Computational Maths for Dummies 2

Stephen Wade

July 12, 2008

Before we summarise what we found last time, this article in the series is going to need a slight bit of patience with the mathematics. Read out equations if it will help you understand what's happening, try and use similar words from the examples in the previous article to help you get a feel for reading the mathematics.

Last time we

- observed a *governing equation*, the diffusion equation, which is a *partial differential equation*,
- examined the concept of *discretising* time and space,
- considered how to compute an approximation to an equation as opposed to finding an exact solution,
- were given an approximate formula for calculating the first and second derivative
- we used these ideas to compute result for the value u at jth pixel and the $k+1$th frame using the formula

$$u_j^{k+1} = 2\alpha\left(u_{j-1}^k - 2u_j^k + u_{j+1}^k\right) + u_j^k,$$

to approximate the one-dimensional diffusion equation,

$$\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial^2 x}. \tag{1}$$

A quick note about extending this into higher dimensions. The original equation

$$\frac{\partial u}{\partial t} = \alpha\nabla^2 u \tag{2}$$

has a Laplacian operator, ∇^2, which does have expressions in different co-ordinate systems. What we normally use is a Cartesian coordinate system (feel free to look these up) with coordinates (x, y, z) and in this system the Laplacian is

$$\nabla^2 u == \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}.$$

For the discrete space $(x, y, z) = (x_i, y_j, z_k)$, denoting $u_{i,j,k}$ as the value of u at $(x, y, z) = (x_i, y_j, z_k)$, the derivative at that point can be approximated

$$\begin{aligned}
\left.\frac{\partial^2 u}{\partial x^2}\right|_{i,j,k} &= \alpha(u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}) \\
\left.\frac{\partial^2 u}{\partial y^2}\right|_{i,j,k} &= \alpha(u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}) \\
\left.\frac{\partial^2 u}{\partial z^2}\right|_{i,j,k} &= \alpha(u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}).
\end{aligned}$$

We should be able to see that for any derivative, we can discretise the space or time, and then compute an approximation to the derivative. Something interesting in the example code from last time is that as the α value increases beyond 0.25 there are large errors in the solution, and it's even worse for the $2D$ case. To get to the bottom of this, and to rectify it, we need to do some more careful work.

Firstly we need to introduce the idea of the *size* of the discretisation. Let's say we have a two dimensional grid $(x, y) = (x_i, y_j)$ which represents the *discretisation scheme*, see Figure 1

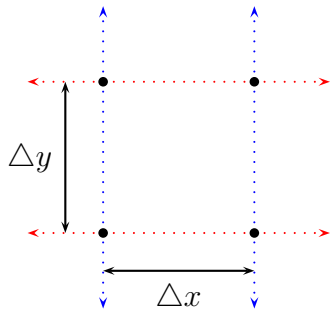

Figure 1: Discretisation scheme

So $\triangle y$ and $\triangle x$ represent the distance between the discrete elements, this should feature in our equations. If we have some quanitity u_i (such as heat) at the point $s = s_i$ in some discretisation scheme with length $\triangle s$ then we have some standard formulas we use to approximate derivatives.

Approximations to the first derivative ...

$$\begin{aligned}
\left.\frac{\partial u}{\partial s}\right|_i &\approx \frac{u_{i+1} - u_i}{\triangle s} && \text{1st order forward difference} \\
&\approx \frac{u_i - u_{i-1}}{\triangle s} && \text{1st order backward difference} \\
&\approx \frac{u_{i+1} - u_{i-1}}{2\triangle s} && \text{2nd order centred difference} \\
&\approx \frac{-u_{i+2} + 4u_{i+1} - 3u_i}{2\triangle s} && \text{2nd order forward difference} \\
&\approx \frac{3u_i - 4u_{i-1} + u_{i-2}}{2\triangle s} && \text{2nd order backward difference.}
\end{aligned}$$

Approximations to the second derivative ...

$$\begin{aligned}
\left.\frac{\partial^2 u}{\partial s^2}\right|_i &\approx \frac{u_{i+1} - 2u_i + u_{i-1}}{(\triangle s)^2} && \text{2nd order centred difference} \\
&\approx \frac{u_{i+2} - 2u_{i+1} + u_i}{(\triangle s)^2} && \text{1st order forward difference} \\
&\approx \frac{u_i - 2u_{i-1} + u_{i-2}}{(\triangle s)^2} && \text{1st order backward difference.}
\end{aligned}$$

The *1st order* and *2nd order* term is an indication of the accuracy of the approximation. Centred difference approximations appear to have greater accuracy given the same number of operations (addition and subtraction) as per the *one sided* forward or backward difference approximations.

