Raunak Singh

Civil & Environmental Engineering

UC Berkeley

Solving the Double Slit Experiment Using Finite Element Method

Introduction

We use FEM to numerically solve for u in the wave equation given as:

$$\ddot{u} - c^2 \nabla \cdot \nabla u = f \ in\, \Omega\ \times I$$

And are given the initial and boundary conditions:

$$u = u_0 \quad \vec{x}\ \in\ \Omega, t = 0$$

$$\dot{u} = v_0 \quad \vec{x}\ \in\ \Omega, t = 0$$

$$u = g \ \ on\, \Gamma_g \times I$$

$$c^2 \nabla \mathrm{u}\ \cdot\ \hat{n} = p\ on\ \ \Gamma_h \times I$$

$$c = 1$$

Where $\Gamma_g, \Gamma_h, \Omega, I$ are the Neumann boundary, Dirichlet boundary, physical domain, and temporal domain, respectively. The Dirichlet function g(t) is given as $0.5 * sin(6\pi t)$, and models the wavelength of light coming into the double slits. The Neumann function p is given as 0 along the Neumann boundary, which is the boundaries of our 2-D box (this represents the approximation that no light will travel through the box).

We solve this equation numerically using a semi-discrete Finite Element Model with linear basis functions and triangular elements. To approximate how these functions change over time, we average the forward and backward Euler approximations using the θ method, where $\theta \in [0,1]$ represents how much weight we give to the backward Euler approximation (and therefore $1 - \theta$ represents how much weight we give to the forward approximation).

We compare the stability, runtime, and accuracy of our numerical solutions across different values of θ. Because mass lumping can be used for the $\theta = 0$ case (the forward Euler approximation), we also compare mass-lumping and non-mass lumping solutions. Lastly, we conduct an extended simulation, where we set the Dirichlet conditions to 0 after t=3, so that we can see how energy dissipates with a 0 input to the system.

Methods

To simplify notation, we define:

$$\dot{u} = v$$

$$\dot{v} = \nabla \cdot \nabla u$$

$$\boldsymbol{U} = \begin{bmatrix} u \\ v \end{bmatrix}$$

$$\boldsymbol{f} = \begin{bmatrix} v \\ \nabla \cdot \nabla u \end{bmatrix}$$

It is clear from the equations above that we can set $\dot{\boldsymbol{U}} = \boldsymbol{f}$**.** Note that the second row of this equation is the same as the given equation $\ddot{u} - c^2 \nabla \cdot \nabla u = f \ in\ \Omega\ \times I$ after plugging in given values for c and f. We move $\boldsymbol{f}$ to the left side, so that the equation is equal to 0:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix} - \begin{bmatrix} v \\ \nabla \cdot \nabla u \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We ensure that the Neumann boundary condition holds by requiring:

$$(c^2 \nabla \mathrm{u} - \mathrm{p}) \cdot \hat{n} \ = 0$$

Because the above equation also equals 0, we can add this to the second row of $\dot{\boldsymbol{U}} - \boldsymbol{f}$. After plugging in the given values of p and c, we get:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix} - \left(\begin{bmatrix} v \\ \nabla \cdot \nabla u \end{bmatrix} \right) + \begin{bmatrix} 0 \\ \nabla \mathrm{u} \cdot \hat{n} \ d \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Using Galerkin's method, we approximate u and v as a linear combination of basis functions. We give a basis function to each node in our mesh. We use the superscript h to denote them as approximations, where h represents the number of nodes in our mesh.

$$u^h = \sum_{B \in \eta - \eta_g} N_B d_B + \sum_{B \in \eta_g} N_B g_B$$

$$w^h = \sum_{A \in \eta - \eta_g} N_A C_A$$

$$v^h = \sum_{B \in \eta - \eta_g} N_B e_B + \sum_{B \in \eta_g} N_B \dot{g}_B$$

Where η represents the set of all nodes, and η_g represents the set of all Dirichlet nodes. To account for an irregularly shaped domain, we let all basis functions equal 1 on their respective node, and 0 on all other nodes. N_x represents the basis function that equals 1 at node x, and 0 at all other nodes. Because w^h equals 0 at all Dirichlet nodes, we do not sum over Dirichlet nodes for the approximation w^h.

To find the optimal solution, we enforce that our solution is orthogonal to a residual which we define as $\dot{\boldsymbol{U}} - \boldsymbol{f} + \begin{bmatrix} 0 \\ \nabla \mathrm{u} \cdot \hat{n} \end{bmatrix}$. We enforce this by multiplying by our approximate solutions u^h and v^h; and since the right-hand side is 0, we can add arbitrary weights to the approximate solution we are multiplying by. Thus, this is the same as multiplying by an arbitrary weighting function w, which is represented by the same basis functions used for $\dot{\boldsymbol{U}}$. For simplicity, we use the same basis functions for $\boldsymbol{u_1}$ and for $\boldsymbol{u_2}$. We require, however, that w goes to 0 on the Dirichlet nodes because we already have our Dirichlet node values, and do not want to over constrain our equation. To ensure that this holds over the whole domain, we integrate both rows and sides over the whole domain:

$$\begin{bmatrix} \int\limits_{\Omega\times I} w^h(\dot{u}^h - v^h)\, d(\Omega \times I) \\ \int\limits_{I} \left(\int_{\Omega} (w^h \dot{v}^h - w^h \nabla \cdot \nabla u^h) d\Omega + \int_{\Gamma_h \times I} w^h \nabla \mathrm{u} \cdot \hat{n}\ d(\Gamma_h \times I) \right)\ dI \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We apply the divergence theorem and the product rule to the $w^h \nabla \cdot \nabla u^h$ term:

