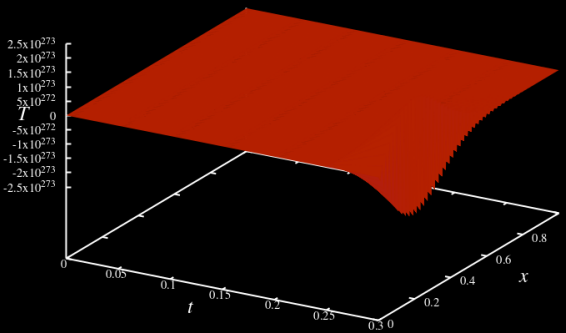
$$\Delta t \leq \frac{\Delta x^2}{2\kappa}$$

(Ansatz for eigenvectors: $u_j = e^{2\pi ijk} \Rightarrow \lambda_k = 1 - 4\alpha \cos^2(\pi k)$, demand $\lambda_k \geq -1$ for all k.)

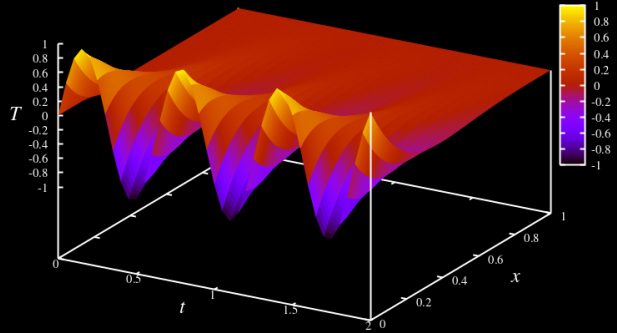
Can we save it?

Yes! We just need to pick a better value of Δt.

$\Delta t = 0.0005$



$\Delta t = 0.00005$



Rising costs

$$\Delta t \leq \frac{\Delta x^2}{2\kappa}$$

- Because of this condition, doubling the spatial resolution means an 8-fold increase in computation:
  - a factor of 2 related to the spatial resolution
  - times a factor of 4 because Δt needs to be that much smaller
- It's even worse in higher dimenions: In a 3d application, doubling the linear spatial resolution means a factor of 8 in number of points, so

  For a 3d parabolic equation doubling the resolution increases the computational cost by a factor of 32.
- For hyperbolic equations, it's only slightly better, as $\Delta t \sim \Delta x$ ("CFL condition").

Notes about this implementation

A few things to recognize about this implementation:

- The code allocates and fills the entire matrix G. Most of the matrix contains zeros.
- This is a good way to implement things initially, for testing. Not good for production runs.
- Why? Because in this case the matrix is banded, and so a routine for a banded matrix should have been used to solve this problem.
- Who cares? Banded routines are faster and use much much less memory. Use them!
- Downside: banded matrices are a little more complicated to store.

Next lecture, we'll discuss linear algebra libraries that implement these features:

Basic **L**inear **A**lgebra **S**ubroutines