As it turns out, the error observed when using larger values of α arises not from the accuracy of the approximation to the derivative, but in our choice of using a *forward* difference approximation for the time derivative and the resulting computational scheme.

Define a *constant* $\triangle t$ representing the time elapsing between each frame, and a *constant* $\triangle x$ representing the physical distance between adjacent texture pixels. Using a *backward* difference approximation to the time derivative and the same centred difference for the spatial derivative gives, for the jth pixel and kth frame in a one-dimensional texture,

$$\begin{aligned}
\left.\frac{\partial u}{\partial t}\right|_j^k &= \frac{u_j^k - u_j^{k-1}}{\triangle t} \\
\left.\frac{\partial^2 u}{\partial x^2}\right|_j^k &= \frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{(\triangle x)^2}
\end{aligned}$$

and so

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

becomes

$$\frac{u_j^k - u_j^{k-1}}{\triangle t} = \alpha \frac{u_{j+1}^k - 2u_j^k + u_{j-1}^k}{(\triangle x)^2}$$

If we want to compute the current frame, k, then we can only know the terms from the previous frame $k-1$, i.e. the terms similar to u_*^{k-1} (where $*$ can be the number of any pixel). All the unknown terms can be written as u_*^k. If we get these on the right and left hand side of the equation respectively, then

$$(\triangle x)^2 u_j^k - \triangle t \alpha (u_{j+1}^k - 2u_j^k + u_{j-1}^k) = (\triangle x)^2 u_j^{k-1}$$

The computational 'stencil' now looks different,

Repeating, we have *three unknown* terms $u_{j+1}^k, u_j^k, u_{j-1}^k$ on the left hand side, and *one known* term u_j^{k-1} on the right hand side. Let's have a look at an example of this problem for a very small data set, say a 5×1 pixel texture. For this we consider that we know the values $u_j^{k-1} = (1, 3, 3, 3, 1)$, $\triangle x = 1, \triangle t = 1$, and the boundary condition is,

$$u_1^k = u_5^k = 1$$

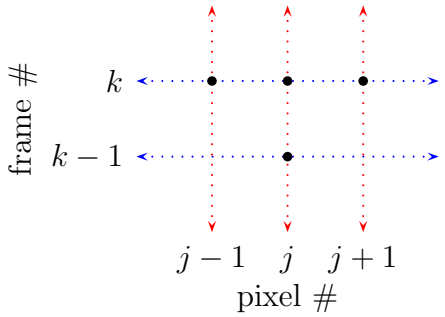

Figure 2: Computational 'stencil'

see Figure 3. The lines in the figure represent the equations we get using the backward difference approximation for the time derivative and the centred difference for the spatial derivative, the equations are

$$\begin{aligned} u_2^k - \alpha(u_3^k - 2u_2^k + u_1^k) &= u_2^{k-1} \\ u_3^k - \alpha(u_4^k - 2u_3^k + u_2^k) &= u_3^{k-1} \\ u_4^k - \alpha(u_5^k - 2u_4^k + u_3^k) &= u_4^{k-1} \end{aligned}$$

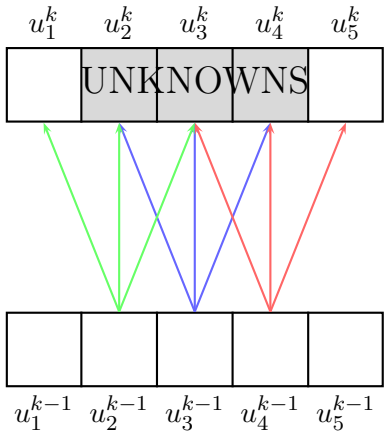

Figure 3: 5×1 - one dimensional diffusion problem

If we carefully replace the values that we have *assumed* as either known or boundary conditions, then the equations listed are

$$\begin{aligned} u_2^k - \alpha(u_3^k - 2u_2^k + 1) &= 3 \\ u_3^k - \alpha(u_4^k - 2u_3^k + u_2^k) &= 3 \\ u_4^k - \alpha(1 - 2u_4^k + u_3^k) &= 3 \end{aligned}$$