$$\begin{bmatrix} \int\limits_{\Omega\times I} w^h(\dot{u}^h - v^h)\, d(\Omega \times I) \\ \int\limits_{I} \left(\int\limits_{\Omega\times I} w^h \dot{v}^h + \nabla u^h \cdot \nabla w^h d(\Omega \times I) - \int_{\Gamma} w^h \nabla u^h \cdot \hat{n}\ d\Gamma + \int_{\Gamma_h} w^h \nabla \mathrm{u} \cdot \hat{n}\ d\Gamma_h \right)\ dI \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Because we require w to go to 0 on Dirichlet nodes, $\int_{\Gamma_g} w^h \nabla u^h \cdot \hat{n}\ d\Gamma = 0$. Thus, our surface integral becomes $\int_{\Gamma} w^h \nabla u^h \cdot \hat{n}\ d\Gamma = \int_{\Gamma_g} w^h \nabla u^h \cdot \hat{n}\ d\Gamma + \int_{\Gamma_h} w^h \nabla u^h \cdot \hat{n}\ d\Gamma = \int_{\Gamma_h} w^h \nabla u^h \cdot \hat{n}\ d\Gamma$.

This cancels with the term which specifies our Neumann boundary condition to give us:

$$\begin{bmatrix} \int\limits_{\Omega\times I} w^h(\dot{u}^h - v^h)\, d(\Omega \times I) \\ \int\limits_{I} \left(\int\limits_{\Omega\times I} w^h \dot{v}^h + \nabla u^h \cdot \nabla w^h d(\Omega \times I) \right)\ dI \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We plug in our approximations so that we can solve for the weights on the linear combination of basis functions. Because we have a semi discrete model, we let our basis functions stay constant over time, and let our basis function weights differ over time. Thus, when we differentiate each function with respect to time, we can take the basis functions as constants:

$$\begin{bmatrix} \int\limits_{\Omega\times I} \sum\limits_{A\in\eta-\eta_g} N_A c_A \left(\sum\limits_{B\in\eta-\eta_g} N_B\, \dot{d}_B + \sum\limits_{B\in\eta_g} N_B \dot{g}_B - \sum\limits_{B\in\eta-\eta_g} N_B e_B - \sum\limits_{B\in\eta_g} N_B \dot{g}_B \right) d(\Omega\times I) \\ \int\limits_I \left(\begin{array}{c} \int\limits_{\Omega\times I} \left(\sum\limits_{A\in\eta-\eta_g} N_A c_A \right)\left(\sum\limits_{B\in\eta-\eta_g} N_B \dot{e}_B + \sum\limits_{B\in\eta_g} N_B \ddot{g}_B \right) \\ + \nabla\left(\sum\limits_{B\in\eta-\eta_g} N_B d_B + \sum\limits_{B\in\eta_g} N_B g_B \right)\cdot\nabla\left(\sum\limits_{A\in\eta-\eta_g} N_A c_A \right) d(\Omega\times I) \end{array} \right) dI \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

We factor out the c_A terms from the integrals because they are arbitrary- we can just assume that the integral of any arbitrary constants will map to other arbitrary constants. We can also take the basis weights out of the gradients in the second row, because we assume our weights are constant over a basis function- that is to say, $\nabla N_x d_x = d_x \nabla N_x$. We use similar reasoning for integrals of basis function weights over the spatial domain Ω. We can then define matrices to simplify our notation:

$$M_{AB} = \int_\Omega N_A N_B d\Omega$$

$$K_{AB} = \int_\Omega \nabla {N_A}^T \nabla N_B d\Omega$$

$$F_A = -\left(\sum_{B\in\eta_g} M_{AB} \ddot{g}_B + \sum_{B\in\eta_g} K_{AB} g_B \right)$$

Where A and B refer to the rows and columns, respectively, of the matrix referring to (e.g. M_{AB} refers to the entry in row A and column B).

This allows us to represent our equation as:

$$\begin{bmatrix} \vec{c}^T \int_I \ M\overrightarrow{(\dot{d})} + M\vec{e}\, dI \\ \vec{c}^T \int\limits_I \ M\overrightarrow{(\dot{e})} + K\vec{d} - F\, dI \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Here M is the mass matrix, K is the stiffness matrix, and F is the force matrix. Note that we are now representing the basis weights as vectors that can be multiplied by the basis functions, which are in the M and K terms. We can now choose as many arbitrary $\vec{c}$ as we need to have an equation for each unknown. To make things simple, we select $\vec{c}$ for each basis function, such that each entry in $\vec{c}$ is 1 for the respective basis function, and 0 otherwise. This is the same as requiring that the integrand is equal to 0 for all t and x. Thus, we get:

$$\begin{bmatrix} M\overrightarrow{(\dot{d})} + M\vec{e} \\ M\overrightarrow{(\dot{e})} + K\vec{d} - F \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \overrightarrow{(\dot{d})} \\ \overrightarrow{(\dot{e})} \end{bmatrix} = \begin{bmatrix} M^{-1}(M\vec{e}) \\ -M^{-1}(K\vec{d} - F) \end{bmatrix}$$

We use the theta method to approximate the derivatives for a given time step, so that we can calculate how d will change based on initial conditions and the derivative equations in the above matrix. Using an average of forward and backward Euler approximations, we approximate the values of a variable at the next time step by:

$$\dot{x}\big(x(t + \theta\Delta t)\big) \approx \dot{x}(\theta x(t + \Delta t) + (1 - \theta)x(t))$$

