M639 Non-Linear Waves Final Project
An Introduction to Finite Element Methods (FEM) and Application to the Non-Linear Schrödinger Equation

Seth Minor, Stefan Cline

12 December 2022

Abstract

This paper describes how one can construct numerical solutions to PDEs via the method of Finite Elements (FEM) in one and two dimensions, utilizing various types of boundary conditions. The eventual goal is to produce a functioning FEM model with which to simulate the dynamics of superfluid vortices on the surface of a torus. (All numerical results shown are supplemented by applicable code.)

The intention is that this paper serves as an introduction to FEM for those that may be familiar with other numerical approximation techniques (such as FDS), thereby reducing the overhead required to understand FEM at a basic level.

Index

Section 1. The first section of the paper develops a FEM solution for the 1D steady state heat equation (i.e., Laplace equation) with a forcing function. This introduces the various complexities and differences between the Finite Difference Scheme (FDS) methodology and FEM on an otherwise simple example.

Section 2. The same problem that was done in Section 1 is now done for the dynamical problem (instead of the steady state problem); it starts at some time t_0 and iterates to a some final time step $t_f > t_0$. We also include a periodic boundary problem for completeness of boundary types.

Section 3. A two-dimensional FEM solution to the heat equation using periodic boundary conditions is presented (illustrating a higher-dimensional problem).

Section 4. Attempts at numerical solutions to the NLS are made via the open-source software `FreeFem++`, developing using FEM concepts discussed in previous sections. Adaptive mesh refinement is discussed in the context of the NLS.

References and Appendices. Some helpful references as well as some of the MATLAB and `FreeFem++` code developed are given here (if not given earlier).

Note to the Reader: This document assumes the reader has a working understanding of MATLAB, has taken an introductory course in Partial Differential Equations (PDEs), and though likely not required, a course in Computational Methods for PDEs.

1 Finite Element Method (FEM) for the Steady State of the 1D Heat Equation with a Forcing Function

1.1 The Exact Solution to the Example Problem

We start by considering the 1D heat equation, where $f(x,t) = f(x)\ \forall t$ will be our *forcing function*:

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2} + f(x).$$

We then set the LHS equal to zero for our *steady state solution*, i.e., when the system exhibits no change with respect to time, giving us:

$$-\frac{f(x)}{k} = \frac{\partial^2 u}{\partial x^2}.$$

Now, for this example we'll be solving, we set $f(x) = k = 1$ which simplifies the above to:

$$u'' = -1. \tag{1}$$

To be clear, this is a forcing across the domain of a heat value of positive one. If we then impose Dirichlet boundary conditions $u(0,t) = u(1,t) = 0$ and define the finite domain $\mathcal{D} = [L,R] = [0,1]$, then we can analytically solve our now second order linear ODE:

$$\begin{aligned}
u'' &= -1 \\
u' &= -x + C \\
u(x) &= -\frac{1}{2}x^2 + Cx + E \\
BCs: &\begin{cases} 0 = u(0) = E & \to E = 0 \\ 0 = u(1) = -\frac{1}{2} + C & \to C = 1/2 \end{cases} \\
u(x) &= -\frac{1}{2}x^2 + \frac{1}{2}x.
\end{aligned}$$

We have that our *exact* steady state solution for our PDE on $\mathcal{D}$ is: $u(x,\infty) = -\frac{1}{2}x^2 + \frac{1}{2}x$. For the time being, we'll let $u_0 = u(x,0)$ be arbitrary.

1.2 Developing the FEM

Keeping the above from Section 1.1 in mind, we'll analytically build out an approximation to the above solution using FEM. First, we must *discretize* our domain. Because FEM are meant to be solved computationally (and therefore done quasi-discretely), we can no longer treat our domains as completely continuous. As such, we'll break up our domain as follows.

Figure 1: Discrete representation of x on the domain $\mathcal{D}$ with Dirichlet BCs.

We make an important note here: FEM domains are not always broken up nicely in the same step sizes, i.e., in the above figure we allow for $\Delta x \not\equiv$ constant in general (for this example to keep things simple we will however assume $\Delta x \equiv$ const.). Also, for the 1D case, an 'element' refers to the two points and the space joining them as shown above. When moving to 2D these elements can be triangles or squares; in 3D pyramids and cubes, etc.

With the domain discretized, we then label the points starting at the index of zero, noting that L would belong to the zero index and R to the final index:

$$\text{Dirichlet: } i = 0, \underbrace{1, ..., n-1}_{j}, n \tag{2}$$

The reason that the secondary index j exists is to capture the points we actually need to solve for. Because we know the value for $u(L,t)$ and $u(R,t)$, we don't need to compute them. However, we must be careful! If one wanted to do zero flux boundary conditions on both ends of the domain (i.e., homogeneous Neumann BCs), j would include the end points as there only the flux would be known, and *not* the value of $u(L,t)$ and $u(R,t)$.

Returning to Eq.(1), generalizing $f(x)$ and k again, and moving the negative gives us (and letting $1/k = \kappa$ for easier notation)

$$-u'' = \kappa f \tag{3}$$

Now for the bit that might feel a touch like magic. Because our domain is discrete, we can utilize a set of orthogonal basis functions multiplied by constants to approximate the true value of u. We'll call this approximation u_δ:

$$u \approx u_\delta(x) = \sum_{j=1}^{n-1} c_j \phi_j(x). \tag{4}$$

Now, because we don't want to restrict ourselves yet to exactly which defined functions will act as a basis, we'll multiply Eq.(3) by a test function, $v(x)$ that we maintain must also satisfy the boundary conditions (this is a point not to be overlooked or forgotten!). Namely, $v(0) = v(1) = 0$ for this example (note that in (5)-(8) we'll still be handling continuous functions),

$$-u''v = \kappa f v \qquad \Rightarrow \qquad \int_L^R -u''v \; dx = \int_L^R \kappa f v \; dx \tag{5}$$

Cleverly using integration by parts, and again recalling we're operating on our domain $\mathcal{D}$, observing the LHS of Eq.(5):

$$\int_0^1 -u''v \; dx = \underbrace{-u'v\Big|_0^1}_{0} + \int_0^1 u'v' \; dx \tag{6}$$

$$\int_0^1 -u''v \; dx = \int_0^1 u'v' \; dx. \tag{7}$$

Note, in Eq.(6) the value is zero because evaluated at 1 and 0, the test function $v(x) = 0$ to match our boundary conditions. Placing Eq.(7) back into Eq.(5) gives:

$$\underbrace{\int_0^1 u'v' \; dx = \int_0^1 \kappa f v \; dx}_{\text{Weak Form}}. \tag{8}$$

Pausing briefly, we note that if $u'(x_B) \neq 0$ and $v(x_B) \neq 0$ at the boundaries (where $x_B \equiv x$ boundary value), then the weak form fundamentally changes! Therefore, caution should be taken while building the weak form for general BCs.

The above mentioned *Weak Form* is a critical step when attempting to solve any PDE using more advanced software. The weak form is typically what is required as an input before the system can solve the PDE. Luckily, arriving at the weak form really is just a calculus exercise.

With the weak form established, we can now move to take the continuous weak form, and turn it into a discrete set of finite elements to be solved for. Plugging in u_δ for u (and remembering that c_j are constants):

$$\int_0^1 \sum_{j=1}^{n-1} c_j \phi_j'(x) v'(x) \; dx = \int_0^1 \kappa f(x) v(x) \; dx. \tag{9}$$

Here we'll employ another small bit of what might feel like magic: we make the assumption that the basis functions are the

test functions. Namely $v_i = \phi_i$, so that

$$\underbrace{\sum_{j=1}^{n-1}\left[\int_0^1 \phi_j'\phi_i'\,dx\right]}_{\mathbf{A}}\underbrace{c_j}_{\mathbf{c}} = \underbrace{\int_0^1 \kappa f_i\phi_i\,dx}_{\mathbf{b}}. \tag{10}$$

If we turn Eq.(10) into matrices to better solve our system (made more complex with two indices i and j in the summation), we'll have:

$$\mathbf{Ac} = \mathbf{b} \tag{11}$$

$$\mathbf{c} = \mathbf{A}^{-1}\mathbf{b}. \tag{12}$$

An astute reader may have now asked themselves: why are we solving this system for $\mathbf{c}$? Recall that $u(x,t) \approx u_\delta = \sum_{j=1}^{n-1} c_j\phi_j$, so now, if we simply have an expression for ϕ_j for all j, then we'll have an approximate solution for $u(x,t)$. Nicely, we get to pick our choice of ϕ_j, which we'll choose as *tent functions* (the reason for this choice will be clear later on). These tent functions will be 1 at the discrete point indexed by j or i, and zero everywhere else. As such, they can be written as (note that the denominator is simply $\Delta x \equiv$const. if the domain is evenly spaced):

$$\phi_n(x) = \begin{cases} \dfrac{x - x_{n-1}}{x_n - x_{n-1}}, & \text{if } x_{n-1} \le x < x_n, \\ \dfrac{x_{n+1} - x}{x_{n+1} - x_n}, & \text{if } x_{n-1} \le x < x_n, \\ 0, & \text{otherwise} \end{cases}$$

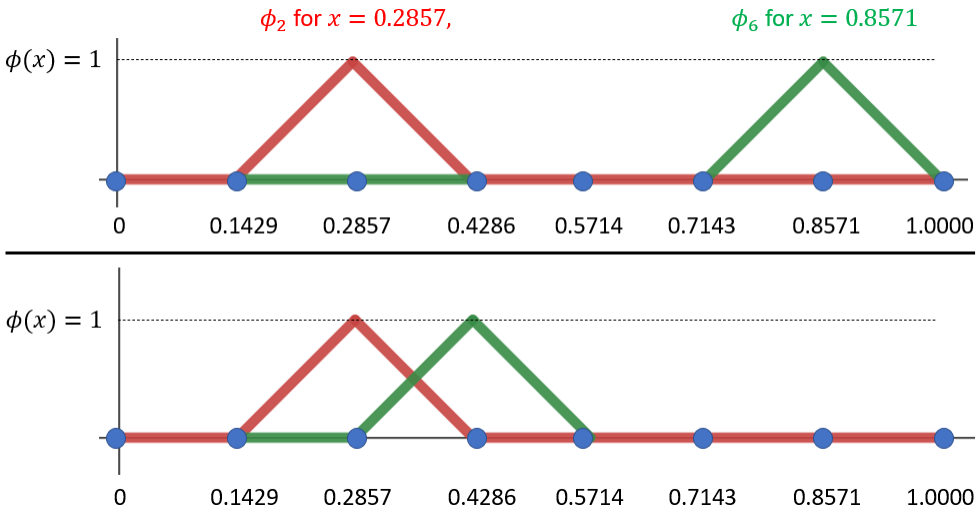


Figure 2: Tent Functions on an evenly discretized domain from 0 to 1 with 8 points. The top image has no overlap between them. The bottom shows possible overlap to the side. (A third not shown possibility is the indices could be the same meaning two tent functions would be on top of one another.)

With the height of the tent functions being 1, and the width being Δx, we can quickly observe that the derivative of these functions is either $1/\Delta x$ or $-1/\Delta x$, i.e., $\phi_n' = \pm 1/\Delta x$ or $\phi_n' = 0$ depending.

As such, we now have all of the tools we'll need to fully expand the system in Eq.(11). We pick Eq.(11) and not Eq.(12) as MATLAB can handle the matrix inversion, so we won't have to analytically invert the matrix. Hence, for elements

indexed as a_{ij}:

$$\mathbf{A} = \underbrace{\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1,n-1} \\ a_{21} & \ddots & & a_{2,n-1} \\ \vdots & & \ddots & \vdots \\ a_{n-1,1} & \dots & \dots & a_{n-1,n-1} \end{bmatrix}}_{\text{Index: } a_{ij}}, \qquad \mathbf{c} = \underbrace{\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix}}_{c_i}, \qquad \mathbf{b} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}}_{b_i} \tag{13}$$

Because of our choice of our tent function, our system of matrices will be (letting $\delta = \Delta x$ for easier notation):

$$\begin{bmatrix} 2/\delta & -1/\delta & 0 & 0 & \dots & 0 \\ -1/\delta & 2/\delta & -1/\delta & 0 & \dots & 0 \\ 0 & -1/\delta & 2/\delta & -1/\delta & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1/\delta & 2/\delta & -1/\delta \\ 0 & \dots & 0 & 0 & -1/\delta & 2/\delta \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_{n-1} \end{bmatrix} \tag{14}$$

Recall that $a_{ij} = \int_0^1 \phi_i' \phi_j' \, dx$ which should make the calculations in $\mathbf{A}$ matrix more straightforward (if not just a touch obnoxious). For our various b_i we nicely chose $f(x) = \kappa = 1$, therefore, the area under each tent function is the same, namely $\int_0^1 \kappa f b_i \, dx = \delta$. This yields:

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ \vdots \\ c_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} 2/\delta & -1/\delta & 0 & 0 & \dots & 0 \\ -1/\delta & 2/\delta & -1/\delta & 0 & \dots & 0 \\ 0 & -1/\delta & 2/\delta & -1/\delta & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1/\delta & 2/\delta & -1/\delta \\ 0 & \dots & 0 & 0 & -1/\delta & 2/\delta \end{bmatrix}^{-1}}_{\mathbf{A}^{-1}} \begin{bmatrix} \delta \\ \delta \\ \vdots \\ \vdots \\ \delta \end{bmatrix} \tag{15}$$

MATLAB can easily handle $\mathbf{A}^{-1}\mathbf{b}$ as $\mathbf{A}\backslash\mathbf{b}$, given its *tridiagonal* nature which should allow for quick computations.