Now, this is the part you need patience with, firstly we re-write these formulas,

$$\begin{aligned} (1+2\alpha)u_2^k - \alpha u_3^k - \alpha &= 3 \\ -\alpha u_2^k + (1+2\alpha)u_3^k - \alpha u_4^k &= 3 \\ -\alpha u_3^k + (1+2\alpha)u_4^k - \alpha &= 3 \end{aligned}$$

and a bit more spaced out

$$\begin{array}{ccccc} (1+2\alpha)\times u_2^k & -(\alpha)\times u_3^k & (0)\times u_4^k & = & 3+\alpha \\ -(\alpha)\times u_2^k & +(1+2\alpha)\times u_3^k & -(\alpha)\times u_4^k & = & 3 \\ (0)\times u_2^k & -(\alpha)\times u_3^k & +(1+2\alpha)\times u_4^k & = & 3+\alpha \end{array}$$

Now for some, this is the tricky part, we write this as a matrix

$$\begin{bmatrix} (1+2\alpha) & -\alpha & 0 \\ -\alpha & (1+2\alpha) & -\alpha \\ 0 & -\alpha & (1+2\alpha) \end{bmatrix} \begin{bmatrix} u_2^k \\ u_3^k \\ u_4^k \end{bmatrix} = \begin{bmatrix} 3+\alpha \\ 3 \\ 3+\alpha \end{bmatrix}$$

Matrices are not difficult to grasp, compare the first equation in the spaced out form to the first row of the matrix. The first element on the right hand side matches the first equation's right hand side $(3+\alpha)$. Also, if you take each 'element' in the first row of the matrix, and multiply it by the respective element from the 'column' matrix next to it, and multiply them together, then you have exactly the left hand side of the equation. The first element of the first row is $(1+2\alpha)$ and you multiply that by the first element of the column matrix, u_2^k, etc. Figure 4 shows this roughly by colours, similarly the other equations follow

$$\begin{bmatrix} (1+2\alpha) & -\alpha & 0 \\ -\alpha & (1+2\alpha) & -\alpha \\ 0 & -\alpha & (1+2\alpha) \end{bmatrix} \begin{bmatrix} u_2^k \\ u_3^k \\ u_4^k \end{bmatrix} = \begin{bmatrix} 3+\alpha \\ 3 \\ 3+\alpha \end{bmatrix}$$

e.g.

$$(1+2\alpha)\times u_2^k - \alpha \times u_3^k + 0 \times u_4^k = 3+\alpha$$

$$-\alpha \times u_2^k + (1+2\alpha)\times u_3^k + \alpha \times u_4^k = 3$$

Figure 4: Matrix multiplication by colours

So now that we have this represented by a matrix, we could solve the unknowns u_2^k, u_3^k, u_4^k. We let these unknowns be the vector $\mathbf{u}_k = (u_2^k, u_3^k, u_4^k)$ and we let the right hand side of the equation be the vector $\mathbf{d} = (3+\alpha, 3, 3+\alpha)$ which is *constant*. The matrix on the left hand side is denoted A. We can now write

$$A\mathbf{u}_k = \mathbf{d}$$

This being a computational mathematics article it would be more pertinent to *compute* approximations to $\mathbf{u}_k$, than solve the unknowns directly. One reasonable method for doing this is *Gauss-Siedel iteration*.

Gauss-Siedel iteration, by name, is an iterative method. Iterative methods for solving equations are essentially like any iterative method implemented in programming before, it simply runs through the same steps over and over until some stop condition is met. In this case, the stop condition is usually some minimum accuracy (or in other words some maximum error) in the approximate solution.

Gauss-Siedel iteration requires we take some initial stab in the dark at a solution, noted as $\mathbf{u}_k^{[0]}$, then we will produce the next approximate solution, and another, and another until we reach the required accuracy. The solutions generated are labelled as $\mathbf{u}_k^{[1]}, \mathbf{u}_k^{[2]}, \mathbf{u}_k^{[3]}, \ldots$ and so forth. For most finite difference methods we expect that things are going to behave smoothly, and as such a good guess to what things will look like for the next frame is to guess that they will look *exactly* like the do in the current frame.

