Noah Shore

MATH 528

Roberto Camassa

April 28, 2022

# Convection-Diffusion

In this report, I plan to discuss the (one-dimensional) convection-diffusion equation, its applications, as well as a handful of numerical methods used to solve this equation and its close relative- the heat equation. The form of this differential equation that is most familiar is written as follows:

$$\frac{\partial \Phi}{\partial t} = -u\frac{\partial \Phi}{\partial x} + k\frac{\partial^2 \Phi}{\partial x^2} + f$$

Where $\Phi$ (phi) is a function of x and t, u is the velocity of convection, k is a diffusion constant, and f is an outside source or sink of phi. In dimensions higher than one, we can insert a vector instead of a scalar for x that carries its coordinates.

The left-hand side of the equation represents the change of $\Phi$ over time, with respect to the terms on the right-hand side. The first term here is called the *convection* term, because it describes how phi is moving though the x spacial direction. This convection is subject to a velocity (u), which can either be constant or a function within space and time. The second term here is called *diffusion.* This represents the curvature of phi flattening out. The second derivative can be thought of as the magnitude of increase or decrease of phi, so it makes sense that a *curvier* function would have a larger second derivative, therefore increasing the rate change in phi over time. This diffusion is also subject to a constant *k,* which dictates the rate at which phi diffuses.

This equation is useful in modelling anything that undergoes convection, diffusion, or both. Temperature, for example, can be inserted in for phi in order to model heat transfer over time within a medium. Temperature is a function of both space and time, so we need to use the partial derivative embedded into the convection-diffusion to accurately describe heat transfer. Radiation, chemical species, and fluid mixture all utilize these same principles.

Both the scalar-transport and the heat equation are embedded into this formula, simply by selectively letting u or k be zero and getting rid of f. Starting with scalar-transport, if we let k = f = 0, then we end up with the form:

$$\frac{\partial \Phi}{\partial t} = -u\frac{\partial \Phi}{\partial x}$$

This simple formula can be used to transport a passive scalar through space over time.

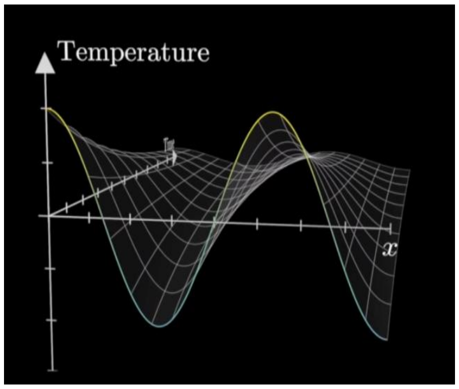
## Heat Flow in One Dimension

The heat equation, on the other hand, is obtained by setting u = 0 and substituting temperature (T) in for $\Phi$ :

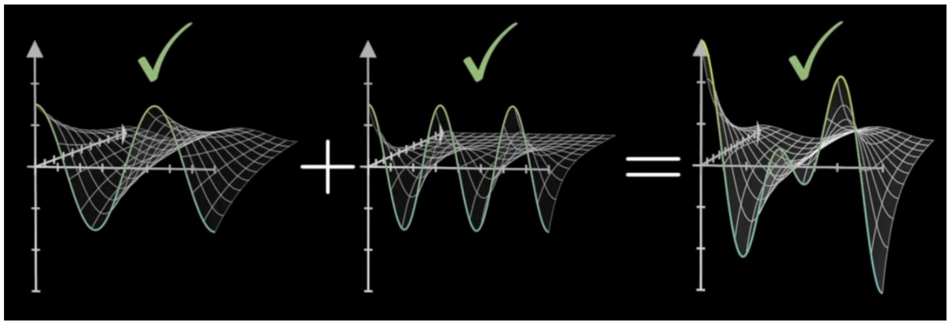
$$\frac{\partial T}{\partial t} = k\frac{\partial^2 T}{\partial x^2}$$

This famous equation was introduced by Joseph Fourier in 1822, along with his invaluable discovery of the Fourier series, with which he was able to approximate the solution to the heat equation for any temperature distribution using a combination of sine waves. The heat equation states that the change in temperature over time at each point in a body is proportional to its second spacial derivative. In other words, the temperature will increase faster at points where the function is more convex and decrease faster where it is more concave.