Theta represents the weight that we give the value of the derivative at t vs the value of the derivative at t+Δt, and ranges from 0 (Forward Euler) to 1 (Backward Euler). We approximate $\dot{x}$ is a linear function of x (which should be valid for small time steps). Thus,

$$\dot{x}\big(x(t + \theta\Delta t)\big) \approx \theta\dot{x}(x(t + \Delta t)) + (1 - \theta)\dot{x}(x(t))$$

We use the short hand notation $\dot{x}\big(x(t + \theta\Delta t)\big) \equiv \dot{x}_{i+1}$ and $\dot{x}(x(t)) \equiv \dot{x}_i$:

$$\dot{x}_{i+1} = \theta\dot{x}_{i+1} + (1 - \theta)\dot{x}_i$$

Now we multiply by the time step to calculate the value at i+1:

$$x_{i+1} = x_i + (\,\theta\dot{x}_{i+1} + (1 - \theta)\dot{x}_i)\Delta t$$

We can use the above equation to approximate the additional time step. We plug in the matrix of differential equations into this equation, and move unknown terms to the left and known terms to the right around to get:

$$\begin{bmatrix} Md_{i+1} - \theta\Delta t Me_{i+1} \\ \theta\Delta t K d_{i+1} + Me_{i+1} \end{bmatrix} = \begin{bmatrix} Md_i + (1 - \theta)\Delta t Me_i \\ (\theta - 1)\Delta t K d_i + Me_i + \theta\Delta t F_{i+1} + (1 - \theta)\Delta t F_i \end{bmatrix}$$

We can represent this as:

$$\begin{bmatrix} M & -\theta\Delta t M \\ \theta\Delta t K & M \end{bmatrix}\begin{bmatrix} d_{i+1} \\ e_{i+1} \end{bmatrix} = \begin{bmatrix} M & (1 - \theta)\Delta t M \\ (\theta - 1)\Delta t K & M \end{bmatrix}\begin{bmatrix} d_i \\ e_i \end{bmatrix} + \begin{bmatrix} 0 \\ \theta\Delta t F_{i+1} + (1 - \theta)\Delta t F_i \end{bmatrix}$$

For $\theta = 0$, mass lumping can be used. This is just setting M such that entries are only on the diagonal, and each entry in the ith row will contain the sum across all entries across that row. This speeds up runtime as it is much easier to invert a matrix with only values on the diagonal, than a regular matrix.

Building M and K

Before we can apply the derivative equations, we first need to build the M and K matrices through summing over all of elements and adding to the global M and K matrices the contribution of each element. To make computations repeatable, we define our calculations in local basis coordinates, with each element scaled to an isosceles right triangle with the right angle at the origin, and side lengths of 1. Thus, we transform each triangle in our mesh to the local right triangle, then do calculations in local coordinates, and transform the local contribution back into global coordinates, and stamp the contribution into the global M and K matrices.

The local basis functions which make up an element are:

$$N_1 = 1 - r - s$$

$$N_2 = r$$

$$N_3 = s$$

Where r, and s are the x and y coordinates in the local coordinate system, respectively. We define a matrix B which consists of the gradients of each local basis function with respect to each local coordinate so that we can evaluate the integrand of the K term. To convert to global coordinates, we multiply this by the Jacobian of the local to global transformation. To find this Jacobian, we take the inverse of the Jacobian from global to local coordinates, which is much easier to compute. This Jacobian is just a matrix consisting of the global differences of the nodes in an element:

$$\mathrm{J} = \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - x_1 \end{bmatrix}$$

This can be shown by differentiating the function which transforms global to local coordinates, which is just given by scaling each segment on the global triangular element to its corresponding segment onto the local unit triangular element.

To build M and K in local coordinates, we multiply by the determinant of the transformation (this is required from the integral chain rule. Because we have linear basis functions, our integrand for K does not change over each element, so we do not need to evaluate an integral. Thus, our integrand for the K contribution for each element becomes:

$$K_e = \int_{\Omega_e} \nabla N_A \cdot \nabla N_B d\Omega = \ .5 * B^T J^{-1}[J^{-1}]^T B * \det(J)$$

To evaluate the local integral of M, we use tabulated gauss quadrature values for our unit right triangle. We simplify the M contribution for each element to:

$$M_e = \int_{\Omega_e} N_A N_B d\Omega = \frac{\det(J)}{24}[2\ 1\ 1;\ 1\ 2\ 1;\ 1\ 1\ 2]$$

M_e and K_e are calculated for each element and put into the global M and K matrices through the information in the connectivity matrix.

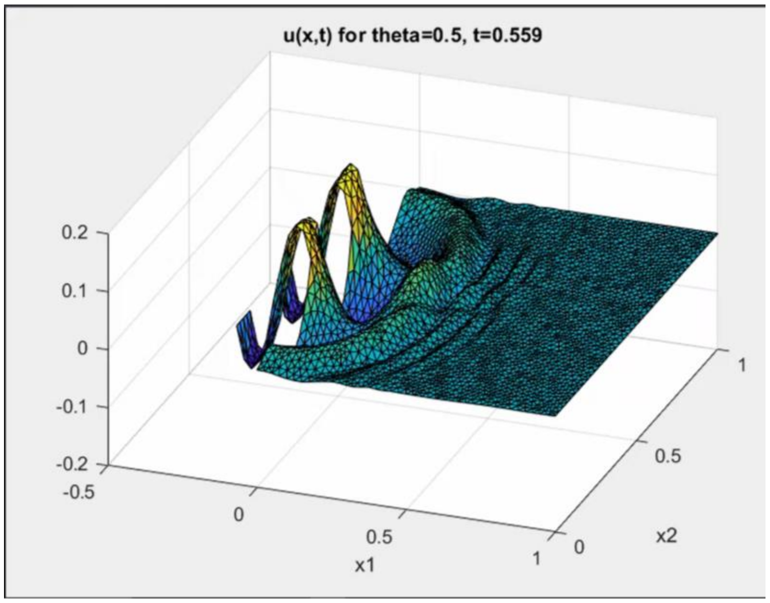
After assembling M and K, we calculate each row of F by summing up the respective row in M and K, and multiplying by the double derivative of the Dirichlet condition, and the Dirichlet function, respectively, both evaluated at the time. To make computation faster, we sum each row of M and K outside of the time loop (since neither M nor K depend on time), and change F only based on how the Dirichlet and Dirichlet double derivative change with each time step.