We're now in a good position to give an approximation for $u(x, \infty)$. We have values for c_i and we know all of our functions ϕ_i, so that:

$$u_\delta(x) = \sum_{i=1}^{n-1} c_i \phi_i. \tag{16}$$

Our choice of tent functions will really shine here! Because they are 1 at their indexed value of x_i, and zero everywhere else, this means our $u_\delta(x)$ if written as a vector, is simply $\mathbf{c}$ plus the BCs! Just to make it completely clear:

$$c_1 \underbrace{[1, 0, ..., 0]}_{\phi_1} + c_2 \underbrace{[0, 1, 0, ..., 0]}_{\phi_2} + \ldots + c_{n-1} \underbrace{[0, ..., 0, 1]}_{\phi_j} = [c_1, c_2, ..., c_{n-1}] \rightarrow u_\delta(x) = [u(L), c_1, c_2, ..., c_{n-1}, u(R)]$$

This can be seen nicely once the MATLAB code runs for a small number of elements.

Lastly, we note that the idea of the image on the right in Figure (4) shows how a combination of tent functions times a value of c when summed create the generated curve in Figure (3).

1.3 Numerical Results for the 1D Steady State

Here we have selected a small number of elements and nodes, showing how the function is discretely compiled to give a good approximation to the exact solution.

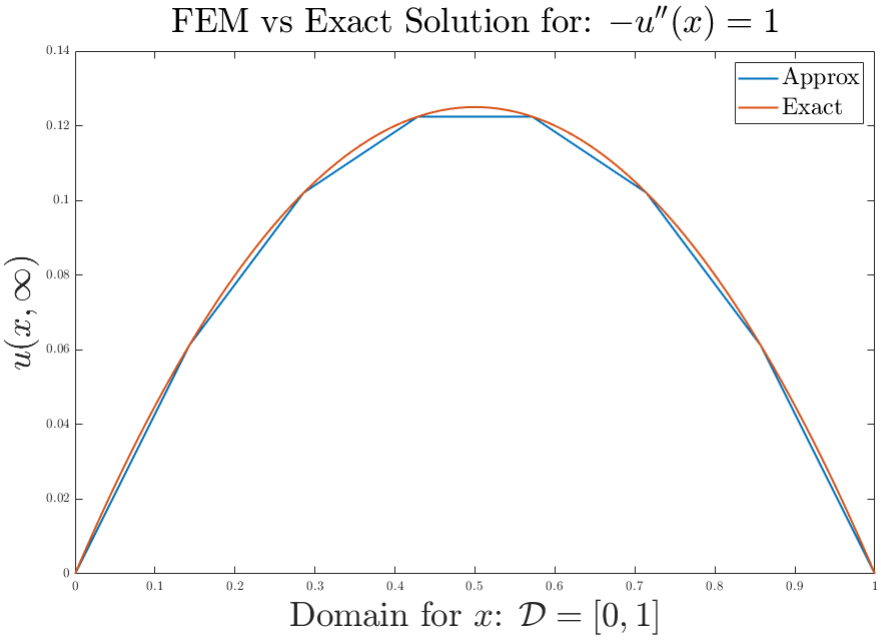


Figure 3: The resulting approximation plotted against the exact solution for $-u'' = 1$ using 8 discrete points.

```
% Simple FEM Example for the 1D Laplace Equation

% Solving the equation:
% -u''=f(x)     Note! f here is not an IC but a forcing function!
% as a steady state solution we've assumed the u_t=0 and hence are just
% solving the Laplace equation i.e. the 1D heat Eq. steady state

% BCs: u(0)=0, u(1)=0 <-- double sink at boundaries, expect steady to
% go to zero at the boundaries

% Will eventually solve: c = A\b
%                      : u_h=sum(c_i*phi_i)

close all; clear all; clc;
set(groot, 'defaultAxesTickLabelInterpreter','latex');
set(groot, 'defaultLegendInterpreter','latex');
set(groot, 'defaultTextInterpreter','latex');
lw = 1.5; % plot line widths
FS = 26;  % plot larger fonts
fs = 18;  % plot smaller fonts

% BC Types
uL = 0;  % u(L) =
uR = 0;  % u(R) =
%uPL = 0; % u'(L) =
%uPR = 0; % u'(L) =

n = 6;                      % density of x axis, Note:# of line elements = n+1
L = 0;                      % left boundary in 1D
R = 1;                      % right boundary in 1D
x = linspace(L,R,(n+2)); % 1D domain
xe = linspace(L,R,1000); % Exact Solution domain
dx = x(2)-x(1);             % delta between evenly spaced domain values
f = ones(1,length(x));   % making the Forcing Function f(x)=1
                            % NOTE! If you change this, the code must be
                            % changed drastically for the b_vals function!
f(1) = [];                  % making the f vector size n
f(end) = [];                % making the f vector size n
f = f';                     % making into column vector
b_i = 0.*f;                 % empty b_i matrix

% Need to solve the matrix equation in the form:
% -->   u = inv(Aij)*b
% Setting up loops to calculate all of the values of bi and Aij
% A must be an nxn matrix, also note we don't need to solve for the BCs
% as they're given to us (again n+2 is the full size of the discrete
% domain)
A_ij = zeros(n,n);

for ii = 1:n % rows
    for jj = 1:n % columns
        A_ij(ii,jj) = A_vals(ii,jj,dx); %filling out our A matrix
    end
        b_i(ii) = b_vals(dx); %filling out the b column vector
end

c = A_ij\b_i; % using the \ instead of inv for efficiency
              % solving c gives us the c in
              % u_h = sum(c_j*phi_j)

```

```
% Now, complete the approximation of the function u_h
% u_i = c_i*phi_i
% This is a bit tricky. Becuase we're dealing with discrete hat functions,
% if we simply look at each phi function, it's one at that value of x_i,
% but otherwise zero. Ex: c_3*phi_3 = [0,0,c_3,0...,0,0]. Then, summing
% them all together is adding 1xn matricies that are zero everywhere but at
% that index, so we simply get u = [c1,c2,c3,...,cn]
% Note if we don't use hat functions, this simple switch doesn't work.
u = c;

u = [uL;u;uR]; % Adding our two BCs back to build the full u(x) for the full
               % original x domain

% Plotting the final result and comparing to the exact
plot(x,u,'LineWidth',lw)
hold on
uexact = -1/2*xe.^2+1/2.*xe;
plot(xe,uexact,'LineWidth',lw)
title("FEM vs Exact Solution for: $-u''(x)=1$",'FontSize',FS)
xlabel("Domain for $x$: $\mathcal{D}=[0,1]$",'FontSize',FS)
ylabel("$u(x,\infty)$",'FontSize',FS)
legend('Approx','Exact','FontSize',fs)



%%
%%%%%%%%%%%% Functions for Aij and bi %%%%%%%%%%%%

function Aij = A_vals(ii,jj,dk)
    % note that A_ij = int^R_L (v'_i*v'_j) dx (v and phi here are the same)
    %
    if ii == jj
        Aij = 2/dk;
    elseif abs(ii-jj) == 1
        if ii > jj
            Aij = -1/dk;
        else % ii < jj
            Aij = -1/dk;
        end
    elseif abs(ii-jj)>=2
        Aij = 0;
    end
end

function bi = b_vals(dx)
    % note that this is the simplest case
    % we set f(x)=1, so f*phi --> 1*phi --> int (phi)
    % for all tophat functions int phi is just 1*dx = dx
    bi = dx;
end
```

1.4 Deltas that Stand Out Between FDS and FEM

For those familiar with Finite Difference Schemes (FDS), there are a few striking differences.

1. FDSs don't use any continuous alterations of the the original PDE. This is in stark contrast to FEM where a function doing quite a bit of overhead work is in fact a continuous (albeit abused) basis function.
2. The addition of a term in a FDS that isn't a time derivative term, especially if constant or evaluated at that point, can be handled rather easily. In an FEM setup however, the addition of a term drastically adjusts the weak form, and those changes must be carried algebraically through to ensure the various matrices are built correctly.
3. There is a clear trade-off between ease and time on the one hand, and computational speed on the other. A FDS is much easier to set up; however, it may be slower in time (c.f. a quick evaluation of matrices with iterative solving). In contrast, building the FEM may take much longer and be more prone to algebraic errors (from the hand of the mathematician), but once established, is likely a much more powerful tool. (This will become more apparent when it is solved on a more complex geometry than a 1D line.)
4. FDS cannot be easily applied to odd geometries, whereas FEM can be applied to effectively any arbitrary domain.

2 Time Iteration of the 1D Heat Equation using FEM

2.1 General Setup Using Euler

For those familiar with FDS, this will be a nice break from the new complexity of FEM. There are several techniques for performing time integration, but the most approachable is coupling a FEM-like spatial discretization on each spatial step while continuing to use finite-differencing in time. For simplicity's sake, we'll be performing Forward Euler for temporal integration, but if used practically, a better time stepping method should be utilized.

We start with the 1D heat equation just like we did in Section 1, and keeping the same set up ($k = f = 1$, and $u(L) = u(R) = 0$) and using the shorthand notation for time and space derivatives as subscripts of the respective variable:

$$\frac{\partial u}{\partial t} = k\frac{\partial^2 u}{\partial x^2} + f(x) \rightarrow u_t = ku_{xx} + f.$$

We now note that we'll want to keep our u_t term instead of setting it to zero. We again recall that the steady state solution to this problem is (which we'll use later):

$$u(x, \infty) = -\frac{1}{2}x^2 + \frac{1}{2}x. \tag{17}$$

Then, similarly to Section 1, we'll multiply by a test function, $v(x)$:

$$vf + vku_{xx} = vu_t \tag{18}$$

$$\int_L^R [vf + vku_{xx}]\ dx = \int_L^R vu_t\ dx \tag{19}$$

$$\int_L^R vf\ dx + k\int_L^R vu_{xx}\ dx = \int_L^R vu_t\ dx. \tag{20}$$

Performing integration by parts for the right term on the LHS gives (noting that our boundaries again are zero), we get:

$$\int_L^R vf\ dx + k\left[\underbrace{u_x v\Big|_L^R}_{0} - \int_L^R u_x v_x\ dx\right] = \int_L^R vu_t\ dx \tag{21}$$

$$\int_L^R vf\ dx - k\int_L^R u_x v_x\ dx = \int_L^R vu_t\ dx. \tag{22}$$

As before, we'll assume some form for our function u_δ, but now with a twist. We'll want our c values to change as time progresses. Therefore, along a fixed x_p, we'll have a unique function, $c_p(t)$ (where $p \equiv$ some discretized point of the spatial domain):

$$u_\delta(x, t) = \sum_{i=1}^{n-1} c_i(t)\phi_i(t). \tag{23}$$

We attempt a graphical aid of Eq.(23) in Figure(4). Continuing, we place u_δ into Eq.(22) (ignoring the sum signs for notational convenience) and again assume that our test function v is the same as our previous tent functions, ϕ.

$$\int_L^R \phi_i f_i\ dx - k\int_L^R c_i\phi_i'\phi_j'\ dx = \int_L^R \phi_i\phi_j\dot{c}_i\ dx \tag{24}$$

$$\underbrace{\int_L^R \phi_i f_i\ dx}_{\mathbf{F}_i} - k\underbrace{\int_L^R \phi_i'\phi_j'\ dx}_{\mathbf{K}_{ij}}\ \dot{c}_i^n = \underbrace{\int_L^R \phi_i\phi_j\ dx}_{\mathbf{M}_{ij}}\ \dot{c}_i^n \tag{25}$$

$$\mathbf{F} - k\mathbf{K}\mathbf{c}^n = \mathbf{M}\dot{\mathbf{c}}^n. \tag{26}$$

In the last two lines above, note that $c_i \rightarrow c_i^n$ and $\dot{c}_i \rightarrow \dot{c}_i^n$. This is because those individual collection of points then become the entire c vector at that time-step, n. Also, our matrices K and M are shorthand for 'scanning' through the possibilities for i and j. (As an aside, if it helps, $\mathbf{K}$ is the same as $\mathbf{A}$ from before.)

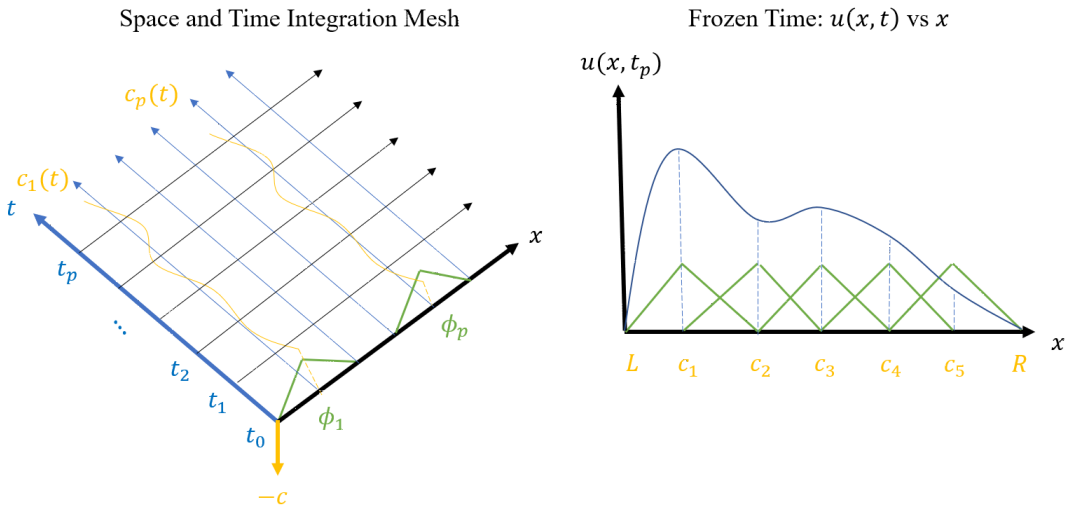


Figure 4: The values of c that change as a function of time build, in conjunction with tent functions, approximations of u at some time step t. The image on the right shows an example of solving u_n at a single instance of time t_n. In Section 1, this was the final solution to produce the steady state.

