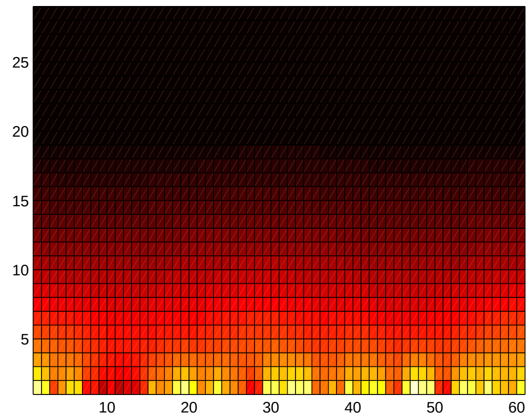
Computational Maths for Dummies 1

Stephen Wade

July 11, 2008



Does anyone remember a very old, simple fire effect from back in the era of demo programming? It looks like a sheet of fire slowly making it's way through the black abyss of the screen. By modern standards it looks woefully pitiful, but it does do a reasonably good job of introducing something that is the basis of a huge portion of computational mathematics.

The algorithm of the fire demo could be roughly summarised as follows; At the bottom row of the screen a randomly generated set of red, orange, white, yellow pixels is generated per frame. Then for each row from the top of the screen, down to the bottom, each pixel's colour is taken to be the average of the three nearest pixels in the row beneath.

If we think a little, this isn't a bad model of the fire effect. At the very bottom there is a very random energy profile, there are bright spots (white) that represent a dense pocket of heat energy, and darker spots (dark, deep red colours) that represent low amounts of heat energy. This bottom row is like chemical reaction, for example combustion. As we go upwards in the screen we find that at each point, the heat energy is affected by the amount of energy underneath it, assuming that the heat energy is travelling upwards.

The basic *physical* motion here is some kind of advective/diffusive motion. As well as many other physical effects, this has been given a mathematical description. Descriptions like these are often called *governing equations*, as they *govern the motion.*

Now please, pretty please, don't run away yet. This is the scariest part, it's a partial differential equation,

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

Most governing equations are partial differential equations, but that's not going to be a problem because the whole point of this is to work through computing these *rather than* solving them.

Imagine a really shallow puddle of water, see Figure 1. Consider a variable u that represents the concentration of a dye for some tiny area of interest in that puddle, and a variable t representing the current point in time. If we 'zoom out' and look at the whole puddle, we expect that the dye will slowly dissipate and spread, this is known as 'diffusion'.