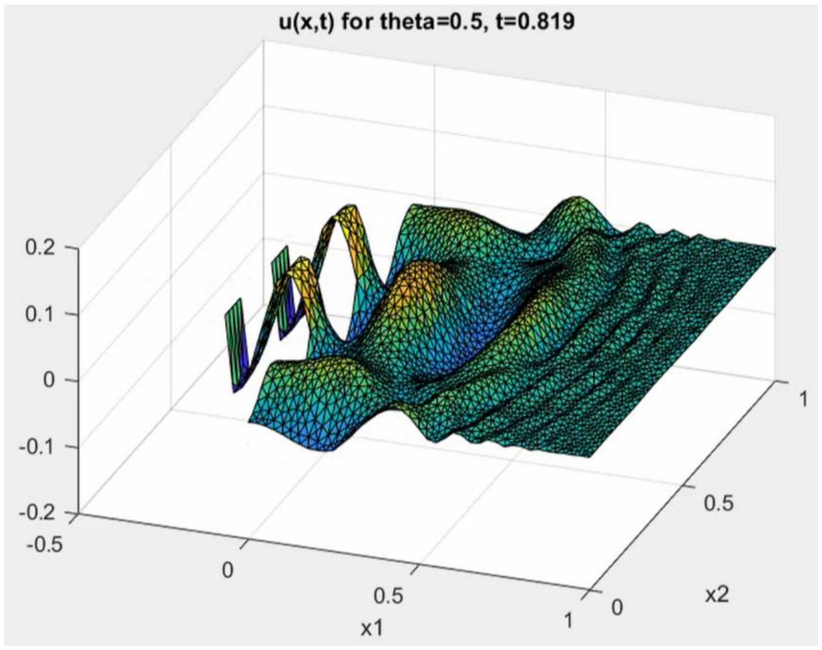
<u>General Algorithm</u>

We now implement this math into MATLAB. We first define all given variables, then define some necessary variables (such as the derivatives of Dirichlet functions, the B matrix, and variables for videotaping the output). We then iterate over each element, adding the element contributions to the global K and M matrices. After we have assembled the M and K matrices, we calculate F for the initial conditions, and solve the derived equation for the d vector, which represents the solutions for our function. We then iterate through each time step, first calculating a new F, then solving for the basis function weights based on the previous calculated solution (for the first time step, the initial values were used). Then we graph this and send it to an output matrix, which holds the d values for each time step that is to be included in the output video. We generate videos for each trial, and test accuracy by seeing how well the value of the function resembles the values of the Dirichlet nodes at different time steps. We also compare how each approximation conserves energy by setting the Dirichlet function to 0 after t=3, so that we can observe how the solution behaves when we do not add energy to the solution.