Here is where we switch to FDS and invoking the power of (or lack thereof) Euler. By picking

$$\dot{\mathbf{c}}^n = \frac{\mathbf{c}^{n+1} - \mathbf{c}^n}{\Delta t} \tag{27}$$

and then placing this into Eq.(26) and solving algebraically, one gets

$$\mathbf{F} - k\mathbf{K}\mathbf{c}^n = \mathbf{M}\left[\frac{\mathbf{c}^{n+1} - \mathbf{c}^n}{\Delta t}\right] \tag{28}$$

$$\mathbf{c}^{n+1} = \mathbf{M}^{-1}\Delta t(\mathbf{F} - k\mathbf{K}\mathbf{c}^n) + c^n. \tag{29}$$

Therefore, keeping our boundary conditions in mind (as we assumed them to be zero, i.e. homogeneous Dirichlet BCs),

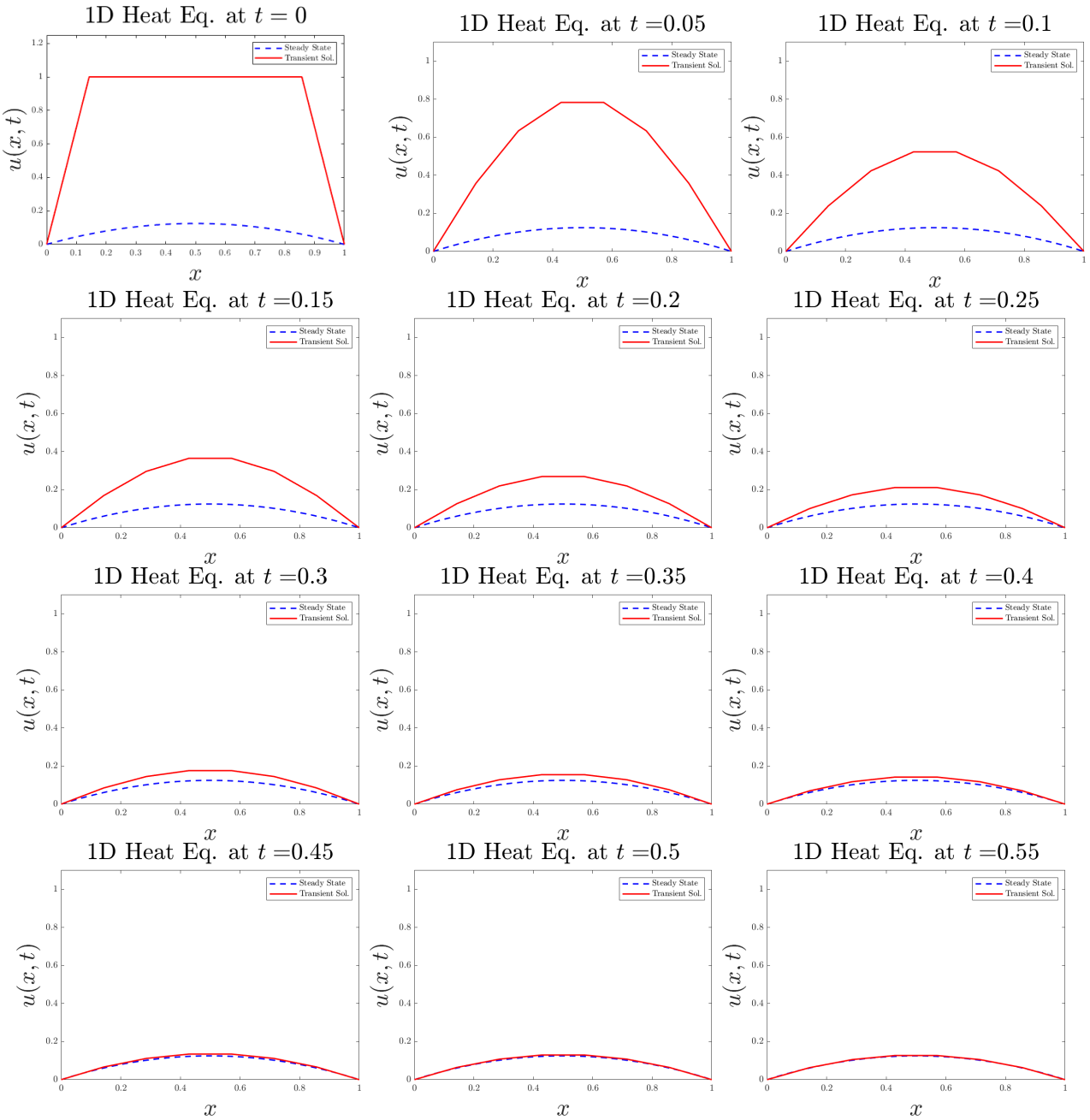
$$u_{n+1} = [u(L), c^{n+1}, u(R)]. \tag{30}$$

This can be quickly and easily iterated in MATLAB to produce quick and clean solutions (see Section 2.1.1).

2.1.1 Numerical Results for the 1D Heat Equation using FEM for Space and FDS for Time

Below are the results if we assume an initial condition of $u_0 = [0, 1, 1, ..., 1, 1, 0]$. Again we note that here we've assumed:

$$k = 1, \qquad f(x) = 1, \qquad u(L) = u(R) = 0, \qquad \mathcal{D} = [0, 1], \qquad t \in [0, 0.55]$$



```
%% 1D Heat Equation

close all; clear all; clc;
set(groot, 'defaultAxesTickLabelInterpreter','latex');
set(groot, 'defaultLegendInterpreter','latex');
set(groot, 'defaultTextInterpreter','latex');

lw = 1.5; % plot line widths
FS = 26;  % plot larger fonts
fs = 18;  % plot smaller fonts

% BC Types (picked Dirichlet BCs)
uL = 0;  % u(L) =
uR = 0;  % u(R) =

k = 1;                      % constant from the PDE
n = 6;                      % density of x axis, Note:# of line elements = n+1
L = 0;                      % left boundary in 1D
R = 1;                      % right boundary in 1D
x = linspace(L,R,(n+2)); % 1D domain
xs = linspace(L,R,1000); % Stead State Solution domain
dx = x(2)-x(1);             % delta between evenly spaced domain values
f = ones(1,length(x));    % making the Forcing Function f(x)=1
                            % NOTE! If you change this, the code must be
                            % changed drastically for the b_vals function!
f(1) = [];                  % making the f vector size n
f(end) = [];                % making the f vector size n
f = f';                     % making into column vector
ts = 0; tf = .55;           % start and end time points
Den = 1000000;              % density of the time-steps
t = linspace(ts,tf,Den); % creating the time axis
dt = t(2)-t(1);             % delta t
Fmax = max(f);              % for nice plot limits

% Building the Matricies that won't change every iteration
%      K and M as defined in the paper
% Note that our nice choice of f means that it's simply
%      a comlumn vector filled with del_x (not always the case)
M = zeros(n,n);
K = M;

for i = 1:n
    for j = 1:n
        if i == j
            M(i,j) = 2*dx/3;
            K(i,j) = 2/dx;
        elseif abs(i-j) == 1
            M(i,j) = dx/6;
            K(i,j) = -1/dx;
        else
            M(i,j) = 0;
            K(i,j) = 0;
        end
    end
    f(i) = dx;
end

% The initial condition and steady state
% note that because we have fixed BCs, we'll solve n-2 points
% if we had Neumann BCs, we'd need to solve n points
```

```
% i.e., be careful here!

u_old = ones(1,n);          % IC of zeros at the boundaries, one elsewhere
u_new = zeros(1,n);         % Empty matrix to hold the n+1 solution in time
usted = -1/2*xs.^2+1/2.*xs; % plotting the steady state solution

% Completing the time integration

% Plotting the initial condition
figure
plot(xs,usted,'--b','LineWidth',lw)      % steady state solution
hold on
plot(x,[0,u_old,0],'-r','LineWidth',lw)  % initial condition
xlabel('$x$','FontSize',FS)
ylabel('$u(x,t)$','FontSize',FS)
title("1D Heat Eq. at $t=0$",'FontSize',FS)
ylim([0,1.25])
xlim([0,1])
legend('Steady State','Transient Sol.','location','northeast')
hold off
pause(.1)
count = 1;
timecount = ceil(length(t)/10);
% The actual integration
for T = t(2:end) % this "T" acts like the 'nth' timestep in the paper
        u_new = M\(dt.*(f-k.*K*u_old')) + u_old'; % finding u_{n+1}
        count = count+1;
        if mod(count,timecount) == 0
            plot(xs,usted,'--b','LineWidth',lw) % plotting the steady state
            hold on
            plot(x,[0;u_new;0],'-r','LineWidth',lw)
            xlabel('$x$','FontSize',FS)
            ylabel('$u(x,t)$','FontSize',FS)
            title("1D Heat Eq. at $t=$"+T,'FontSize',FS)
            ylim([0,(Fmax+Fmax/10)])  % keeping the frame 'steady'
            xlim([0,1])
            legend('Steady State','Transient Sol.','location','northeast')
            hold off
            pause(0.1)
            Tstr = num2str(T);
            %filename = strcat("C:\Users\scline8247\Pictures\heat1d_dirch\
                Heat_to_steady_t_",Tstr,".png");
            %filename = convertCharsToStrings(filename);
            %exportgraphics(gcf,filename)
        end
        u_old = u_new'; % making the updated frame become the 'old' frame
end
```

2.2 A Quick Example with 1D FEM Periodic Boundary Conditions

As a starting note, if you did not read Section 2.1 we recommend doing so or this section might be a bit too brisk. In this section, we present an example of periodic boundary conditions for the 1D case before moving on to the more complicated 2D case. Hopefully this boundary adjustment isn't too painful as it shouldn't be as taxing as the previous material.

Here, we will not have Dirichlet or Neumann boundary conditions because we won't really have a boundary. So what will we have? Effectively, we'll impose the following condition: *the final point's value is equal to the first point's value in our 1D domain.* Going back to developing the weak form of the PDE from Eq.(21):

$$\int_L^R vf\ dx + k\left[\underbrace{u_x v\Big|_L^R}_{?} - \int_L^R u_x v_x\ dx\right] = \int_L^R vu_t\ dx$$

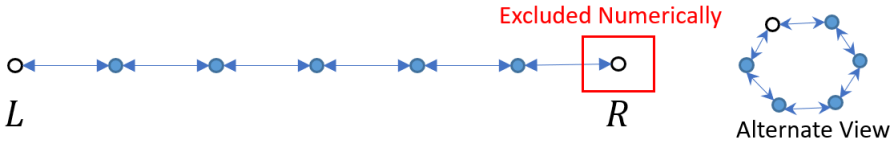
Note that we treat the domain such that $L = R$ (see the black circle white fill dots below). Hence, it doesn't matter what the $u_x v$ term contains as that expression is always zero (the right minus the left is zero, since the two are equal!). Hence, we again achieve:

$$\int_L^R vf\ dx - k\int_L^R u_x v_x\ dx = \int_L^R vu_t\ dx.$$

As for our function u_δ that's approximating our solution, we make a note about the indexing that is critical to then solving with periodic boundary conditions.

$$u_\delta(x,t) = \sum_{i=0}^{n-1} c_i(t)\phi_i(t).$$

Note that $i = 0$ and we go until $n-1$. We *do not* continue on to n as the nth point already exists in our domain which is L, because again, $L = R$.



How are the matrices $\mathbf{K}$ and $\mathbf{M}$ adjusted? Recall from Eq.(29)

$$\mathbf{c}_{n+1} = \mathbf{M}^{-1}\Delta t(\mathbf{F} - k\mathbf{K}\mathbf{c}_n) + \mathbf{c}_n$$

Because we need the boundaries to 'talk' to each other still (because our domain isn't ending, it's wrapping around), we extend the values and arrive at the following matrix (note the same thing is done for $\mathbf{M}$, i.e. $\mathbf{M}_{n-1,1} = \mathbf{M}_{1,n-1} = \int_L^R \phi_1\phi_2\ dx$):

$$\mathbf{K}_{ij} = \begin{bmatrix} 2/\delta & -1/\delta & 0 & 0 & \dots & -1/\delta \\ -1/\delta & 2/\delta & -1/\delta & 0 & \dots & 0 \\ 0 & -1/\delta & 2/\delta & -1/\delta & \dots & 0 \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & -1/\delta & 2/\delta & -1/\delta \\ -1/\delta & \dots & 0 & 0 & -1/\delta & 2/\delta \end{bmatrix}, \qquad \text{where } \mathbf{K} \equiv n \times n, \text{ as } i,j \in \{0,1,...,n-1\}$$