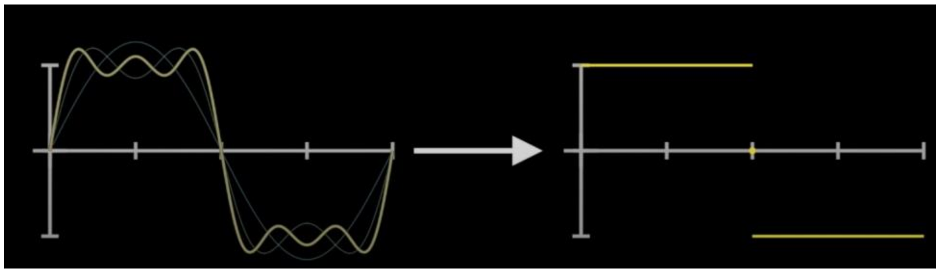
To explore how Fourier came up with his series solution, let us begin by looking at the temperature diffusing over a one-dimensional body, a metal rod for example. Because sine waves have the unique property that the second derivative is equal to the negative of the original wave, starting with an initial temperature distribution in the shape of a sine wave makes it easy to trace the path of the wave over time, as shown in the graph below[1].



Combining this with the fact that the sum of multiple solutions to a linear differential equation will always return another solution, we get that the temperature distribution of any combination of sine waves will give us an easy solution to the heat equation, as illustrated below:



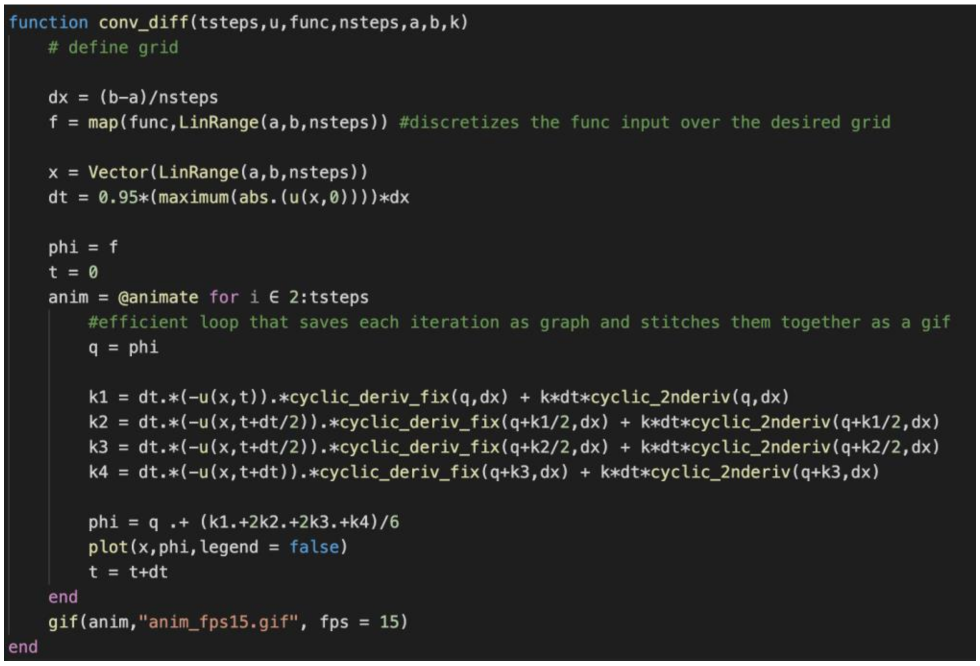
Fourier's most important discovery to solve the heat equation came from recognizing that any function can be represented as a series sum of sine and cosine waves. By assigning the initial temperature distribution as function of space, we can write that temperature as a Fourier series to solve the heat equation.



For any heat distribution in one dimension represented by $f(x)$, we can write a *Fourier Series* for that allows us to find $T(x,t)$ as a series sum, as we have detailed in MATH 528.