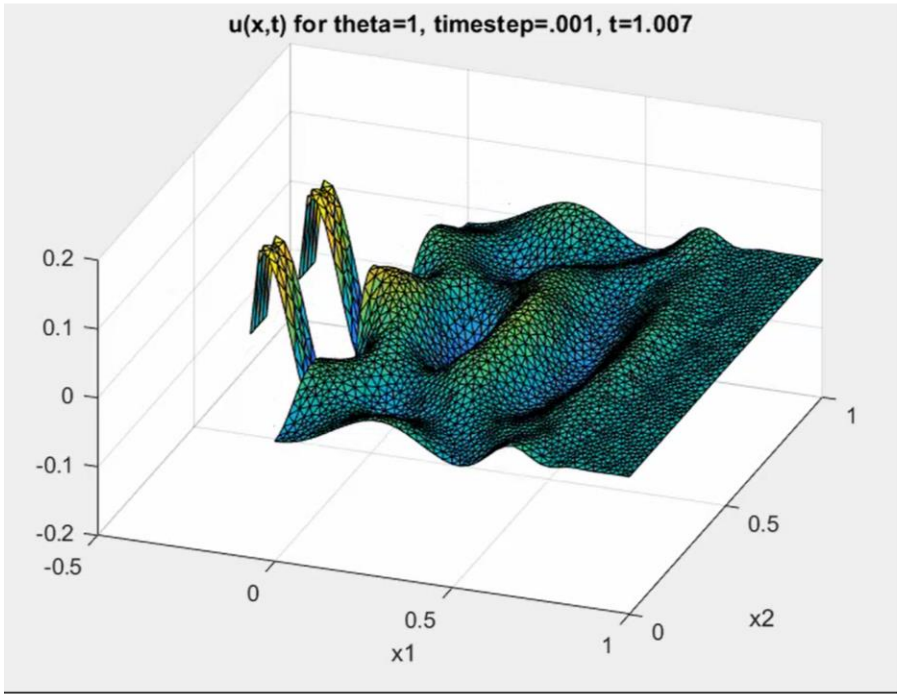
Results

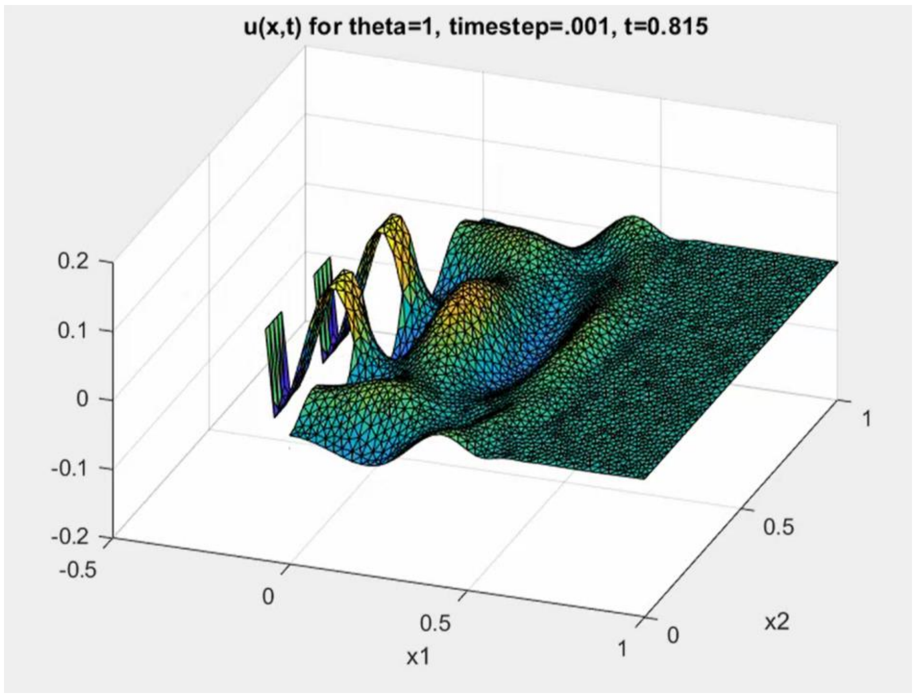
$\underline{\theta = 0.5}$: A timestep of **.001** was required to get a reasonable approximation. Though our solution is likely more accurate for this approximation than for other values of θ (which can be shown by a Taylor series expansion of our evaluation method), we have large values for u than for the other trials. Even within t=0.5, we can see that the values have already reached above 0.1, which is more than the amplitude of the Dirichlet conditions. Though this is expected for later time steps where the waves have a chance to bounce back and cause constructive interference, we can see that this is happening right away. This can be fixed through shorter time steps, or a higher order Taylor Series approximation for each timestep. The second snapshot shows a lot of ripples towards the right side- this approximation did not result in a smooth solution.





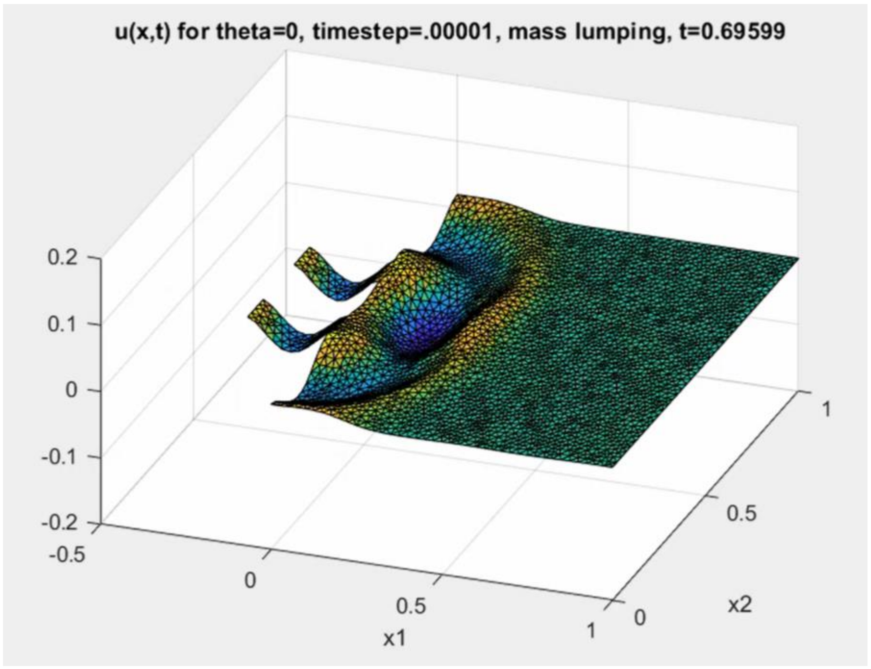
$\underline{\theta = 1:}$ Using the same timestep of **.001**, we see that the initial wave gives slightly higher values than the previous approximation, meaning it is less accurate. However, we can see from the second photo that there are less ripples near the same time value, implying that this method might be useful for smooth expected solutions.