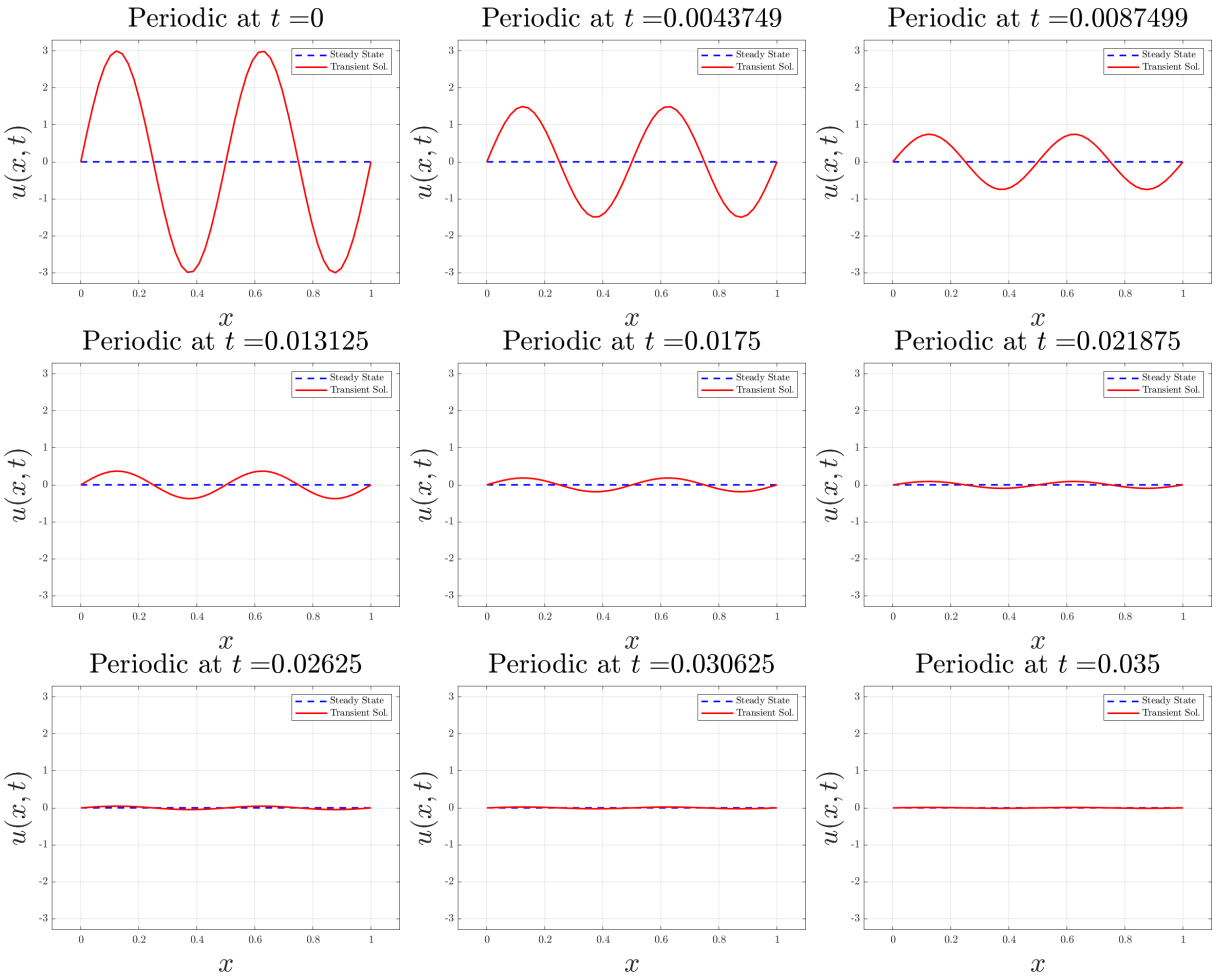
Lastly, we force the last point, $u(R)$, to be the same as the first point, $u(L)$, after computing the solution at time t_n (i.e., $\mathbf{c}_{n+1}$):

$$\mathbf{u}_{n+1} = [\mathbf{c}_{n+1}, u(R) = u(L)].$$

2.2.1 Numerical Results for the 1D Periodic Boundary Conditions

For this example, we'll make the domain denser and set $n = 48$, and again let $L = 0$ and $R = 1$. Also, for an initial condition, we need to pick something that is periodic. Therefore, we'll let $u_0 = 3\sin(4\pi x)$ (which is zero at both $L = x = 0$ and $R = x = 1$) to nicely see the behavior, and have $f = 0$ so that we're not dealing with a forcing function. This also has the nice benefit that the steady state solution is $u(x, \infty) = 0$ (the average of an n-period sine wave); thereby, allowing us to easily verify if our solution is behaving qualitatively as expected.

Below are the results from the code posted after the images.



```
%% 1D Heat Equation (Periodic)

close all; clear all; clc;
set(groot, 'defaultAxesTickLabelInterpreter','latex');
set(groot, 'defaultLegendInterpreter','latex');
set(groot, 'defaultTextInterpreter','latex');

lw = 1.5; % plot line widths
FS = 26;  % plot larger fonts
fs = 18;  % plot smaller fonts

% BC Types (picked Dirichlet BCs)
%uL = 1;  % u(L) =
%uR = 1;  % u(R) =

k = 1;                      % constant from the PDE
n = 48;                     % density of x axis, Note:# of line elements = n+1
L = 0;                      % left boundary in 1D
R = 1;                      % right boundary in 1D
x = linspace(L,R,(n+2)); % 1D domain + BCs
x(end) = [];                % kill the last point as it's just the first point
                            % above line for periodic BCs
xs = linspace(L,R,1000); % Stead State Solution domain
dx = x(2)-x(1);             % delta between evenly spaced domain values
f = zeros(1,length(x));  % making the Forcing Function f(x)=0 to have easy convergence to
    0
                            % NOTE! If you change this, the code must be
                            % changed drastically for the b_vals function!
%f(1) = [];                 % we want to keep the size of $f$ to be n+1 now
f(end) = [];                % periodic BCs
f = f';                     % making into column vector
ts = 0; tf = .035;          % start and end time points
Den = 400000;               % density of the time-steps
t = linspace(ts,tf,Den); % creating the time axis
dt = t(2)-t(1);             % delta t
view_max = 3;               % for nice plot limits
view_min = -3;              % for nice plot limits

% Building the Matricies that won't change every iteration
%      K and M as defined in the paper
% Note that our nice choice of f means that it's simply
%      a comlumn vector filled with 0 (not alwasy the case)
M = zeros(n+1,n+1);
K = M;

for i = 1:(n+1) % n+1 for periodic BC
    for j = 1:(n+1) % n+1 for periodic BC
        if i == j
            M(i,j) = 2*dx/3;
            K(i,j) = 2/dx;
        elseif abs(i-j) == 1
            M(i,j) = dx/6;
            K(i,j) = -1/dx;
        else
            M(i,j) = 0;
            K(i,j) = 0;
        end
    end
    f(i) = 0; %kept in case wanted to be adjusted, but zero for this ex
end
```

```
K(1,end) = -1/dx;
K(end,1) = -1/dx;
M(1,end) = dx/6;
M(end,1) = dx/6;

% The initial condition and steady state
% note that because we have fixed BCs, we'll solve n-2 points
% if we had Neumann BCs, we'd need to solve n points
% i.e., be careful here!

u_old = 3*sin(4*pi*x); % IC of zeros at the boundaries, one elsewhere
u_new = zeros(1,(n+2));      % Empty matrix to hold the n+1 solution in time
usted = 0*xs; % plotting the steady state solution

% Completing the time integration

% Plotting the initial condition
figure
plot(xs,usted,'--b','LineWidth',lw)                    % steady state solution
hold on
grid on
plot([x,R],[u_old,(3*sin(4*pi*R))],'-r','LineWidth',lw)              % initial condition
xlabel('$x$','FontSize',FS)
ylabel('$u(x,t)$','FontSize',FS)
title("Periodic at $t=$"+0,'FontSize',FS)
ylim([(view_min+view_min/10),(view_max+view_max/10)])  % keeping the frame 'steady'
xlim([L-.1,R+.1])
legend('Steady State','Transient Sol.','location','northeast')
hold off
pause(.1)
Tstr = num2str(0);
filename = strcat("C:\Pictures\head1d_period\Heat_to_steady_t_",Tstr,".png");
%exportgraphics(gcf,filename)
count = 1;
timecount = ceil(length(t)/8);

% The actual integration
for T = t(2:end) % this "T" acts like the 'nth' timestep in the paper
        u_new = M\(dt.*(f-k.*K*u_old')) + u_old'; % finding u_{n+1}
        count = count+1;
        if mod(count,timecount) == 0 || T == t(end)
            plot(xs,usted,'--b','LineWidth',lw)  % plotting the steady state
            hold on
            grid on
            plot([x,R],[u_new',u_new(1)],'-r','LineWidth',lw)
            xlabel('$x$','FontSize',FS)
            ylabel('$u(x,t)$','FontSize',FS)
            title("Periodic at $t=$"+T,'FontSize',FS)
            ylim([(view_min+view_min/10),(view_max+view_max/10)])  % keeping the frame '
                steady'
            xlim([L-.1,R+.1])
            legend('Steady State','Transient Sol.','location','northeast')
            hold off
            pause(0.1)
            %Tstr = num2str(T);
            %filename = strcat("C:\Pictures\head1d_period\Heat_to_steady_t_",Tstr,".png");
            %exportgraphics(gcf,filename)
        end
        u_old = u_new'; % making the updated frame become the 'old' frame
end
```

3 Time-Dependent 2D Heat Equation using FEM

Let's now attempt to wrangle a numerical solution obtained via FEM for a problem simultaneously illustrating several of the complicating factors mentioned above, i.e.:

- higher-dimensionality [we'll consider a 2D heat equation];
- time integration [we'll consider a problem with temporal evolution, and not just a steady state problem];
- more interesting boundary conditions [we'll consider a rectangular region exhibiting periodic boundary conditions (identifying the top and bottom boundaries) with homogeneous Dirichlet boundary conditions (top and bottom boundaris)].

Such a problem is given by the following initial boundary value problem (IBVP):

$$u_t - k\nabla^2 u = f, \quad \text{subject to} \quad \Big\{ u(x,y,0) = u_0(x,y) \qquad \text{and} \qquad \begin{cases} u(x,0,t) = u(x,H,t) \\ u_y(x,0,t) = u_y(x,H,t) \\ \\ u(0,y,t) = u(L,y,t) = 0 \end{cases} \tag{31}$$

where the scalar field $u = u(x,y,t)$ is defined on the rectangular domain:

$$(x,y) \ \in \ \Omega = [0,L] \times [0,H], \tag{32}$$

for the time range $t \in [0,T]$. The region Ω is illustrated below.

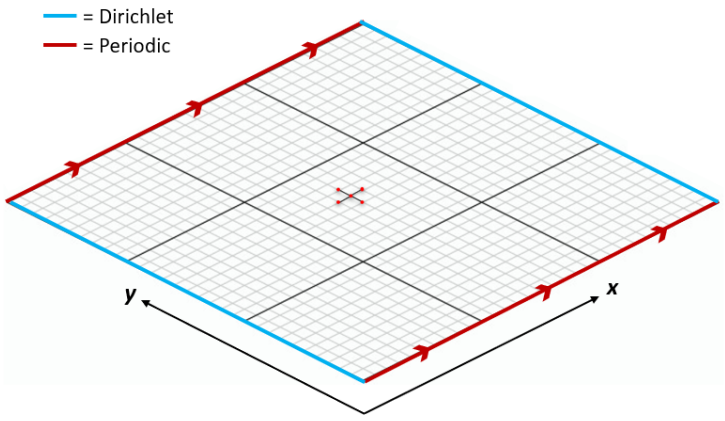


Figure 5: Illustrating the domain Ω and boundary conditions of the IBVP in (31).

To implement a FEM solution to the IBVP above, we rewrite (31) in its weak form by multiplying by a well-behaved test function $v = v(x,y)$ (satisfying $v(x,0) = v(x,H) = v(0,y) = v(L,y) = 0$) and integrating over the domain [1]:

$$\int_\Omega v u_t \, d\Omega - k \underbrace{\int_\Omega v \nabla^2 u \, d\Omega}_{\text{Integral}} = \int_\Omega f v \, d\Omega. \tag{33}$$

One can employ the usual sneakiness of Green's first identity (a higher-dimensional analog of integration by parts) on the integral above to obtain

$$\text{Integral} = \int_\Omega v \nabla^2 u \, d\Omega = \oint_{\partial\,\Omega} \big[(v \nabla u) \cdot \mathbf{n} \big] d(\partial\,\Omega) - \int_\Omega \big[\nabla u \cdot \nabla v \big] d\Omega,$$

[1] Here, we use the notation $\int_\Omega (\dots) d\Omega$ to indicate iterated integration over the domain, i.e., $\int_{y=0}^{y=H} \int_{x=0}^{x=L} (\dots) dx dy$.

where $\mathbf{n}$ denotes the unit vector normal to the boundary $\partial\,\Omega$, and where substitution back into (33) gives:

$$\int_\Omega vu_t \; d\Omega - k \underbrace{\oint_{\partial\,\Omega} \big[(v\nabla u)\cdot \mathbf{n}\big] d(\partial\,\Omega)}_{\text{contour integral}} + k\int_\Omega \big[\nabla u \cdot \nabla v\big] d\Omega = \int_\Omega fv \; d\Omega. \tag{34}$$

The contour integral above is determined by our boundary conditions, and simplifies to the tune of

$$\oint_{\partial\,\Omega} \big[(v\nabla u)\cdot \mathbf{n}\big] d(\partial\,\Omega) = \underbrace{\int_0^H \big[\big(v(0,y)\nabla u(0,y,t)\big)\cdot \mathbf{n}\big] dx}_{v(0,y)=0} + \underbrace{\int_0^H \big[\big(v(L,0)\nabla u(L,y,t)\big)\cdot \mathbf{n}\big] dy}_{v(L,0)=0}$$
$$+ \underbrace{2\int_0^L \big[\big(v(x,0)\nabla u(x,0,t)\big)\cdot \mathbf{n}\big] dx}_{v(0,y)=v(L,y)=0},$$

eventually vanishing:

$$\oint_{\partial\,\Omega} \big[(v\nabla u)\cdot \mathbf{n}\big] d(\partial\,\Omega) = 0.$$

The much simplified weak formulation in (34) then becomes:

$$\int_\Omega vu_t \; d\Omega + k\int_\Omega \big[\nabla u \cdot \nabla v\big] d\Omega = \int_\Omega fv \; d\Omega. \tag{35}$$

Falling prey to the ever-seductive powers of discretization, one can now provide piece-wise linear approximations to the solution u and test function v by expanding each, respectively, over a finite (N-dimensional) basis V of trial functions [2] $\phi = \phi(x,y)$, yielding:

$$u \approx \sum_{i=1}^{N} u_i(t)\phi_i(x,y) \quad \text{and} \quad v(x,y) \mapsto \phi_i(x,y).$$

Two examples of piece-wise linear trial functions $\phi_{80}(x,y)$ and $\phi_{140}(x,y)$ are illustrated below [3]. A snippet of MATLAB code that discretizes a rectangular domain into triangular regions and creates 2D elements such as these is also provided.

