Simulation of 2D Quantum Wavepacket Dynamics using the Shrödinger equation and the Finite Element Method in Julia

Author: Victor Emmanuel Meimaris A.M: 58855

Democritus University of Thrace, Xanthi, Greece

Date: March 25, 2023 started / May 30, 2025 written document

Abstract

This document outlines a Julia-based numerical simulation framework for studying the time evolution of a two-dimensional quantum mechanical wavepacket. The core of the simulation relies on solving the Time-Dependent Schrödinger Equation (TDSE). Spatial discretization is achieved using the Finite Element Method (FEM) with bilinear quadrilateral elements, particularly tailored for an L-shaped domain. The temporal evolution is handled by the robust and unitary Crank-Nicolson scheme. The script is modular, allowing for different potential energy landscapes and initial wavefunction configurations. Key aspects such as mesh generation, FEM matrix assembly, initial state definition, and time-stepping solution are discussed, along with the influence of various simulation parameters on the output.

1. Introduction

The study of quantum mechanical systems is fundamental to understanding the behavior of matter at atomic and subatomic scales. The Time-Dependent Schrödinger Equation (TDSE) governs the evolution of a quantum system's wavefunction, from which all observable properties can be derived. Due to the complexity of analytical solutions, especially for systems with non-trivial potentials or geometries, numerical methods are indispensable.

This Julia script provides a framework to simulate the dynamics of a single quantum particle in a two-dimensional space. The primary objectives are to accurately model the spatial characteristics of the wavefunction using the Finite Element Method (FEM), propagate the wavefunction in time using the Crank-Nicolson method, known for its stability and preservation of probability and to visualize the evolution of the probability density of the particle under various user-defined potential fields and initial conditions.

2. Physical and Mathematical Framework

2.1. The Time-Dependent Schrödinger Equation (TDSE)

The cornerstone of this simulation is the 2D TDSE:

$$i\hbar\frac{\partial\Psi(\mathbf{r},t)}{\partial t} = \hat{H}\Psi(\mathbf{r},t)$$

where:

- The complex-valued wavefunction of the particle at position $\mathbf{r} = (x, y)$ and time t is represented as: $\Psi(\mathbf{r}, t)$
- The reduced Planck constant is: $\hbar$
- The Hamiltonian operator, which describes the total energy of the system, is: $\hat{H}$

  For a single particle of mass m in a potential $V(\mathbf{r})$, it is given by:
  $$\hat{H} = -\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r}) = -\frac{\hbar^2}{2m}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) + V(x, y)$$

2.2. Numerical Discretization

To solve the TDSE numerically, we discretize both space and time.

2.2.1. Spatial Discretization: Finite Element Method (FEM)

The FEM is employed for spatial discretization. This involves:

1. **Weak Formulation:** Multiplying the TDSE by a test function $N_i(\mathbf{r})$ and integrating over the spatial domain Ω:
   $$\int_\Omega N_i\left(i\hbar\frac{\partial\Psi}{\partial t}\right)d\Omega = \int_\Omega N_i\left(-\frac{\hbar^2}{2m}\nabla^2\Psi + V\Psi\right)d\Omega$$

   Applying Green's first identity (integration by parts) to the Laplacian term:
   $$\int_\Omega N_i\left(-\frac{\hbar^2}{2m}\nabla^2\Psi\right)d\Omega = \int_\Omega \frac{\hbar^2}{2m}\nabla N_i\cdot\nabla\Psi d\Omega - \oint_{\partial\Omega} N_i\frac{\hbar^2}{2m}\frac{\partial\Psi}{\partial n}dS$$

   For Dirichlet boundary conditions ($\Psi = 0$ on $\partial\Omega$), the boundary integral for Neuman or Robin conditions vanishes.