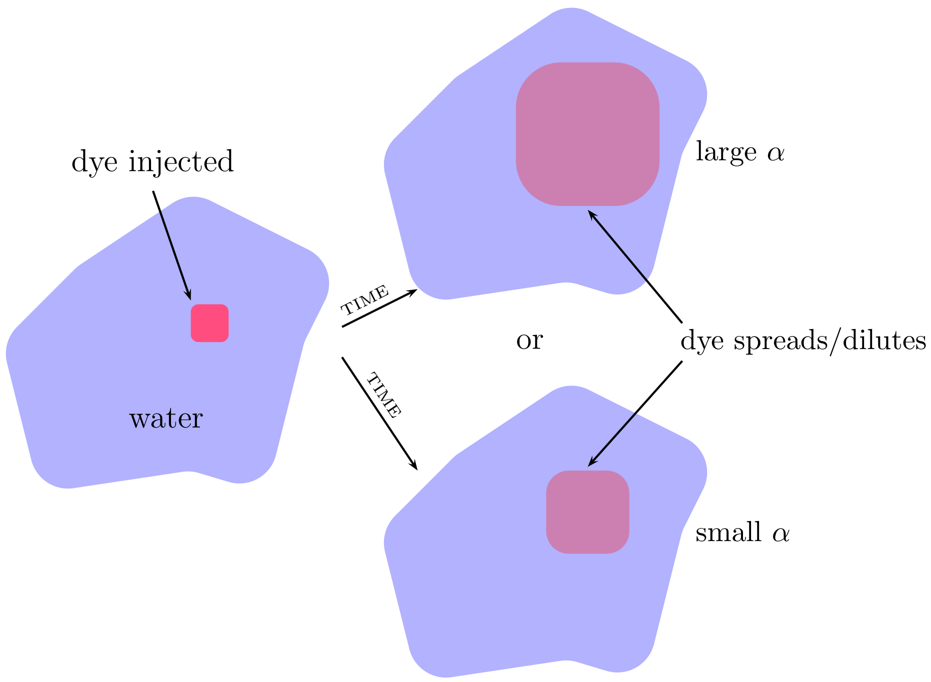


Figure 1: Dye-diffusing in puddle concept

Zooming back in, and looking at a small element of the puddle, the left hand side of the equation reads "the rate of change of u (dye concentration) in that small area with respect to t (time)". The right hand side of the equation is "some constant, α, multiplied by the *Laplacian* (∇^2) of u".

α is what we call a *diffusivity* constant, and it can take any positive value, it doesn't really matter. The Laplacian, ∇^2, is basically function that takes u as an argument. It is a measure of the gradient of the amount of dye entering or leaving through the edges of the tiny puddle area we are looking at.

If we consider one spatial dimension, say the x direction on the screen, the Laplacian looks a little different, and equation (1) can be written as

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

Now this right hand side looks more like the left hand side and it can be read as "α multiplied by the rate of change of *flux* of u (dye concentration) with respect to x (space)". Put both sides together and read it out loud

"The rate of change of u (dye concentration) in a small area with respect to t (time) *is equal to* α multiplied by the rate of change of *flux* of u (dye concentration) with respect to x (space)"

To make things easy, we can simply call these weird terms on the left and right hand side *derivatives*. Remember we don't need to solve these, we need to *compute* them. In fact, these weird terms in an equation can be neatly approximated (although in most cases, not solved) by very simple computer codes.

The computer code we shall first look at is called a finite difference method.

Finite what?

A finite difference method is perhaps the simplest and most crude way of computing or approximating these derivatives. It does suffer from it's simplicity, but it's certainly not useless.

In order to use a finite difference method we need to split the space we are interested in into little pieces. This is known as *discretisation*, and an obvious example of this is a computer screen. We are aiming to *compute* the dye concentration at each *point* we are interested in (say some texture image). This is *fundamentally* different from solving mathematical equations *analytically* where we would solve the dye concentration for all time and space (a continuum as opposed to a discretised space).

Notation is important. Consider one row of pixels on a texture, and we give each pixel a value of x, we say that the jth pixel has a value x_j. Each frame of the rendering has a time value t, we say the kth frame has the time value t_k. Each value of dye concentration is then labelled u_j^k, meaning the value of u at the kth frame, on the jth pixel.

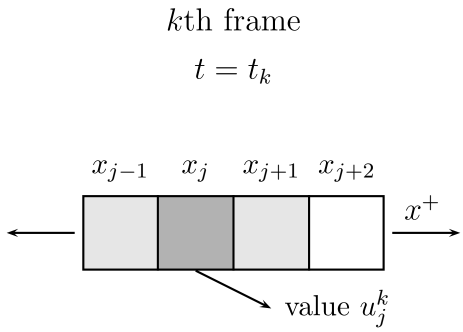


Figure 2: Row of pixels from 'texture'

A finite difference approximation to the left hand side of the governing equation (2) might read as "for the kth frame, the rate of change of u with respect to t at the jth pixel is approximately equal to the value of u at the jth pixel from the next frame minus the value of u at the jth pixel of the current frame". In maths speak,

$$\left.\frac{\partial u}{\partial t}\right|_j^k \approx \frac{u_j^{k+1} - u_j^k}{2}$$

That little bar on the right of that *derivative* term $\frac{\partial u}{\partial t}$ just means the value of that derivative at the point $x = x_j$ at time $t = t_k$.