**Numerical Solutions to Convection-Diffusion**

Solving the convection-diffusion equation numerically is an interesting problem that utilizes repetitive algorithms to iterate through space and time. As a side to this project, I have written the convection-diffusion equation into Julia language. The solver utilizes a fourth order Runge-Kutta loop to advance over time (t), as well as second order finite difference equations to approximate the first and second spatial derivatives. The code for the solver is shown below:



```
function conv_diff(tsteps,u,func,nsteps,a,b,k)
    # define grid

    dx = (b-a)/nsteps
    f = map(func,LinRange(a,b,nsteps)) #discretizes the func input over the desired grid

    x = Vector(LinRange(a,b,nsteps))
    dt = 0.95*(maximum(abs.(u(x,0))))*dx

    phi = f
    t = 0
    anim = @animate for i ∈ 2:tsteps
        #efficient loop that saves each iteration as graph and stitches them together as a gif
        q = phi

        k1 = dt.*(-u(x,t)).*cyclic_deriv_fix(q,dx) + k*dt*cyclic_2nderiv(q,dx)
        k2 = dt.*(-u(x,t+dt/2)).*cyclic_deriv_fix(q+k1/2,dx) + k*dt*cyclic_2nderiv(q+k1/2,dx)
        k3 = dt.*(-u(x,t+dt/2)).*cyclic_deriv_fix(q+k2/2,dx) + k*dt*cyclic_2nderiv(q+k2/2,dx)
        k4 = dt.*(-u(x,t+dt)).*cyclic_deriv_fix(q+k3,dx) + k*dt*cyclic_2nderiv(q+k3,dx)

        phi = q .+ (k1.+2k2.+2k3.+k4)/6
        plot(x,phi,legend = false)
        t = t+dt
    end
    gif(anim,"anim_fps15.gif", fps = 15)
end
```

In the above screenshot, we have the main function, which takes in inputs tsteps, u, func, nsteps, a, b, and k.

- tsteps is the amount of frames that the loop animates over
- U is the input velocity. Conv_diff expects a function of two variables for u.

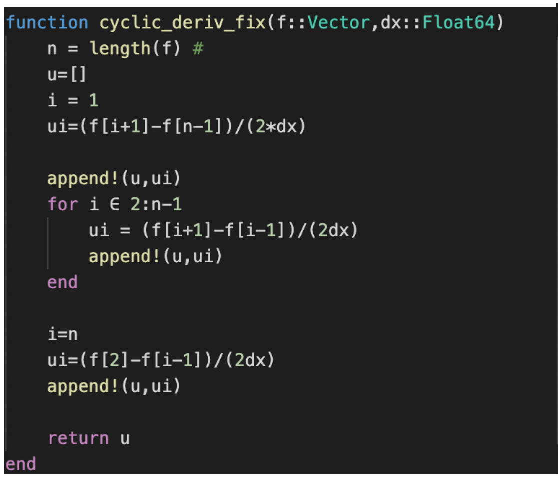
- Func is the function that is being convected and diffused.
- nsteps is the resolution over which func gets discretized
- a and b are the endpoints of the grid that is defined
- K is the constant of diffusion

Conv_diff first defines a grid using a, b and nsteps, and discretizes f over that grid using the map() function. It then enters an *animate* loop that calculates k1 through k4 and then puts them together to construct the next phase of phi, and it finished by graphing it and then advancing to the next timestep. This construction with k values comes directly from the fourth order Runge-Kutta method, which can be seen below[2]:

$$K_1 = hf(x_n, y_n)$$
$$K_2 = hf(x_n + \tfrac{h}{2}, y_n + \tfrac{k_1}{2})$$
$$K_3 = hf(x_n + \tfrac{h}{2}, y_n + \tfrac{k_2}{2})$$
$$K_4 = hf(x_n + h, y_n + k_3)$$
$$y_{n+1} = y_n + k_1/6 + k_2/3 + k_3/3 + k_4/6$$

The input for velocity (u) in this code is not a constant, but instead is written to expect u as a function of space and time.