So we take $\mathbf{u}_k^{[0]} = \mathbf{u}_{k-1}$. Read that as "the guess for the approximate solution for the next frame is equal to the solution we already know from the previous frame".

The most convenient notation for Gauss-Siedel iteration is in matrix notation, however it is not a convenient notation for implementing it. For reference however, here is Gauss-Siedel iteration in matrix form. If we wish to solve a matrix system $A\mathbf{u}_k = \mathbf{d}$ then starting with an initial guess $\mathbf{u}_k^{[0]}$ then

$$N\mathbf{u}_k^{[i+1]} = P\mathbf{u}_k^{[i]} + \mathbf{d} \tag{3}$$

for $i = 1, 2, 3, \ldots$. N is the lower-triangle and the diagonal elements of A, and P is the negative of the upper-triangle of A. For our example,

$$A = \begin{bmatrix} (1+2\alpha) & -\alpha & 0 \\ -\alpha & (1+2\alpha) & -\alpha \\ 0 & -\alpha & (1+2\alpha) \end{bmatrix}$$

$$N = \begin{bmatrix} (1+2\alpha) & 0 & 0 \\ -\alpha & (1+2\alpha) & 0 \\ 0 & -\alpha & (1+2\alpha) \end{bmatrix}, \qquad P = \begin{bmatrix} 0 & \alpha & 0 \\ 0 & 0 & \alpha \\ 0 & 0 & 0 \end{bmatrix}$$

Using matrix multiplication for 3, this becomes three formulas

$$\begin{aligned} (1+2\alpha)u_2^{k[i+1]} &= \alpha u_3^{k[i]} + 3 + \alpha \\ -\alpha u_2^{k[i+1]} + (1+2\alpha)u_3^{k[i+1]} &= \alpha u_4^{k[i]} + 3 \\ -\alpha u_3^{k[i+1]} + (1+2\alpha)u_4^{k[i+1]} &= 3 + \alpha \end{aligned}$$

The first equation can be re-arranged easily to

$$u_2^{k[i+1]} = \frac{\alpha u_3^{k[i]} + 3 + \alpha}{1+2\alpha}$$

So now we know $u_2^{k[i+1]}$ and we can use this value to compute the next formula

$$u_3^{k[i+1]} = \frac{\alpha u_4^{k[i]} + 3 + \alpha u_2^{k[i+1]}}{1+2\alpha}$$

We now know $u_3^{k[i+1]}$ and then this can be substituted into the last formula to give us $u_4^{k[i+1]}$

$$u_4^{k[i+1]} = \frac{3 + \alpha + \alpha u_3^{k[i+1]}}{1 + 2\alpha}$$

There you have it, three easy steps that we can iterate through to compute the $i+1$th approximation to the solution of $A\mathbf{u}_k = \mathbf{d}$. Although we still have a long way to go with our understanding of these methods, and perhaps some notes on how to actually implement them in code, we are going to take a brief break and look at the results we get for this 5×1 texture problem from our original method, using a *forward* difference approximation to time, and the results we get from the formula obtained using a *backward* difference approximate for the time derivative.