A finite difference approximation to the right hand side would read as "for the kth frame, α multplied by the rate of change of flux of u with respect to x at the jth pixel is approximately equal to α multiplied by the sum of the value of u at the $j-1$th pixel minus two times the value of u at the jth pixel plus the value of u at the $j+1$th pixel all from the kth frame". In maths speak

$$\alpha \left.\frac{\partial^2 u}{\partial x^2}\right|_j^k \approx \alpha \left(u_{j-1}^k - 2u_j^k + u_{j+1}^k\right)$$

We may have noticed now that we have a *new* approximate equation,

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

We can see that there are four different values in the equation, $u_j^{k+1}, u_{j-1}^k, u_j^k, u_{j+1}^k$, this gives rise to a computational stencil, like Figure 3.

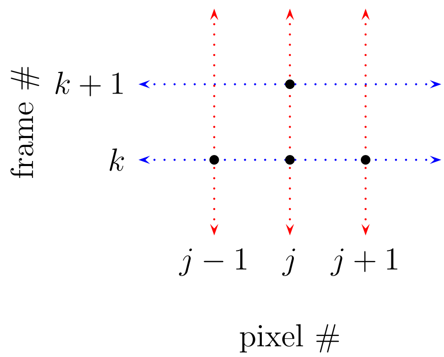


Figure 3: Computational 'stencil'

At a guess we can solve the value of u at the jth pixel for the next frame, based on three values of u from the current frame, that is at the $j-1$th, jth and $j+1$th pixel. This is just a matter of re-arranging the equation to give

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

So we could, for every pixel in the next frame, if we know the current frame, calculate the entire row using this formula. However here's a few things,

- Considering the edges, the equation requires that I know the value to the left and the right of the jth pixel, and these may not be defined at the edges.
- I keep saying that these are *approximate* equations.
- I must know the entire current row of pixels to compute the next frames row of pixels.

The first item introduces us to the concept of *boundary conditions*, that will be discussed in a future article. For a simple demonstration code, you could simply assume that the boundary is fixed at some specific value of u regardles of time. These can be played with further to introduce new different effects as well.

The second item is where the mathematics comes in, as it turns out this method will actually break down badly depending on α. I might also like to point out here, that α must be given in a certain dimensional form in order to be physically correct, this is a matter of *dimensional analysis* and will be discussed in a future article. Simply putting in any value that gives a pretty looking result is useful for now.

The third item is where *initial conditions* come in to play. All we really need to know is the *first* frame of the animation, or physical effect, and we can compute every subsequent one from this starting point. Different starting points will result in different effects.

So to summarise, the choice of *initial condition* (the first frame), and the choice of *boundary conditions* will change what sort of effect we see, while α tends to have a more subtle role. Some basic examples of diffusion is given in Figure 4 as produced by Matlab ™