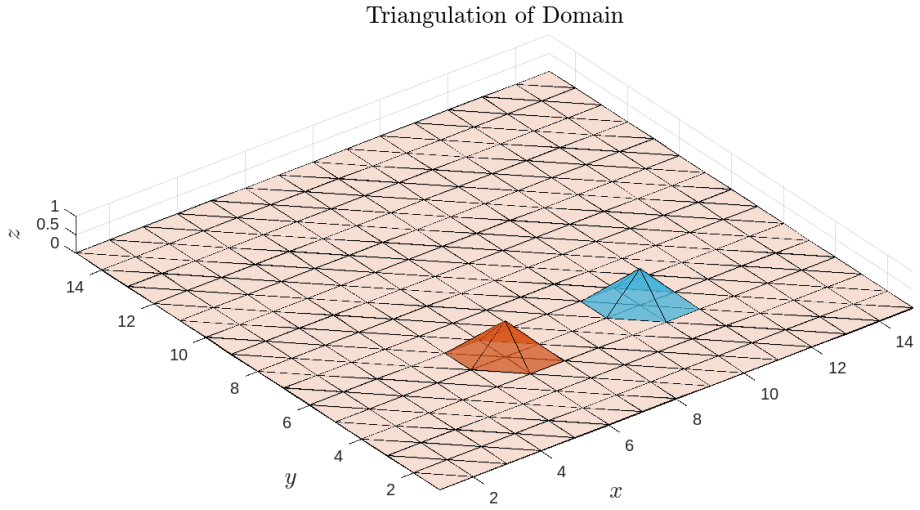


Figure 6: Example piece-wise linear trial functions $\phi_{80}(x,y)$, orange, and $\phi_{140}(x,y)$, blue, on a discretized rectangular domain.

[2] In this case, we choose the Delaunay-Voronoi triangulation τ of unit-height triangular pyramids given by $\phi_i(x,y) = \tau(1 - |x - x_i| - |y - y_i|)$ on a rectangular grid, which one can obtain with MATLAB's `delaunay` command. (That is, our basis/trial functions are the resulting volume obtained by triangulating such a pyramid, shown above.)

[3] The trial functions are numbered according to the number of their central node; here, node 1 starts at the origin and the node-numbering increments up the y-axis before wrapping back to the x-axis.

```
%% Code to discretize 2D rectangular domain into triangular regions
% And plot a few 2D linear elements

% 2D element (pre-triangulation)
myelement =@(x,y,a,b) max(1 - abs(x-a) - abs(y-b),0);

% Triangulate the domain
[x,y] = meshgrid(1:15,1:15);
T = delaunay(x,y);

% Two example elements
z1 = myelement(x,y,10,10);
z2 = myelement(x,y,5,5);

% Plot these examples
figure (1)
trisurf(T,x,y,z1,'facecolor',[0 0.4470 0.7410])
alpha 0.75; hold on;
trisurf(T,x,y,z2,'facecolor',[0.8500 0.3250 0.0980])
alpha 0.75; hold on;
trisurf(T,x,y,0*z2,'facecolor','w')
alpha 0.75;
axis equal;
colormap jet;

title('Triangulation of domain','Interpreter','latex','FontSize',fs)
xlabel('$x$','Interpreter','latex','FontSize',fs)
ylabel('$y$','Interpreter','latex','FontSize',fs)
zlabel('$z$','Interpreter','latex','FontSize',fs)
legend('Example shape function','Example shape function',...
    'Interpreter','latex','FontSize',fs-4)
```

After expanding u in this basis, substitution into the weak form of the PDE in (35) above gives

$$\int_\Omega \phi_j \left(\sum_{i=1}^{N} \dot{u}_i \phi_i \right) d\Omega + k \int_\Omega \left[\nabla \left(\sum_{i=1}^{N} u_i \phi_i \right) \cdot \nabla \phi_j \right] d\Omega = \int_\Omega f_i \phi_i d\Omega,$$

where $f_i \equiv f(x_i, y_i, t)$; equivalently, we have

$$\sum_{i=1}^{N} \dot{u}_i \underbrace{\left(\int_\Omega \phi_i \phi_j d\Omega \right)}_{\mathbf{M}_{ij}} + \sum_{i=1}^{N} u_i \underbrace{\left(k \int_\Omega \nabla \phi_i \cdot \nabla \phi_j d\Omega \right)}_{\mathbf{K}_{ij}} = \underbrace{\left(\int_\Omega f_i \phi_i \, d\Omega \right)}_{\mathbf{f}_i}. \tag{36}$$

In vector form $\mathbf{u} = (u_1, \ldots, u_N)^T$ and $\mathbf{f} = (f_1, \ldots, f_N)^T$ where $N \equiv N_x N_y$, the expression above can be recast to the tune of:

$$\mathbf{M}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}(t), \tag{37}$$

where the (very bold) matrices $\mathbf{M}$ and $\mathbf{K}$ are defined by the integrals over the trial functions above. Discretizing the solution in time via the forward Euler approximation of $\dot{\mathbf{u}}$, one arrives a linear system to iterate through time:

$$\mathbf{M}\left(\frac{\mathbf{u}^{n+1} - \mathbf{u}^n}{\Delta t} \right) + \mathbf{K}\mathbf{u} = \mathbf{f}^n,$$

where $\mathbf{u}^n \equiv \mathbf{u}(t_n)$ and $\mathbf{f}^n \equiv \mathbf{f}(t_n)$, or equivalently,

$$\mathbf{u}^{n+1} = \mathbf{u}^n + \mathbf{M}^{-1}\Delta t\big(\mathbf{f}^n - \mathbf{K}\mathbf{u}\big). \tag{38}$$

This form is preferred mostly because it has more *linear pizzazz*.

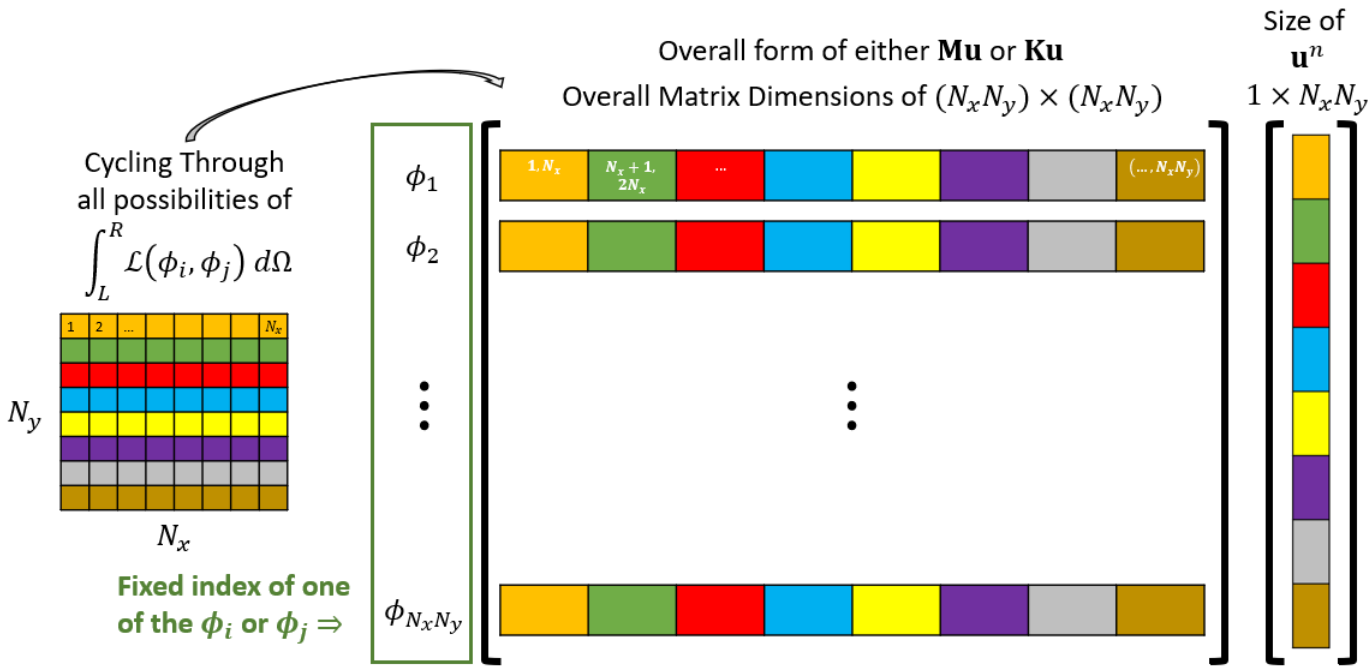


Figure 7: If we let $\mathcal{L}$ be either $\phi_i\phi_j$ or $\nabla\phi_i\nabla\phi_j$, then the formulation of the **Ku** and **Mu** matrix operations are similar in how they cycle through the different nodes of the discrete domain. Effectively the square domain is strung out into extremely long column and row vectors and then put back together after being calculated.

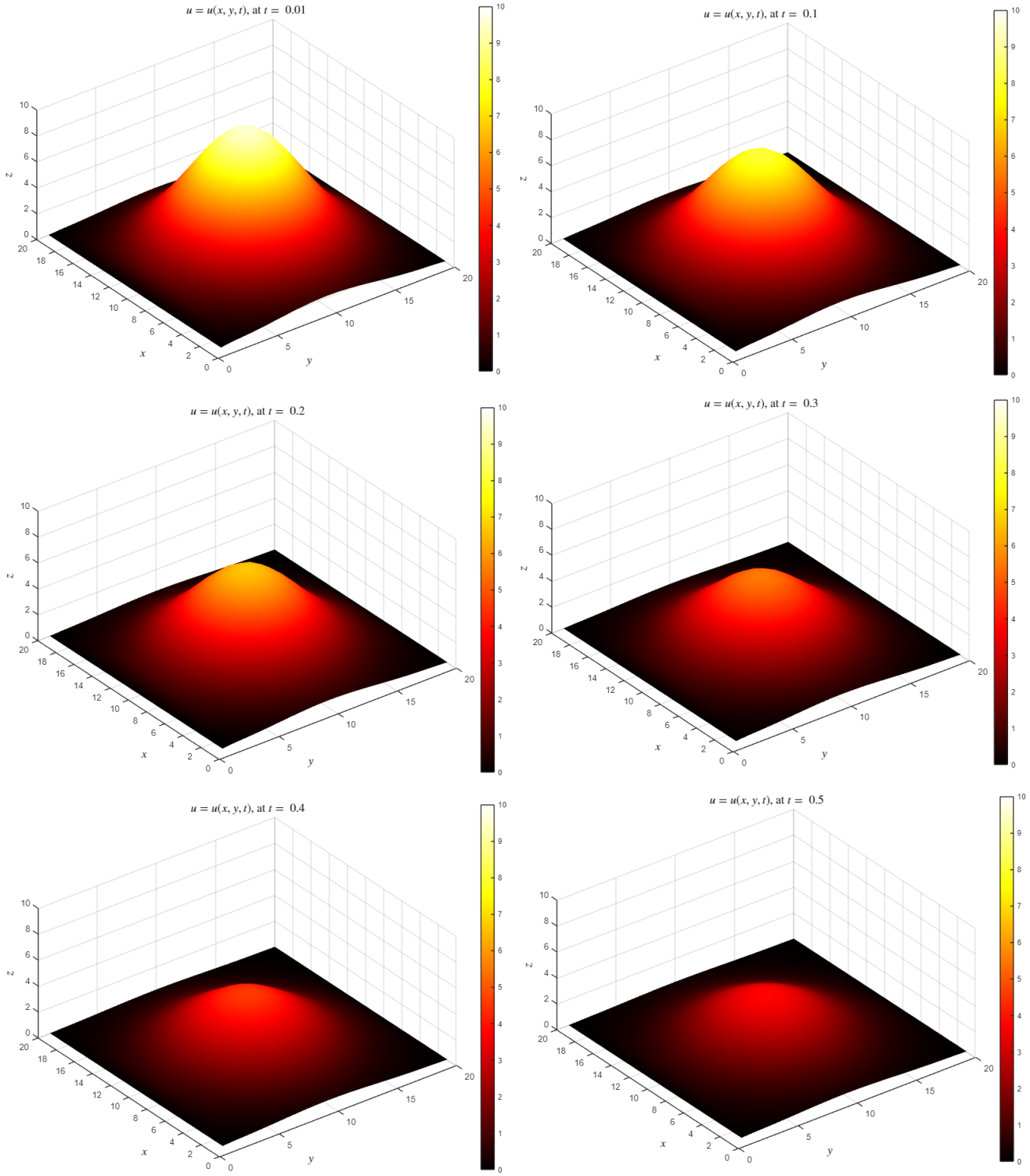
A note of caution here: constructing the matrices **M** and **K** can be tricky, partly due to the fact that each must be consistent with the particular scheme chosen to 'wrap' the 2D discretized grid above into the single skinny column vector **u**, which is certainly not unique. In the case that an entire y-column is placed into **u** at a time (corresponding with the node-numbering above), each matrix takes the form of a $N_xN_y \times N_xN_y$ square, banded block matrix (similar to the matrix formed from discretized a Laplacian operator in 2D). Some MATLAB code constructing both matrices according to such a scheme (for use with periodic boundary conditions) is provided here, for convenience. For converting between matrices and vectors, the `reshape` command can be very helpful (see Figure 7).

```
% Define a helper function for creating the K matrix
function Bingus = PlaceKDelta(Nx,Ny,diagonal)
    Delta = diagonal*diag(ones(Ny,1));

    % Create the upper portion
    DeltaBlockU = {0};
    for m = 1:Ny
        DeltaBlockU{m} = Delta;
    end
    DeltaBlockU = blkdiag(DeltaBlockU{1:end});
    DeltaBlockU = circshift(DeltaBlockU,Nx,2);

    % Create the lower portion
    DeltaBlockD = {0};
    for m = 1:Ny
        DeltaBlockD{m} = Delta;
    end
    DeltaBlockD = blkdiag(DeltaBlockD{1:end});
    DeltaBlockD = circshift(DeltaBlockD,Nx,2)';

    % Return the combined blocks
    Bingus = DeltaBlockU + DeltaBlockD;
end

% Define a function that returns the K matrix
```

```
function K = MakeK(Nx,Ny,diagonal,squr)

    % Add the D blocks that run down the diagonal
    Block = {0};
    for m = 1:Ny
        Block{m} = squr*diag(ones(Nx,1)) + diagonal*diag(ones(Nx-1,1),1) + diagonal*diag(
            ones(Nx-1,1),-1);
    end
    K = blkdiag(Block{1:end});

    % Add the Iu and Id blocks on the off-diagonals
    Bingus = PlaceKDelta(Nx,Ny,diagonal);

    % Deliver the final spicy meatball
    K = K + Bingus;
end


% Define a helper function for creating the M matrix
function Bingus = PlaceDelta(Nx,Ny,diagonal)
    Delta = diagonal*diag(ones(Ny,1)) + diagonal*diag(ones(Ny-1,1),1) + diagonal*diag(ones
        (Ny-1,1),-1);

    % Create the upper portion
    DeltaBlockU = {0};
    for m = 1:Ny
        DeltaBlockU{m} = Delta;
    end
    DeltaBlockU = blkdiag(DeltaBlockU{1:end});
    DeltaBlockU = circshift(DeltaBlockU,Nx,2);

    % Create the lower portion
    DeltaBlockD = {0};
    for m = 1:Ny
        DeltaBlockD{m} = Delta;
    end
    DeltaBlockD = blkdiag(DeltaBlockD{1:end});
    DeltaBlockD = circshift(DeltaBlockD,Nx,2)';

    % Return the combined blocks
    Bingus = DeltaBlockU + DeltaBlockD;
end

% Define a function that returns the M matrix
function M = MakeM(Nx,Ny,diagonal,squr)

    % Add the D blocks that run down the diagonal
    Block = {0};
    for m = 1:Ny
        Block{m} = squr*diag(ones(Nx,1)) + diagonal*diag(ones(Nx-1,1),1) + diagonal*diag(
            ones(Nx-1,1),-1);
    end
    M = blkdiag(Block{1:end});

    % Add the Iu and Id blocks on the off-diagonals
    Bingus = PlaceDelta(Nx,Ny,diagonal);

    % Deliver the final spicy meatball
    M = M + Bingus;
end
```