Matlab ™or Octave code to test this for just *one* frame is

```
function [ ] = Tsumea4( )
   % Compares the new method in the second article to the
   % old method presented in the first article.

   % Assume a larger alpha value than previously
   alpha = 5.0;
   % Create the matrices required for the new method
   A = [(1 + (2*alpha)) -alpha 0; -alpha (1+(2*alpha)) -alpha; ...
       0 -alpha (1+(2*alpha)) ];
   d = [3 + alpha; 3; 3+alpha];
   x0 = [3;3;3];

   % Use Gauss-Siedel iteration to compute solution
   x1 = myGaussSiedel(A,d,x0);

   % Code for computing the solution using the old method
   x0 = [1;3;3;3;1];
   x2(1:3,1) = 2 * alpha * ( x0(1:3,1) - (2 * x0(2:4,1)) + x0(3:5,1) ) ...
      + x0(2:4,1);

   disp('Solution obtained with original method:');
   disp(x2);
   disp('Solution obtained with new method:');
   disp(x1);


   plot([1 ; x1 ; 1]);
   hold on;
   plot([1 ; x2 ; 1]);
   hold off;

   plot([1;3;3;3;1]);
   hold on;
   plot([1 ; x1 ; 1]);
   hold off;

end

function [ x ] = myGaussSiedel( A, d, x0)
   % returns approximate to x for Ax = d
   % using Gauss-Siedel iteration
   % stop condition is that max|Ax - d| < 0.001
   % using initial estimate x = x0

   % Counts how many times we iterate through
```

```
    j = 0;

    % start with initial guess
    x = x0;

    % get the size of the matrix we are computing
    n = size(A);
    n = n(1,2);

    % iterate until desired accuracy required

    while max(abs((A * x) - d)) > 0.001
        for i = 1:1:n
            x(i,1) = x(i,1) + ((-A(i,:) * x) + d(i,1)) ./ A(i,i)
        end
        j = j + 1;
    end

    disp('Number of iterations required to converge:');
    disp(j);

end
```

The output graph is

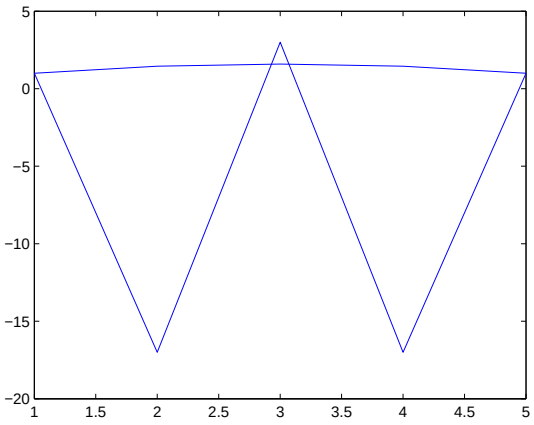

Figure 5: Graph of old vs. new method

The highly jagged line in Figure 5 is the old computational method in the first article, and the smooth-ish line represents the new method for calculating. The diffusion equation is supposed to be smooth, it's like watching heat slowly dissipate, or watching dye dilute into a solution. If we consider this to be the model of some heated metal object like a nail, and we specified the ends to be held at a fixed temperature, we would expect that eventually the rod will settle into a uniform temperature.

We plot the initial state of our variable $\mathbf{u}_0$ and compare it to $\mathbf{u}_1$ obtained with the new method, and we get Figure 6

Figure 6 strongly follows the model that we are thinking of where heat is diffusing out of the material and it is slowly heading towards having a uniform temperature as that of the two ends.

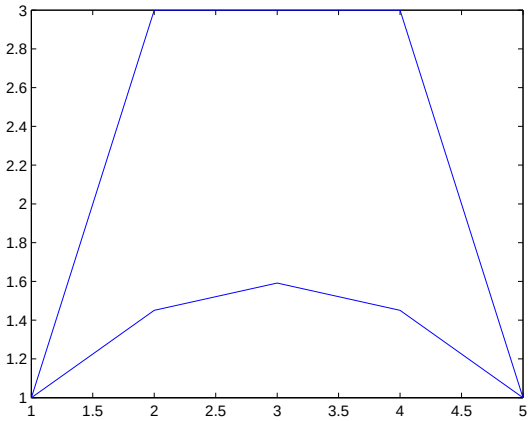

Figure 6: Graph of $\mathbf{u}_0$ and $\mathbf{u}_1$

Well, that was *a lot* to get through in one article. Here's a list of what we covered, we

- looked briefly at the Laplacian, ∇^2, in higher dimensions,
- have more formulas for approximating first and second derivatives to higher accuracy, introduced the notion of the size of the discretisation scheme (more on this later)
- know that the first articles method is inaccurate for large values of α,
- used a *backward* difference for the time derivative in the *diffusion equation* to create new formulas to solve for the next frame of a texture (or next time step of any particular problem)
- used *matrices* and *vectors* to re-write the equations found and
- used *Gauss-Siedel iteration* to calculate an approximate solution to those equations and noted that they seemed accurate for a large test value of α

We can now get into some more implementation details with the next article with some examples, but take some time to get through the details of this article to cement your understanding and maybe even learn a few mathematics tricks of the trade along the way.