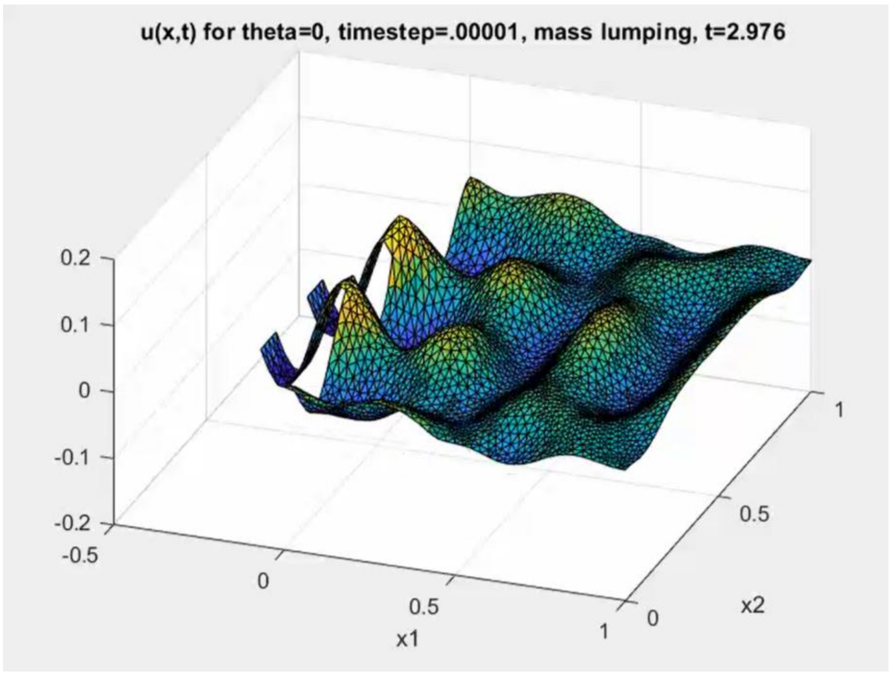


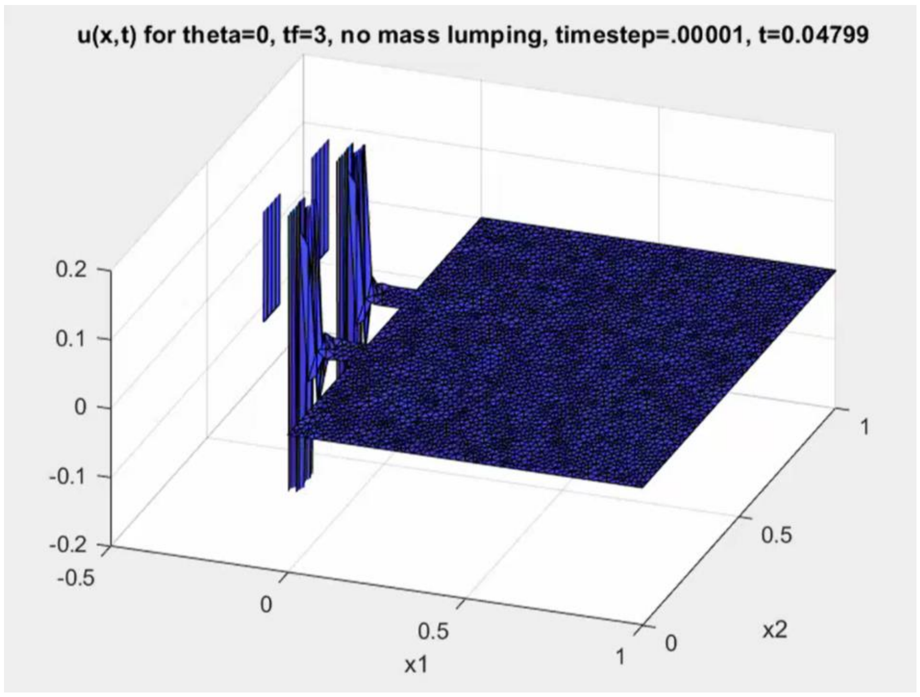


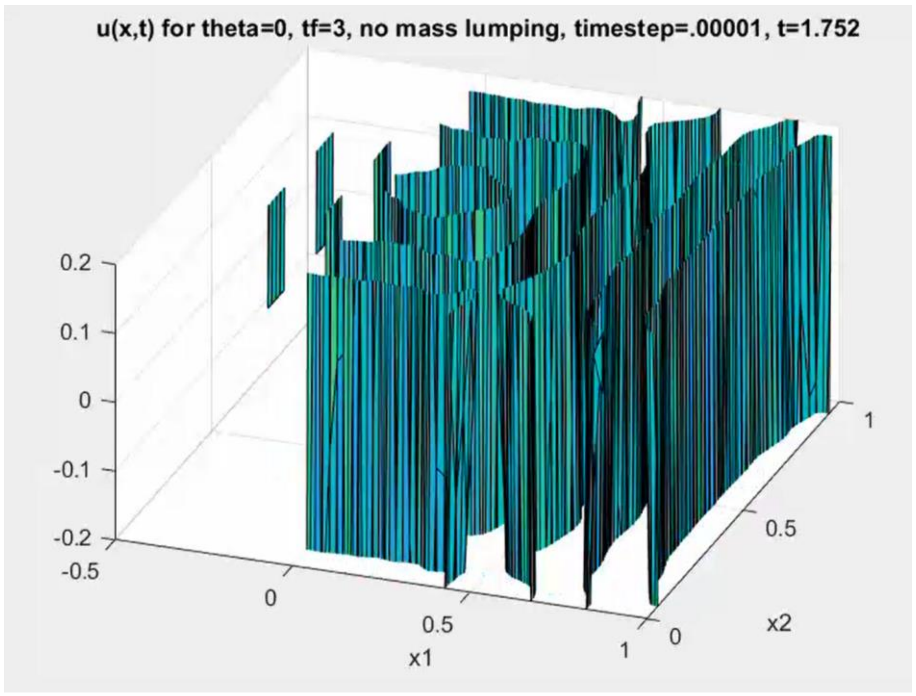
<u>$\theta = 0$:</u>This approximation required much shorter time steps to converge, though each calculation took slightly less time than for the previous approximations due to mass lumping. A timestep of **0.00001** was required to reach a reasonably convergent solution with mass lumping. The solution without mass lumping took a lot longer to compute (on the order of a few minutes), and still did not converge for the same timestep. Even though analytically the same solution is represented whether mass lumping is used or not, mass lumping required fewer calculations, which resulted in a smaller total error (because each calculation contained small inaccuracies in representing the tiny timestep, leading to ill conditioning).