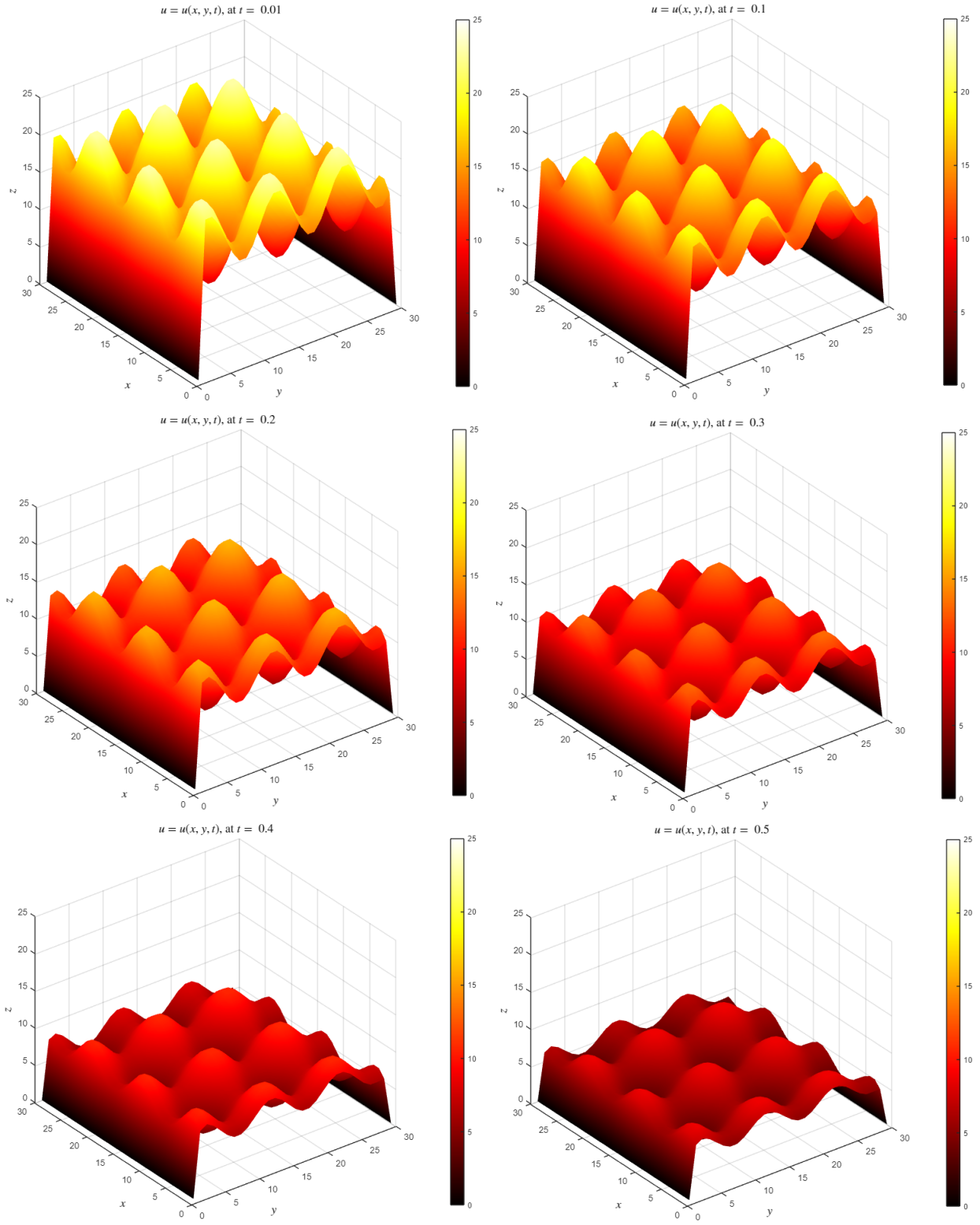
For the slightly anti-climatic piéce de résitance of this section, we now plot a numerical solution obtained using the above FEM techniques on a 20×20 discrete grid, using the initial condition $u(x, y, 0) = u_0(x, y)$ and $f = 0$, where

$$u_0(x, y) = 10 \exp\Big[-0.03\Big((x-10)^2 + (y-10)^2\Big)\Big].$$



The effect of the periodic boundary conditions are more clearly seen (and accurately used) with a periodic initial condition, such as the following nuclear pasta on a 30×30 grid (here, the forcing term $f = 0$):

$$u_0(x, y) = 4\left[\sin\left(\frac{3\pi x}{15}\right) + \sin\left(\frac{3\pi y}{15}\right) + 4\right].$$



3.1 MATLAB Code

```
%% 2D heat equation using FEM, with forcing
% (Cook the 2D turkey)
clear, clc, clf;

% Font size, for plotting
fs = 14;

%% Set up initial condition u0 = u(x,y,0)
% Specify the values of u0 on the grid (xi,yj) = (i,j)

% Initial Gaussian
Amplitude = 4;
Damping = 0.03;
G =@(x,y,xo,yo) Amplitude*exp(-Damping*(x-xo).^2-Damping*(y-yo).^2);
%G =@(x,y,xo,yo) Amplitude*(sin(3*pi*x/15) + sin(3*pi*y/15) + 4);

% Center the Gaussian on the grid
xo = Nx/2;
yo = Ny/2;

% Fill a matrix with the discretized initial conditions
u0 = zeros(Ny,Nx);
for j = 1:Ny
    for i = 1:Nx
        u0(j,i) = G(i,j,xo,yo);
    end
end

% Plot the continuous version of the IC alongside u0
figure (2)

% Continuous
subplot(2,1,1)
[X,Y] = meshgrid(linspace(1,Nx,5*Nx), linspace(1,Ny,5*Ny));
Z = G(X,Y,xo,yo);
surf(X,Y,Z)
colorbar
shading interp;
colormap hot;
axis equal
title('$u_0 = u(x,y,0)$ (Continuous)','Interpreter','latex','FontSize',fs)
xlabel('$x$','Interpreter','latex','FontSize',fs)
ylabel('$y$','Interpreter','latex','FontSize',fs)
zlabel('$z$','Interpreter','latex','FontSize',fs)

% Piece-wise linear approximation
subplot(2,1,2)
s = mesh(u0);
s.FaceColor = 'interp';
colorbar
shading interp;
colormap hot;
axis equal
title('$u_0 = u(x,y,0)$ (Piecewise linear)','Interpreter','latex','FontSize',fs)
xlabel('$x$','Interpreter','latex','FontSize',fs)
ylabel('$y$','Interpreter','latex','FontSize',fs)
zlabel('$z$','Interpreter','latex','FontSize',fs)

```

```
%% Time stepping algorithm
% FORWARD EULER: Stability conidition, dt <= 1/(2*((1/dx)^2+(1/dy)^2))
t0 = 0;  % initial time
tf = 0.5; % final time
dt = 0.02; % time discretization
Nt = ceil((tf-t0)/dt);

% Fill up the M matrix
% (Elements M_ij: phi(i)*phi(j) is zero unless adjacent/identical)

% Integral for tent functions overlapping diagonally
Mdiagonal = 1/12;
Kdiagonal = 0;

% Integral for left/right or up/down overlapping tent functions
Madjacent = 1/12;
Kadjacent = -20*Damping; % -k

% Integral for whole element squr
Msqur = 17/3 + 1/6;
Ksqur = 24*20*Damping; % 24k

% Create the M matrix
M = MakeM(Nx,Ny,Mdiagonal,Msqur);

% Create the K matrix
K = MakeK(Nx,Ny,Kadjacent,Ksqur);

% Define the forcing function
f = 0*ones(Nx*Ny,1);

% Wrap u0 into a vector and use it to initialize u
u0 = reshape(u0',[Nx*Ny,1]);
uold = u0;

figure (3)
for t = t0:dt:tf
    unew = uold + M\(dt*(f - K*uold));

    % Enforce periodic BC's
    unew(1:Ny) = 0;
    unew(end-Ny+1:end) = 0;
    uold = unew;

    es = mesh(reshape(unew,[Nx,Ny]));
    es.FaceColor = 'interp';
    colorbar
    shading interp;
    colormap hot;
    clim([0 25]);
    axis equal
    title("$u = u(x,y,t)$, at $t =$ "+t,'Interpreter','latex','FontSize',fs)
    xlabel('$y$','Interpreter','latex','FontSize',fs)
    ylabel('$x$','Interpreter','latex','FontSize',fs)
    zlabel('$z$','Interpreter','latex','FontSize',fs)
    xlim([0 Nx])
    ylim([0 Ny])
    zlim([0 25])
    pause(dt)
end
```

4 FreeFem++ Simulations of the NLS

We now switch the focus of our FEM analysis to the defocusing nonlinear Schrödinger equation (NLS) in a new geometric setting: the parameterized surface $\mathbf{s} = \mathbf{s}(\phi, \theta)$ of an 'elongated' torus, with an elliptical waist curve (pictured below).

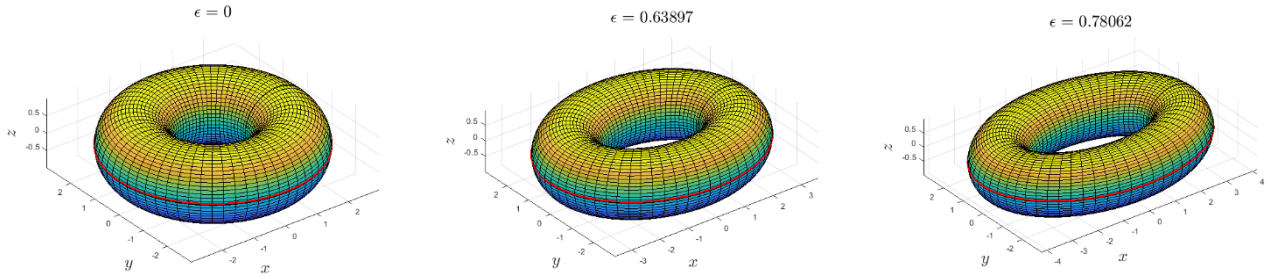


Figure 8: Elongated tori of different eccentricities; the surface coordinates ϕ and θ run horizontally and vertically along the surface, respectively. From left to right, these eccentricities are: $\epsilon = 0$, $\epsilon \approx 0.63897$ and $\epsilon \approx 0.78062$. Note that when the $\epsilon = 0$, the waist curve is a circle and the elongated torus coincides exactly with the usual torus $\mathbb{T}^2$, exhibiting symmetry in the toroidal direction. For eccentricities $\epsilon > 0$, the toroidal symmetry is broken.

Here, the surface coordinates ϕ and θ represent the toroidal (waist) and poloidal (vertical, 'wrapping') directions, respectively, and the corresponding complex-valued wave function $\psi = \psi(\phi, \theta, t)$ satisfies the strong form of the NLS:

$$i\psi_t + \frac{1}{2}\Delta_{\mathbf{s}}\psi - |\psi|^2\psi = 0. \tag{39}$$

Under the curvature[4] induced by the elongated torus, the corresponding Laplace-Beltrami operator $\Delta_{\mathbf{s}}$ takes the form:

$$\Delta_{\mathbf{s}} = \frac{1}{r\gamma}\left[\left(\frac{\gamma}{r}\right)\partial_{\theta\theta} + k_1\,\partial_\theta + k_2\,\partial_\phi + \left(\frac{r}{\gamma}\right)\partial_{\phi\phi}\right], \tag{40}$$

where the coefficients of the single derivative operators ∂_θ and ∂_ϕ are

$$\begin{cases} k_1 = -\dfrac{\sin(\theta)\Big[\cos^2(\phi)\big(b + r\cos(\theta)\big) + \sin^2(\phi)\big(a + r\cos(\theta)\big)\Big]}{\gamma} \\[2ex] k_2 = -\dfrac{2r\cos(\phi)\sin(\phi)\Big[(a + r\cos(\theta))^2 - (b + r\cos(\theta))^2\Big]}{2\gamma^3} \end{cases}.$$

The quantity $\gamma = \gamma(\phi, \theta)$, defined for notational convenience, is given by

$$\gamma \equiv \sqrt{(a + r\cos(\theta))^2\sin^2(\phi) + (b + r\cos(\theta))^2\cos^2(\phi)}\ .$$

The corresponding gradient operator $\nabla_{\mathbf{s}}$ on the surface of the elongated torus is given by:

$$\nabla_{\mathbf{s}} = (\gamma\,\partial_\phi)\hat{\phi} + (r\,\partial_\theta)\hat{\theta}. \tag{41}$$

4.1 Weak Form of the NLS

Under periodic boundary conditions, the weak form of the NLS (39) is given by

$$i\int_\Omega \big[\psi_t v\big]d\Omega + \frac{1}{2}\int_\Omega \big[\Delta_{\mathbf{s}}\psi v\big]d\Omega - \int_\Omega \Big[|\psi|^2\psi v\Big]d\Omega = 0,$$

[4]The effects of curvature on this surface can be described by the metric tensor $\mathbf{g}$ it induces. Although this lies a bit outside of the scope of this paper, the metric tensor $\mathbf{g}$ was used to construct the gradient $\nabla_{\mathbf{s}}$ and Laplace-Beltrami $\Delta_{\mathbf{s}}$ operators used above, using the curvilinear surface coordinates (ϕ, θ). This process produces the coefficients k_1, k_2 and γ, but won't be shown here.

or equivalently (after an application of Green's first identity), by:

$$i\int_\Omega \left[\psi_t v\right] d\Omega - \frac{1}{2}\int_\Omega \left[\nabla_{\mathbf{s}}\psi \cdot \nabla_{\mathbf{s}} v\right] d\Omega - \int_\Omega \left[(\psi\psi^\star)\psi v\right] d\Omega = 0, \tag{42}$$

where $\psi^\star$ denotes the complex conjugate of ψ.