The finite difference differentiators, cyclic_deriv_fix and cyclic_2nderiv are both defined separately, and can be seen in the following screenshots:

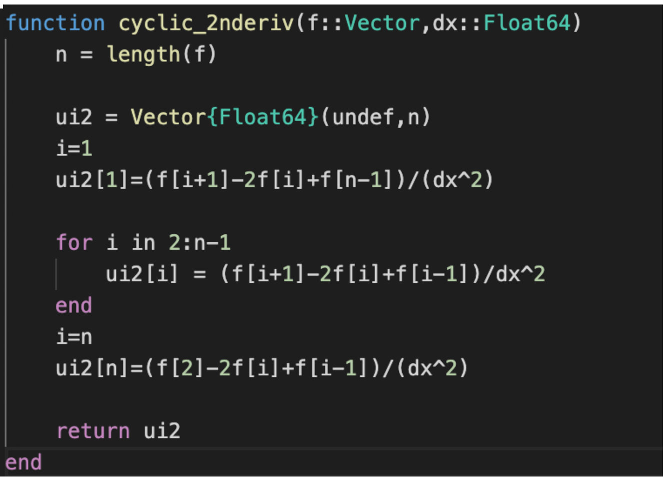


```
function cyclic_deriv_fix(f::Vector,dx::Float64)
    n = length(f) #
    u=[]
    i = 1
    ui=(f[i+1]-f[n-1])/(2*dx)

    append!(u,ui)
    for i ∈ 2:n-1
        ui = (f[i+1]-f[i-1])/(2dx)
        append!(u,ui)
    end

    i=n
    ui=(f[2]-f[i-1])/(2dx)
    append!(u,ui)

    return u
end
```



```
function cyclic_2nderiv(f::Vector,dx::Float64)
    n = length(f)

    ui2 = Vector{Float64}(undef,n)
    i=1
    ui2[1]=(f[i+1]-2f[i]+f[n-1])/(dx^2)

    for i in 2:n-1
        ui2[i] = (f[i+1]-2f[i]+f[i-1])/dx^2
    end
    i=n
    ui2[n]=(f[2]-2f[i]+f[i-1])/(dx^2)

    return ui2
end
```

Both the first and second derivative approximators are looping over the first and last entries of the inputted grid. This limits the solver to only have use over an initial condition that is periodic.

These methods for solving for the derivatives come from Second-Order Taylor Expansions. To approximate the derivative of $\Phi$ , we first write out the Taylor Series[3]:

$$\phi(x) = \phi(x_i) + (x - x_i)\left(\frac{\partial \phi}{\partial x}\right)_i + \frac{(x - x_i)^2}{2!}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i + \frac{(x - x_i)^3}{3!}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i + \cdots + \frac{(x - x_i)^n}{n!}\left(\frac{\partial^n \phi}{\partial x^n}\right)_i + H \ ,$$

Where ***H*** represents higher-order terms. Doing some reorganization of terms on the right- and left-hand sides of this expression, we arrive at the second order Taylor Expansion:

$$\left(\frac{\partial \phi}{\partial x}\right)_i = \frac{\phi_{i+1} - \phi_{i-1}}{x_{i+1} - x_{i-1}} - \frac{(x_{i+1} - x_i)^2 - (x_i - x_{i-1})^2}{2\,(x_{i+1} - x_{i-1})}\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i - \frac{(x_{i+1} - x_i)^3 + (x_i - x_{i-1})^3}{6\,(x_{i+1} - x_{i-1})}\left(\frac{\partial^3 \phi}{\partial x^3}\right)_i + H \ .$$

This equation will give the exact derivative $\frac{\partial \Phi}{\partial x}$ only if all terms are retained on the right-hand side. In our application this is impossible because we are trying to find the first derivative, so higher order derivatives are unknown. However,

$$\left(\frac{\partial \Phi}{\partial x}\right)_i \approx \frac{\Phi_{i+1} - \Phi_{i-1}}{x_{i+1} - x_{i-1}} = \frac{\Phi_{i+1} - \Phi_{i-1}}{2(\Delta x)}$$

Becomes a better and better approximation as $\Delta x$ gets smaller. This is the equation that is coded into the function *cyclic_deriv_fix,* as shown in the first screenshot above. By putting together $\frac{\partial \Phi}{\partial x}$ at both *i+(½)* and *i-(½),* we can find a similar method to approximate the second derivative:

$$\left(\frac{\partial^2 \phi}{\partial x^2}\right)_i \approx \frac{\phi_{i+1} + \phi_{i-1} - 2\,\phi_i}{(\Delta x)^2}$$

This equation is coded into the function *cyclic_2nderiv,* which can be seen in the second screenshot above.