The first snapshot below shows that there are still less ripples than for the θ =0.5 trial. This solution seems to be more accurate than both previous trials, though this can be explained because the time step was 2 orders of magnitude smaller than that used in the previous times. However, the solution with mass lumping had a similar run time even using the smaller time step values. The solution without mass lumping resembled the actual solution right before it diverges completely for larger values of t, but was not favorable as it still diverged even for very small time steps.

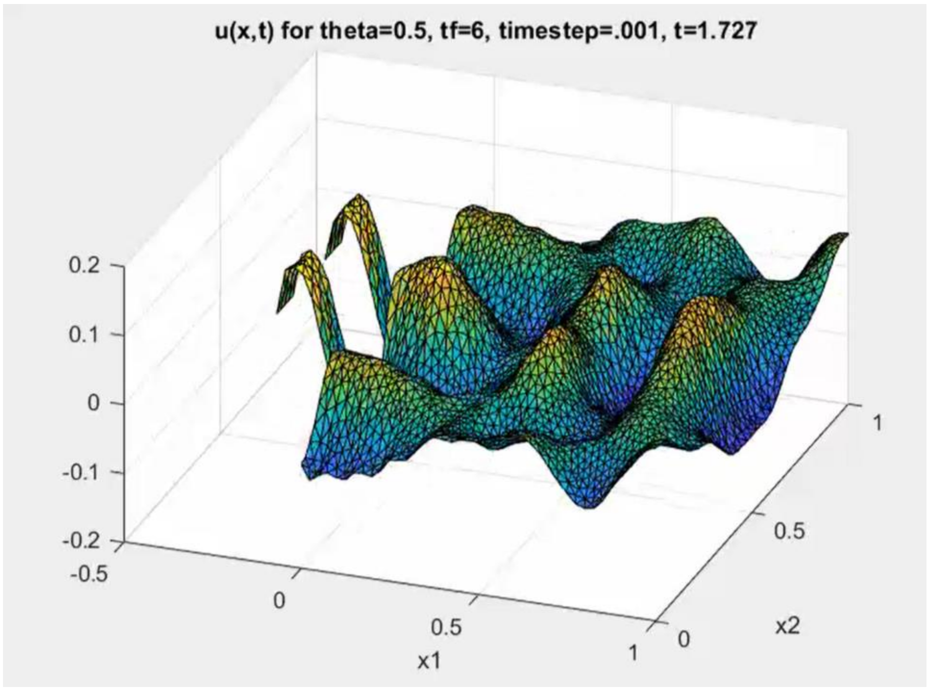

u(x,t) for theta=0, timestep=.00001, mass lumping, t=0.69599
0.2
0.1
0
-0.1
-0.2
-0.5
0
0.5
1
x1
0
0.5
1
x2


u(x,t) for theta=0, timestep=.00001, mass lumping, t=2.976
0.2
0.1
0
-0.1
-0.2
-0.5
0
0.5
1
x1
0
0.5
1
x2