4.2 Setting Up the Problem in `FreeFem++`

Here, we choose to use the popular open-source FEM software **`FreeFem++`** (see https://freefem.org/) to aid our FEM analysis. The **`FreeFem++`** code developed is given below in section (4.6). The nature of the problem described in (39) presents a few practical complications/hurdles that require some care when implementing a FEM solution; namely, the problem is:

- nonlinear
- defined on a periodic domain;
- complex-valued.

Several links that are particularly helpful in learning **`FreeFem++`** are given in the references section below (including one example of a previous implementation of the NLS using the software).

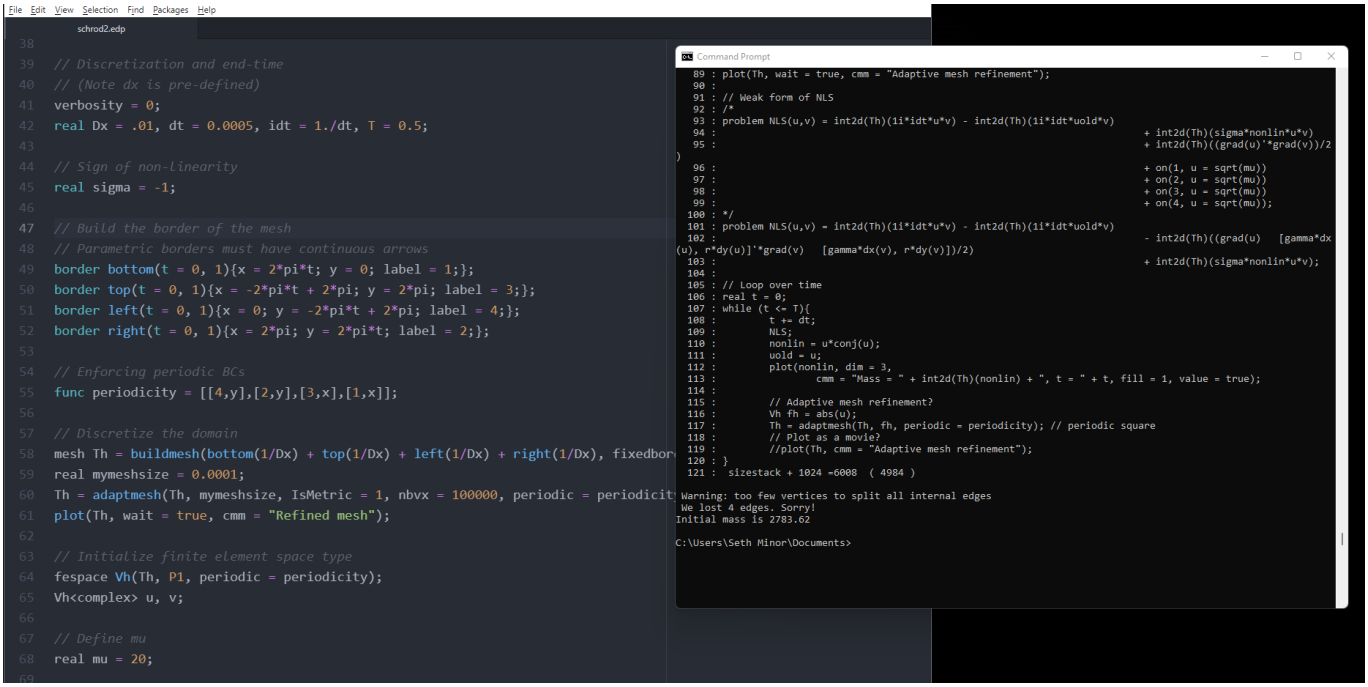


Figure 9: An example of using **`FreeFem++`** on a Windows OS.

To obtain a numerical solution to a time-dependent PDE in **`FreeFem++`**, one needs to provide the weak form of the PDE in question and implement a desired time-integration scheme. For our purposes, we use backwards-Euler in time, reducing the weak form of the NLS in (42) to

$$i\int_\Omega \left[\left(\frac{\psi^n - \psi^{n-1}}{\Delta t}\right) v\right] d\Omega - \frac{1}{2}\int_\Omega \left[\nabla_{\mathbf{s}}\psi^n \cdot \nabla_{\mathbf{s}} v\right] d\Omega - \int_\Omega \left[(\psi\psi^\star)^{n-1}\psi^n v\right] d\Omega = 0, \tag{43}$$

where we define $\psi^n \equiv \psi(\phi, \theta, t_n)$ and we compute the nonlinearity $(\psi\psi^\star)^{n-1}$ at once, using the previous time-step[5] $n-1$.

[5]This is due to practical implementation details in **`FreeFem++`**, in which the weak form definition (see the `int2D` command) of nonlinear problems seems to require some elbow grease.

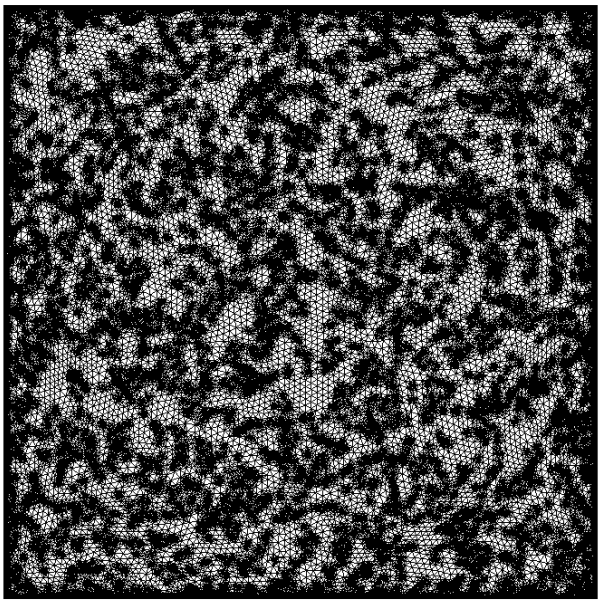


Figure 10: A fine mesh computed without adaptive mesh refinement.

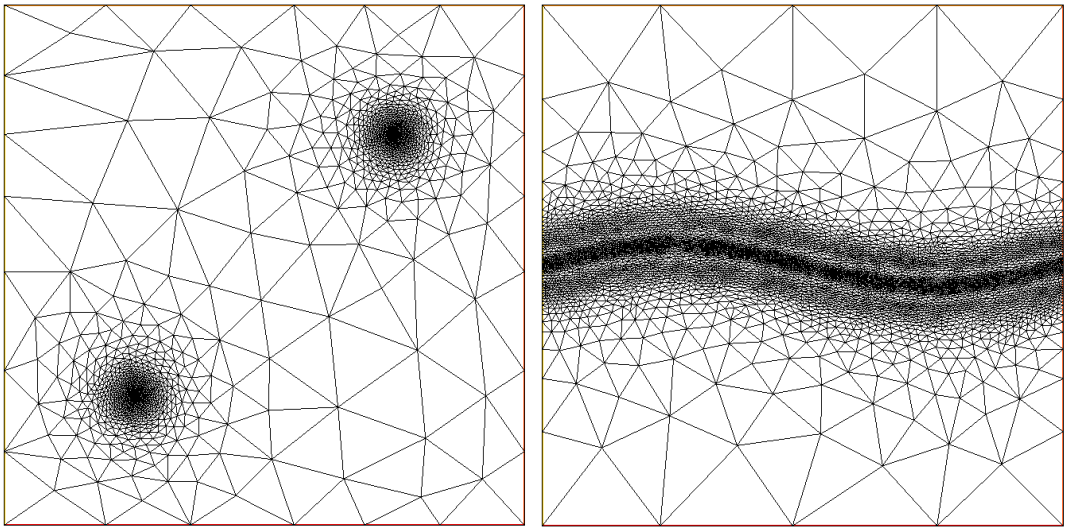


Figure 11: Adaptive mesh refinement for a vortex dipole (left) and a snaking dark soliton stripe (right).

4.3 A Note on Adaptive Mesh Refinement

An attractive feature of finite element analysis is its capacity for dealing with complicated geometries, which are discretized into correspondingly complicated (often triangular) grids. Since these grids or meshes need not be homogeneous (i.e., the area or generalized volume of each element may differ across a given spatial domain), one may adapt the density of a mesh to fit the contours and discontinuities, etc., of a particular problem. The upshot of such adaptive mesh refinement is increased resolution of a numerical solution in spatial regions where it's most needed, but without the the computational burden associated with reducing the size of a homogeneous mesh. The ability to handle adaptive mesh refinement represents one strong advantage FEM have over traditional FDS.

In the case of superfluid vortices on a torus, we expect that the background of the wave-function will be relatively stable, while more intensive dynamics will be occurring locally. Hence, not wasting computational effort on more or less unchanging regions reduces computational overhead, yielding faster and more accurate results. Adaptive mesh refinement is then useful for dark structures supported by the NLS (see Figures 10, 11, and 12). In `FreeFem++`, the `adaptmesh` command presents a powerful tool for adaptive mesh refinement. In our implementation, the `adaptmesh` command is used iteratively (on each time-step) to adapt the working mesh to the contours of the current numerical solution at that timestep.

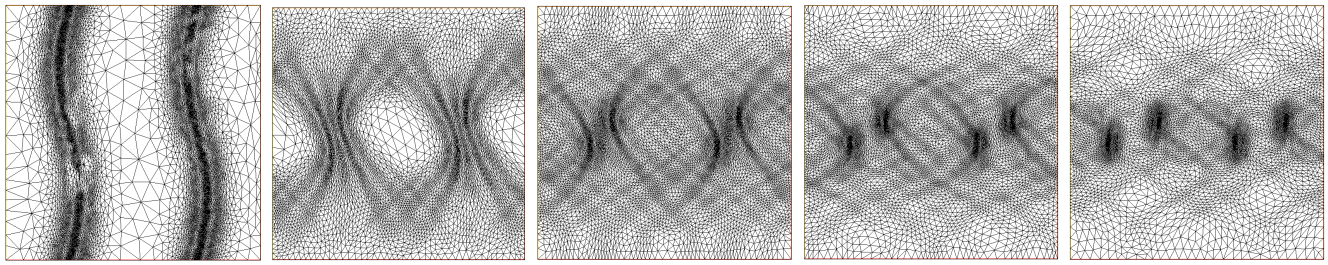


Figure 12: Example of iterative adaptive mesh refinement. In this example, the mesh tracks the dissolution of two snaking poloidal dark soliton stripes into what appear to be four vortices. Each mesh is separated in time by $\Delta t = 0.02$.

4.4 Numerical Results

As a benchmark or gauge of the numerical accuracy of our `FreeFem++` solver, we tested a toroidally-'snaking' dark soliton initial condition $\psi(\phi, \theta, 0) = \psi_0(\phi, \theta)$ (for the chemical potential parameter value $\mu = 20$), given by:

$$\psi_0(\phi, \theta) = 2 + \sqrt{\mu} \tanh\left[\sqrt{\mu}(\sin(x)/4 - y + \pi)\right]^2.$$

The simulation was conducted on the true torus (i.e., for $a = b = 3$ and $r = 2$, with periodic boundary conditions), and adaptive mesh refinement was used on a $[0, 2\pi] \times [0, 2\pi]$ domain with an initial mesh-size of `0.0001`. Time integration was performed using a backwards-Euler scheme with a time-step of `dt = 0.0005`, and the figures below correspond to time $t = 0, 0.1, 0.2$ from left to right, respectively.

In the simulation, the curving dark soliton (initially wrapped along the toroidal axis centered about $\phi = \pi$), appeared to 'dissolve' into two thicker, shallow dark stripes before bending sharply and transforming into what appear to be two vortices of opposing charge (see figures below).