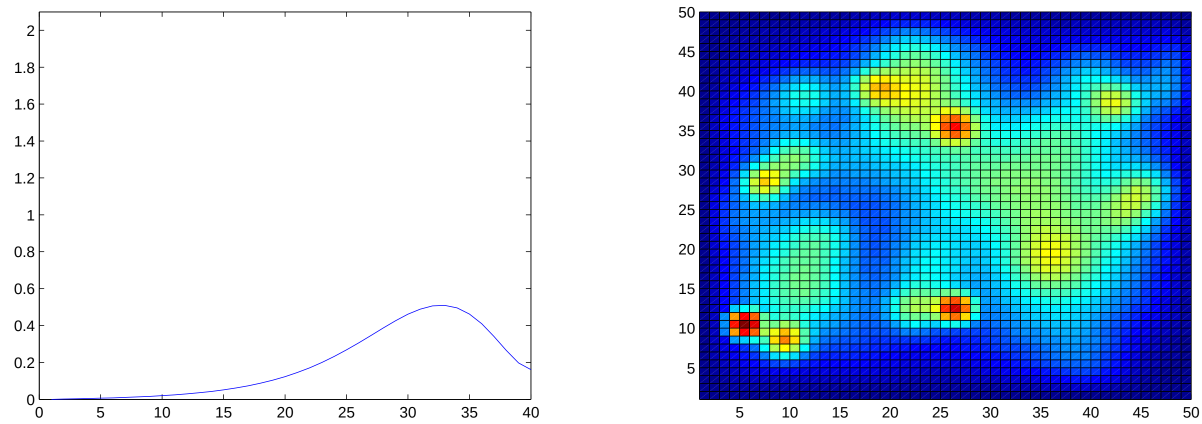


Figure 4: Output from Matlab

Here is the basic code one dimensional code in Matlab ™or Octave (not vectorised)

```
function [ A ] = Tsumea1( )

   alpha = 0.25;
   max_k = 500;
   max_j = 40;
   x = 1 : 1 : max_j;

   % This is our initial condition, that U is equal to zero
   U_k = zeros(1,max_j);

   U_k_plus_one = zeros(1,max_j);

   for k = 1 : 1 : max_k

   % Compute the boundary values for the next frame
   % I have fixed the left boundary at 0, and the right boundary
   % oscillates up and down
   U_k_plus_one(1,1) = 0;
   U_k_plus_one(1,max_j) = 1 + cos(k/50);

   % Compute all the non-boundary values of u for
   % the next frame
```

```
    for j = 2 : 1 : (max_j - 1)
       U_k_plus_one(1,j) = U_k(1,j);
       U_k_plus_one(1,j) = U_k_plus_one(1,j) + (2 * alpha) * U_k(1,j-1);
       U_k_plus_one(1,j) = U_k_plus_one(1,j) + (2 * alpha) * U_k(1,j+1);
       U_k_plus_one(1,j) = U_k_plus_one(1,j) - (4 * alpha) * U_k(1,j);
    end

    U_k = U_k_plus_one;

    % Do a simple plot of the value of U

    pause(0.001);
    plot(x,U_k);
    axis([0 40 0 2.1]);

    end

end
```

An extension into two dimensions in Matlab ™or Octave (vectorised)

```
function [ A ] = Tsumea2( )

    alpha = 0.1;
    max_k = 100;
    max_j = 50;

    x = 1 : 1 : max_j;
    y = 1 : 1 : max_j;
    U_k = zeros(max_j, max_j);

    % This is our initial condition, that U is equal to zero
    U_k = zeros(max_j,max_j);

    U_k_plus_one = zeros(max_j,max_j);

    for k = 1 : 1 : max_k

    % Compute the boundary values for the next frame
    % All edges are set to be zero
    U_k_plus_one(1,:) = 0;
    U_k_plus_one(max_j,:) = 0;
    U_k_plus_one(:,1) = 0;
    U_k_plus_one(:,max_j) = 0;

    % I also have a few random 'discontinuities' to spice up
    % the effect

    i = 2 + ceil(rand() * (max_j-2));
    j = 2 + ceil(rand() * (max_j-2));
    U_k(i-1:i+1,j-1:j+1) = 10;

    % Compute all the non-boundary values of u for
    % the next frame
    U_k_plus_one(2:max_j-1, 2:max_j-1) = U_k(2:max_j-1,2:max_j-1) + ...
    (2 * alpha) * (U_k(1:max_j-2,2:max_j-1) + U_k(3:max_j,2:max_j-1) + ...
    U_k(2:max_j-1,1:max_j-2) + U_k(2:max_j-1,3:max_j) - ...
    (4 * U_k(2:max_j-1,2:max_j-1)));

    U_k = U_k_plus_one;
```

```
    % Do a simple plot of the value of U

    pause(0.05);
    pcolor(U_k);
    caxis([0 10]);

    end

end
```