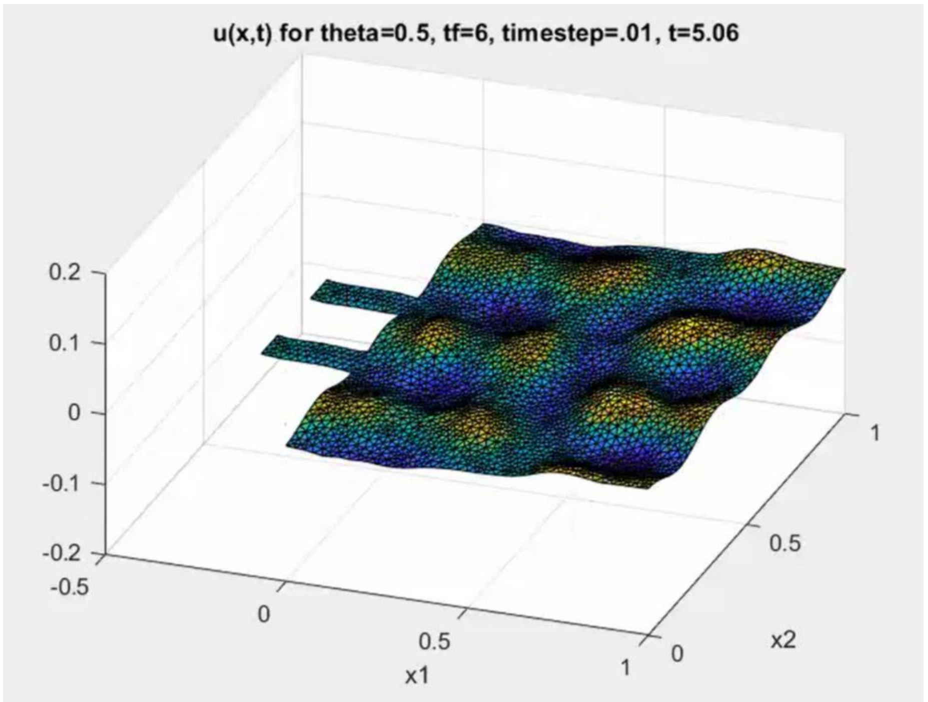




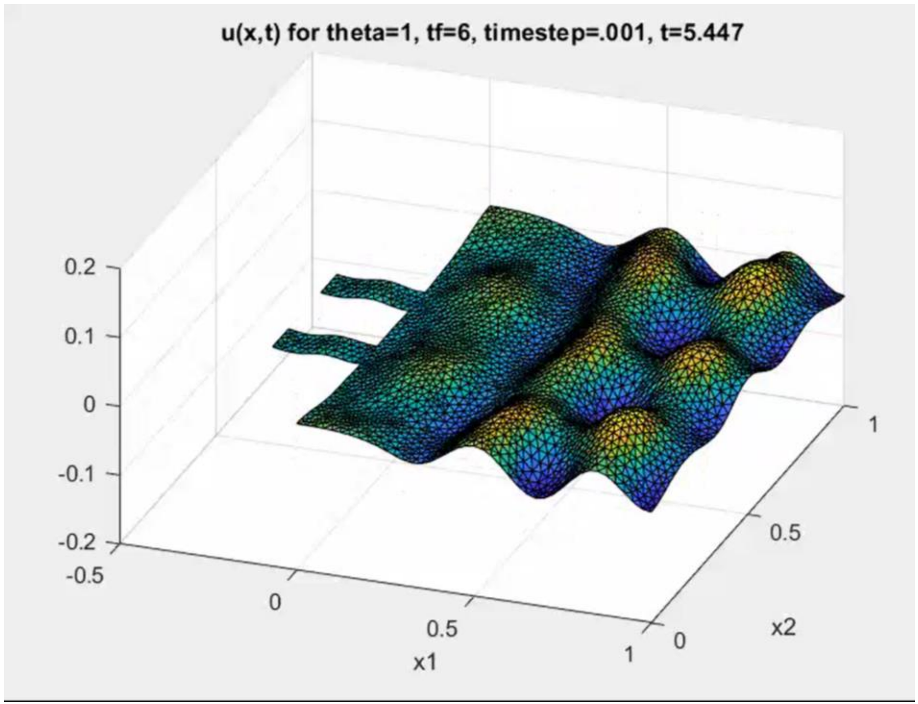
$\underline{Extended\ Integration\ for\ Conservation\ of\ mass: \theta = 1\ vs\ \theta = 0.5}$: We can see that that for θ =1, less energy is conserved. This is very visible for the timestep being **.01**, where the Dirichlet conditions do not even cause ripples towards the other end of the domain. For θ = 0.5, we can see that the waves lasted longer and had a higher amplitude after the Dirichlet conditions were shut off, suggesting that the $\theta = 0.5$ approximation lead to a higher conservation of energy.

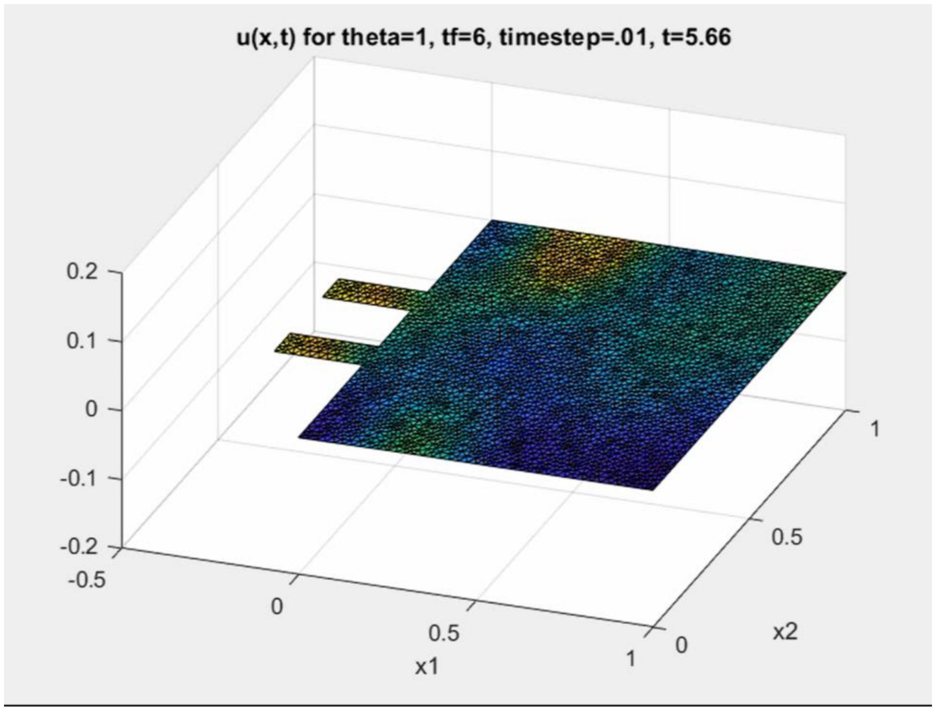


u(x,t) for theta=0.5, tf=6, timestep=.001, t=1.727
0.2
0.1
0
-0.1
-0.2
-0.5
0
0.5
1
x1
0
0.5
1
x2



u(x,t) for theta=0.5, tf=6, timestep=.01, t=5.06
0.2
0.1
0
-0.1
-0.2
-0.5
0
0.5
1
x1
0
0.5
1
x2


u(x,t) for theta=1, tf=6, timestep=.001, t=5.447
0.2
0.1
0
-0.1
-0.2
-0.5
0
0.5
1
x1
0
0.5
1
x2


u(x,t) for theta=1, tf=6, timestep=.01, t=5.66
0.2
0.1
0
-0.1
-0.2
-0.5
0
0.5
1
x1
0
0.5
1
x2

Discussion

The use of different approximation methods illustrates the tradeoffs between accuracy and meeting of requirements. Theta=0.5 was most accurate for a given timestep, but took slightly longer to compute than theta=0, and had a lot more ripples- which might not be optimal for a smoother expected solution. Theta=1 was smooth and was accurate, but did not satisfy the conservation of energy. Theta=0 can be run quickly using mass lumping, but can be very inaccurate for small time steps. This also illustrates the need for efficient code- doing calculations only where necessary reduces the total required time, and allows smaller time steps for a given time period.