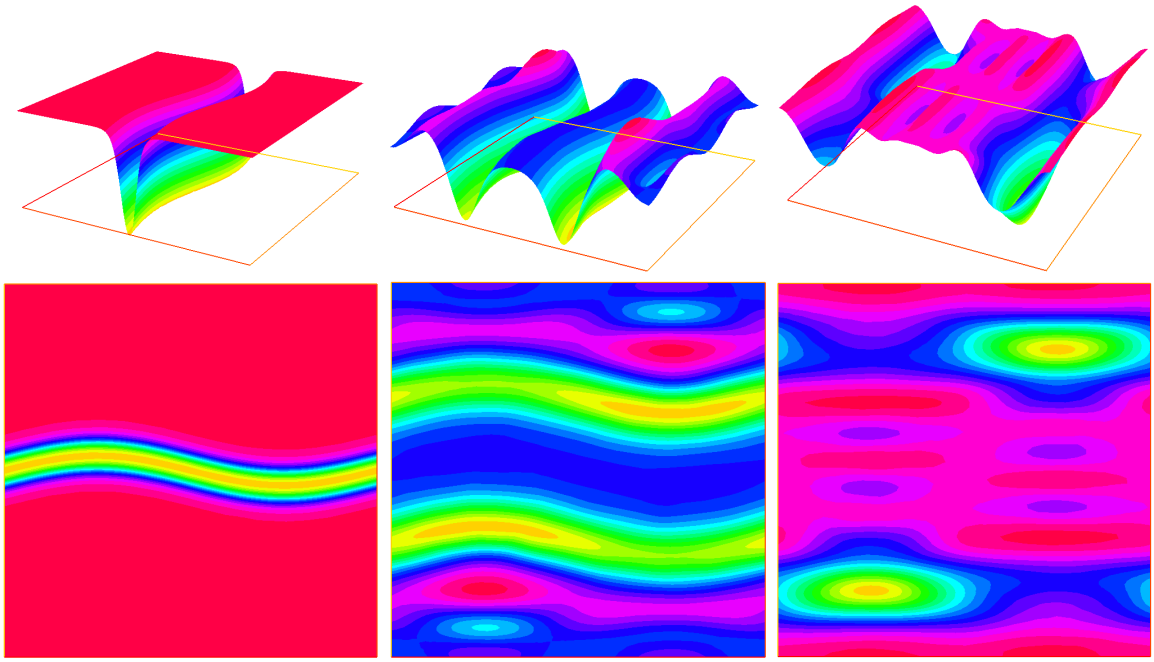


Figure 13: Numerical solution for a dark soliton initial condition snaking around the toroidal axis (centered at $\phi = \pi$), with the chemical potential $\mu = 20$. Images correspond to the times $t = 0, 0.1, 0.2$ from left to right, respectively. In the lower panel, the horizontal axis represent the toroidal axis and the vertical axis represents the poloidal, each running from 0 to 2π.

Another interesting intial configuration is that of two (equally spaced) poloidally-snaking dark soliton stripes, centered about $\theta = \pi/2$ and $\theta = 3\pi/2$, respectively; a numerical solution is obtained using the initial condition

$$\psi_0(\phi, \theta) = \sqrt{\mu}\tanh\left[\sqrt{\mu}(\sin(y)/4 - x + \pi/2)\right]^2 + \sqrt{\mu}\tanh\left[\sqrt{\mu}(\sin(y)/4 - x + 3\pi/2)\right]^2,$$

where we use the same μ and time-discretization parameter values as before (however, this time the figures below are separated in time by $\Delta t = 0.03$). Again, the dark soliton stripes are observed to 'dissolve' into what appear to be vortices of alternating charge.

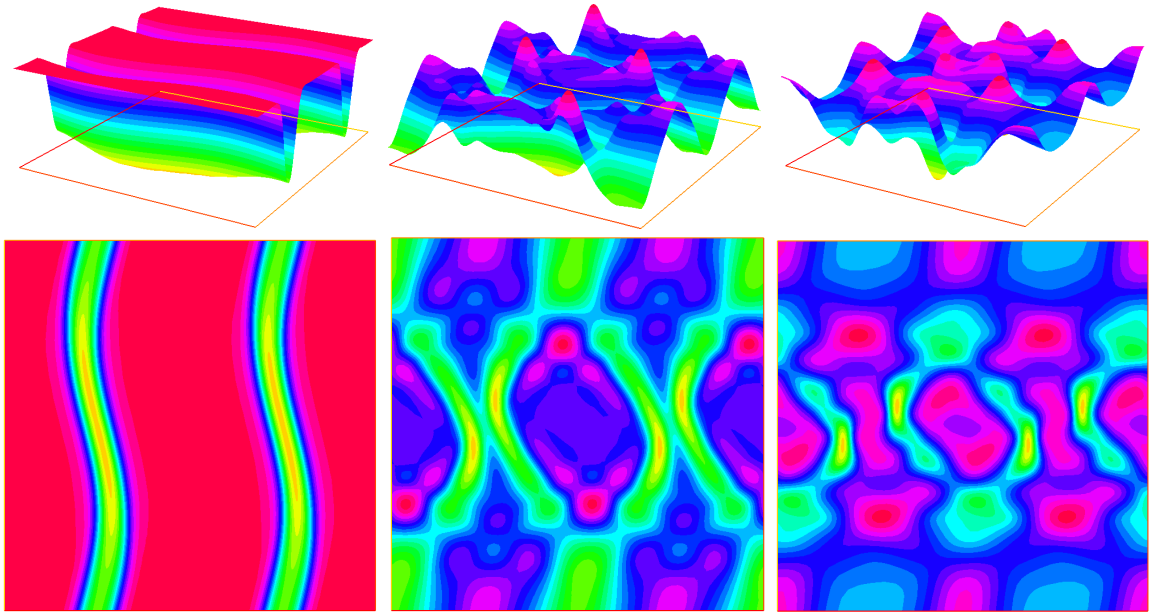


Figure 14: Numerical solution for two dark solitons initially snaking around the poloidal axis (centered at $\theta = \pi/2$ and $\theta = 3\pi/2$, respectively), with the chemical potential $\mu = 20$. Images correspond to the times $t = 0, 0.3, 0.6$ from left to right, respectively. In the lower panel, the horizontal axis represent the toroidal axis and the vertical axis represents the poloidal, each running from 0 to 2π.

Interestingly, the effects of elongating the semi-major axis a of the torus (i.e., giving the torus an elliptical, instead of circular, waist curve) can be explored by plotting the adapted meshes of the solutions over time, for various values of a (see Figures 8 and 15). In these plots, the adaptively-refine mesh acts as a sort of gradient filter, highlighting regions of sharply-varying density.

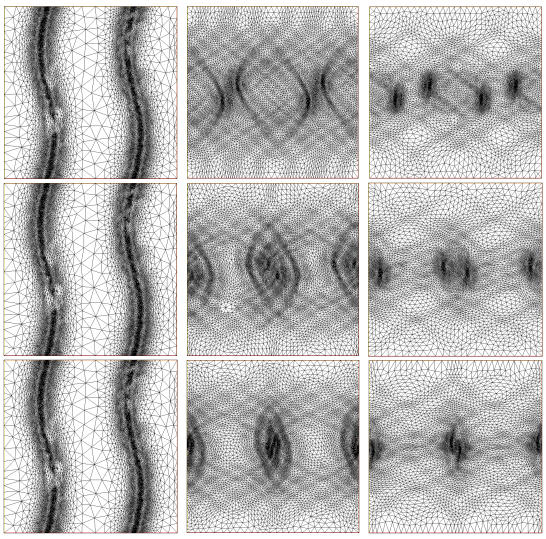


Figure 15: (From left to right) Adaptively-refined meshes for two poloidally-snaking dark soliton stripes at times $t = 0, 0.04, 0.08$; (top to bottom) semi-major axis lengths of $a = 3, 4, 5$, respectively. In each figure, the length of the semi-minor axis b is held at $b = 3$. In each image, the toroidal axis (ϕ) runs from 0 to 2π along the horizontal axis, while the poloidal (θ) runs from 0 to 2π along the vertical.

In the case of the two poloidally-snaking dark soliton stripes examined above, the elongation of the torus has the effect of creating two sharp bends along the waist-curve (see the 'pinched' part of the torus in figure 8 above); consequently, this appears to separate the dark structures located at $\phi = \pi/2$ and $\phi = 3\pi/2$ above (as the pinched corner continues to bend), as well as 'smush' together the dark structures located on the shallow sides of the waist curve ($\phi = 0$ and $\phi = 2\pi$). If the dark structures are in fact well-described by vortices of alternating charge, this elongation mechanism could result in a sort of forced local interaction between the vortices located at the shallow ends, and perhaps result in some interesting dynamics as the system evolves in time.

4.5 Concluding Thoughts, Future Improvements

An item to note regarding the accuracy of the simulations above: we struggled with an observed decline of the mass of the wave-function over time, given by the quantity:

$$M = \int_{\Omega} |\psi|^2 d\Omega,$$

which should (in principle) remain a conserved quantity in a numerical solution to the NLS. A diagnosis of this issue is a bit tough, as there are several plausible possible sources of error; however, the most intuitive candidate is the error introduced by time integrator (recall that a first-order scheme, backward Euler, was used)[6]. In future work a more accurate time integration scheme is recommended.

In general, finite element methods seem to be powerful options for the numerical solution of the NLS, and perhaps for nonlinear wave equations more broadly. Such methods bring a new tools to the table that are normally lost in traditional FDS. In particular, adaptive mesh refinement finds a natural home in the search for efficient simulations of dark-structures (or other locally sharply-varying structures in general). Moreover, the availability of open-source FEM software like `FreeFem++` make a transition from FDS to FEM much more palatable.

[6]The mass loss rate was observed to be roughly linear in time.

4.6 FreeFem++ Code

```
// FreeFem++ code for numerically solving the NLS

// Radius of torus
real r = 2;

// Semi-major axis of ellipse
real a = 3;

// Semi-minor axis of ellipse (a>=b)
real b = 3;

// Toroidal curvature quantity
// phi -> x
// theta -> y
func gamma = sqrt( (a + r*cos(y))^2*sin(x)^2 + (b + r*cos(y))^2*cos(x)^2 );

// Define macros
// Gradient on the surface of the torus
macro grad(u) [gamma*dx(u), r*dy(u)]//EOM


// Discretization and end-time
// (Note dx is pre-defined)
verbosity = 0;
real Dx = .01, dt = 0.0005, idt = 1./dt, T = 0.08;

// Sign of non-linearity
real sigma = -1;

// Build the border of the mesh
border bottom(t = 0, 1){x = 2*pi*t; y = 0; label = 1;};
border top(t = 0, 1){x = -2*pi*t + 2*pi; y = 2*pi; label = 3;};
border left(t = 0, 1){x = 0; y = -2*pi*t + 2*pi; label = 4;};
border right(t = 0, 1){x = 2*pi; y = 2*pi*t; label = 2;};

// Enforcing periodic BCs
func periodicity = [[4,y],[2,y],[3,x],[1,x]];

// Discretize the domain
mesh Th = buildmesh(bottom(1/Dx) + top(1/Dx) + left(1/Dx) + right(1/Dx), fixedborder =
    true);
real mymeshsize = 0.0001;
Th = adaptmesh(Th, mymeshsize, IsMetric = 1, nbvx = 100000, periodic = periodicity); //
    periodic square
plot(Th, wait = true, cmm = "Refined mesh");

// Initialize finite element space type
fespace Vh(Th, P1, periodic = periodicity);
Vh<complex> u, v;

// Define mu
real mu = 20;

// Define IC
//func u0 = exp(-(x - pi)^2 - (y - pi)^2); // Gaussian
//func u0 = 2 + sqrt(mu)*(tanh(sqrt(mu)*(x - pi)))^2 + 1i*sqrt(mu); // Dark soliton stripe
//func u0 = 2 + sqrt(mu)*tanh(sqrt(mu)*(sin(x)/4 - y + pi))^2; // A snaking dark soliton
    stripe
```

```
56 func u0 = sqrt(mu)*tanh(sqrt(mu)*(sin(y)/4 - x + pi/2))^2 + sqrt(mu)*tanh(sqrt(mu)*(sin(y)
       /4 - x + 3*pi/2))^2; // two poloidal stripes
57 //func u0 = sqrt(mu); // flat background
58 cout << "Initial mass is " << int2d(Th)(u0*conj(u0)) << endl;
59
60 // Initialize IC
61 Vh<complex> uold = u0, nonlin = u0*conj(u0);
62
63 // Adaptive mesh refinement based on IC
64 Vh fh = abs(u0), fhplot = fh^2;
65 plot(fhplot, wait = true, cmm = "Initial condition", fill = true, dim = 3, value = true);
66 Th = adaptmesh(Th, fh, periodic = periodicity); // periodic square
67 plot(Th, wait = true, cmm = "Adaptive mesh refinement");
68
69 // Weak form of NLS
70 problem NLS(u,v) = int2d(Th)(1i*idt*u*v) - int2d(Th)(1i*idt*uold*v) - int2d(Th)((grad(u)'*
       grad(v))/2) + int2d(Th)(sigma*nonlin*u*v);
71
72 // Loop over time
73 real t = 0;
74 while (t <= T){
75         t += dt;
76         NLS;
77         nonlin = u*conj(u);
78         uold = u;
79         plot(nonlin, dim = 2,
80                 cmm = "Mass = " + int2d(Th)(nonlin) + ", t = " + t,fill=1,value=true);
81
82         // Adaptive mesh refinement?
83         Vh fh = abs(u);
84         Th = adaptmesh(Th, fh, periodic = periodicity); // periodic square
85         // Plot as a movie?
86         //plot(Th, cmm = "Adaptive mesh refinement");
87 }
```

References:

1 *Superfluid vortex multipoles and soliton stripes on a torus*, [D'Ambroise, Carretero, Schmelcher, and Kevrekidis 2022]

2 *Superfluid vortex dynamics on a torus and other toroidal surfaces of revolution*, [Guenther, Massignan, and Fetter 2020]

3 *Unpublished, Textbook on Nonlinear Waves*, [Carretero et. al.]

4 *The Finite Element Method, Principles and Applications,* [Lewis, Ward]

5 *The Finite Element Method: Its Basis and Fundamentals, Sixth edition* [Zienkiewicz, Taylor, Zhu]

6 *Online German-English Translation Dictionary, "ansatz",* [https://www.leo.org/german-english]

Helpful `FreeFem++` links/tutorials:

- http://ionut.danaila.perso.math.cnrs.fr/zdownload/Tutorial_2019_Singapore/Singapore_Day_01.pdf
- http://ionut.danaila.perso.math.cnrs.fr/zdownload/Tutorial_2019_Singapore/Singapore_Day_02.pdf
- https://www.um.es/freefem/ff++/pmwiki.php?n=Main.Examples2D
- http://www.lamfa.u-picardie.fr/sadaka/FreeFem++/GTA3/GTA3FreeFem++2D.html
- http://ionut.danaila.perso.math.cnrs.fr/zdownload/Tutorial_2016_Fields/Tutorial_2016_Fields_Course_05_to_08.pdf