Attached to this report, I have included a .gif file that was created by the convection-diffusion solver. In this gif, we see a Gaussian distribution undergoing convection and diffusion.

An interesting and somewhat disappointing observation I found while working with this style of solver was that any significant diffusion input into this function would cause a wild instability and make my computer self-destruct. I found that using finite difference for the second derivative would not allow me to go very far with the diffusion term. To solve the diffusion equation, I rewrote the solver using the Crank-Nicholson method, which fixes the stability and is quite accurate.

This method relies on writing the heat equation using the difference notation[4]:

$$q_{t+1}^{x} - q_{t}^{x} = a\left(q_{t}^{x+1} - 2q_{t}^{x} + q_{t}^{x-1}\right)$$

Where $q(x,t)$ now represents heat, and $a$ is the diffusivity. The Crank Nicholson equation is derived by averaging $q(t,x)$ with $q(t+1,x)$, as shown below:

$$q_{t+1}^{x} - q_{t}^{x} = \frac{a}{2}\left[\left(q_{t}^{x+1} - 2q_{t}^{x} + q_{t}^{x-1}\right) + \left(q_{t+1}^{x+1} - 2q_{t+1}^{x} + q_{t+1}^{x-1}\right)\right]$$

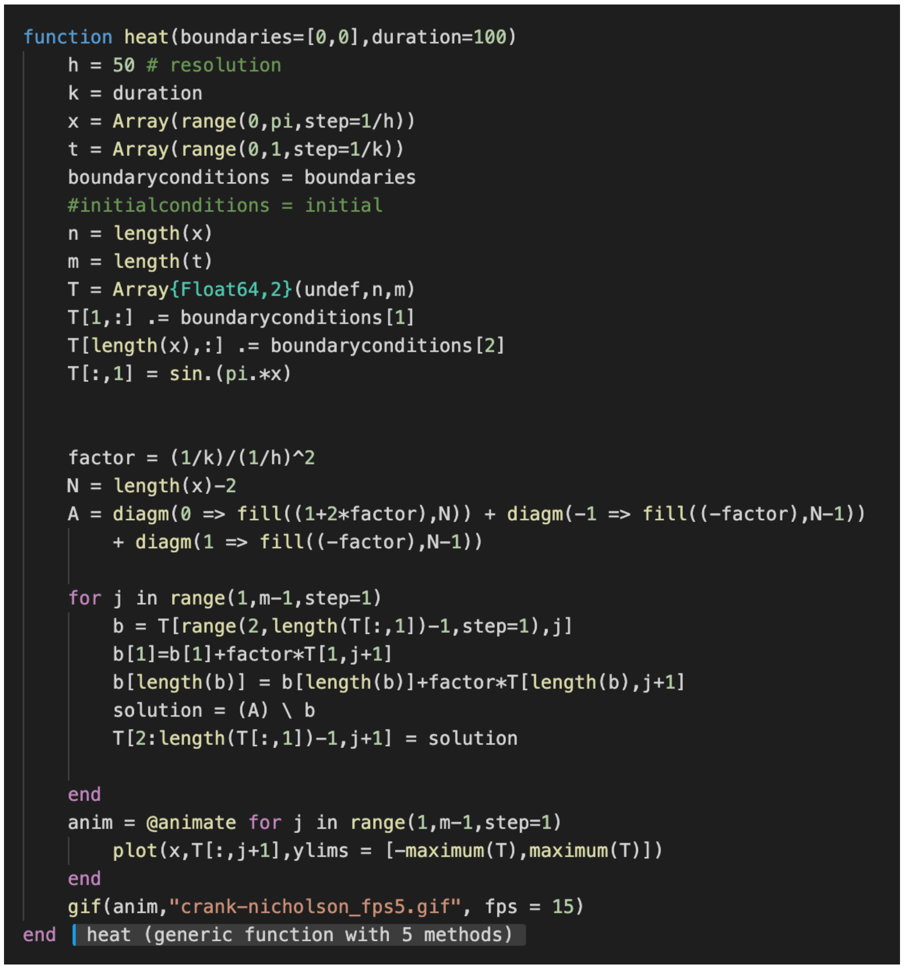
If we define $\alpha = \frac{a}{2}$ and rearrange terms to put the $q_{t+1}$ terms on the right-hand side of the equation, we arrive at:

$$-\alpha q_{t+1}^{x+1} + (1+2\alpha)q_{t+1}^{x} - \alpha q_{t+1}^{x-1} = \alpha q_{t}^{x+1} + (1-2\alpha)q_{t}^{x} + \alpha q_{t}^{x-1}$$

In order to solve for our q values, we put it into a tridiagonal matrix:

$$\begin{bmatrix} e_{\text{left}} & -\alpha & 0 & 0 & 0 \\ -\alpha & 1+2\alpha & -\alpha & 0 & 0 \\ 0 & -\alpha & 1+2\alpha & -\alpha & 0 \\ 0 & 0 & -\alpha & 1+2\alpha & -\alpha \\ 0 & 0 & 0 & -\alpha & e_{\text{right}} \end{bmatrix} \begin{bmatrix} q_{t+1}^{1} \\ q_{t+1}^{2} \\ q_{t+1}^{3} \\ q_{t+1}^{4} \\ q_{t+1}^{5} \end{bmatrix} = \begin{bmatrix} d_{t}^{1} \\ d_{t}^{2} \\ d_{t}^{3} \\ d_{t}^{4} \\ d_{t}^{5} \end{bmatrix}$$

Here, each d index represents all the terms at time *t* because that is the known quantity at each step. The vector of *q* values are what we are solving for, and $e_{right}$and $e_{left}$are the boundary temperatures that remain constant at every step. This linear system represents a computationally inexpensive string of equations that can be programmed into a computer, as shown below:



```julia
function heat(boundaries=[0,0],duration=100)
    h = 50 # resolution
    k = duration
    x = Array(range(0,pi,step=1/h))
    t = Array(range(0,1,step=1/k))
    boundaryconditions = boundaries
    #initialconditions = initial
    n = length(x)
    m = length(t)
    T = Array{Float64,2}(undef,n,m)
    T[1,:] .= boundaryconditions[1]
    T[length(x),:] .= boundaryconditions[2]
    T[:,1] = sin.(pi.*x)


    factor = (1/k)/(1/h)^2
    N = length(x)-2
    A = diagm(0 => fill((1+2*factor),N)) + diagm(-1 => fill((-factor),N-1))
        + diagm(1 => fill((-factor),N-1))

    for j in range(1,m-1,step=1)
        b = T[range(2,length(T[:,1])-1,step=1),j]
        b[1]=b[1]+factor*T[1,j+1]
        b[length(b)] = b[length(b)]+factor*T[length(b),j+1]
        solution = (A) \ b
        T[2:length(T[:,1])-1,j+1] = solution

    end
    anim = @animate for j in range(1,m-1,step=1)
        plot(x,T[:,j+1],ylims = [-maximum(T),maximum(T)])
    end
    gif(anim,"crank-nicholson_fps5.gif", fps = 15)
end  heat (generic function with 5 methods)
```

In this screenshot, the function *heat* constructs its initial matrix A using the $\alpha$ and the desired resolution, and it uses b to store the heat at each point along the x-axis.

T is the matrix that stores each time step as columns, and retains the boundary conditions throughout. For each time step, the program simply solves the linear system *Ax=b* for x, and then stores the calculated x vector into the appropriate column of T. Included in this report, I have attached a .gif file that was outputted by the *heat* function.

# Works Cited

[1]Sanderson, G. (n.d.). *Solving the Heat Equation*. Youtube. Retrieved April 28, 2022, from https://www.youtube.com/watch?v=ToIXSwZ1pJU

[2](2022, January 21). *Runge-Kutta 4th Order Method to Solve Differential Equation*. GeeksforGeeks. https://www.geeksforgeeks.org/runge-kutta-4th-order-method-solve-differential-equation/

[3]Ferziger, J., & Peric, M. (2012). *Computational Methods for Fluid Dynamics, 4th Edition*. Springer Science & Business Media.

[4](2000, December 26). *The Crank-Nicolson method*. Stanford Exploration Project; Stanford University. http://sepwww.stanford.edu/sep/prof/bei/fdm/paper_html/node15.html