2. **Domain Meshing:** The 2D domain is divided into smaller, non-overlapping quadrilateral elements. The script is specifically designed to handle L-shaped domains.
3. **Wavefunction Approximation:** Within each element e, the wavefunction $\Psi^e(x, y, t)$ is approximated as a linear combination of basis (or shape) functions $N_j(x, y)$ and time-dependent nodal values $\psi_j(t)$:

   $$\Psi^e(x, y, t) \approx \sum_{j=1}^{4} N_j(x, y)\psi_j(t)$$

   The script uses bilinear shape functions for quadrilateral elements.
4. **Element & Global Matrices:** Substituting the approximation into the weak form and summing over all elements leads to a system of ordinary differential equations in time:

   $$i\hbar M\frac{d\vec{\psi}}{dt} = H\vec{\psi}$$

   where:
   - $\vec{\psi}(t)$ is the vector of nodal wavefunction values.
   - M is the global **Mass Matrix**:

     $$M_{ij} = \int_\Omega N_i N_j d\Omega$$

   - H is the global **Hamiltonian Matrix**:

     $$H_{ij} = \int_\Omega \left(\frac{\hbar^2}{2m}\nabla N_i \cdot \nabla N_j + N_i V N_j \right) d\Omega$$

     This H matrix is referred to as tempo in the script before further scaling.

2.2.2. Temporal Discretization: Crank-Nicolson Method

The Crank-Nicolson scheme is used for time integration. It is an implicit method, unconditionally stable for this type of problem, and unitary (conserves probability). It approximates the time derivative as:

$$\frac{d\vec{\psi}}{dt} \approx \frac{\vec{\psi}^{n+1} - \vec{\psi}^n}{\Delta t}$$

and averages the right-hand side at times t_n and t_{n+1}:

$$i\hbar\frac{\vec{\psi}^{n+1}-\vec{\psi}^{n}}{\Delta t}=\frac{1}{2}(H\vec{\psi}^{n+1}+H\vec{\psi}^{n})$$

Rearranging this gives:

$$\left(M-\frac{\Delta t}{2i\hbar}H\right)\vec{\psi}^{n+1}=\left(M+\frac{\Delta t}{2i\hbar}H\right)\vec{\psi}^{n}$$

Or, equivalently, as implemented by scaling H first:

$$\left(M+\frac{i\Delta t}{2\hbar}H\right)\vec{\psi}^{n+1}=\left(M-\frac{i\Delta t}{2\hbar}H\right)\vec{\psi}^{n}$$

This can be written as:

$$A\vec{\psi}^{n+1}=B\vec{\psi}^{n}$$

where $A=M+\frac{i\Delta t}{2\hbar}H$ and $B=M-\frac{i\Delta t}{2\hbar}H$. At each time step Δt, this linear system is solved for $\vec{\psi}^{n+1}$. Consequently, both A and B are **complex-valued** matrices. They inherit the **sparsity** from M and H. These matrices are generally **non-Hermitian**. Because they are non-Hermitian, they are **not symmetric positive definite (SPD)** in the standard sense, nor are they generally symmetric in the complex sense unless H is zero. Furthermore, A and B are **not necessarily diagonally dominant**, as the off-diagonal elements of H, scaled by the complex factor, can be significant relative to the diagonal entries which also include contributions from M. The specific conditioning and other numerical properties of A and B depend on the characteristics of M, H, and the time step Δt. The matrix A must be **invertible** to solve the linear system at each time step, a condition satisfied by the unconditionally stable Crank-Nicolson method. Given that the system matrices A and B are not Symmetric Positive Definite (SPD), iterative methods designed for such systems, like the Conjugate Gradient method, are not directly applicable. Consequently, a robust

solver for non-symmetric systems, such as the **Biconjugate Gradient Stabilized (BiCGSTAB) method**, is an appropriate choice for solving the system at each time step.

Furthermore, considering that these matrices are also **not necessarily diagonally dominant**, the convergence of simpler iterative schemes like the damped Jacobi method, often employed as a smoother in multigrid approaches, cannot be guaranteed. This lack of guaranteed convergence for basic smoothers further underscores the suitability of employing a more general and robust iterative solver like BiCGSTAB for the overall solution of the linear system.

3. Code Implementation Details (Module Functions)

The script uses SparseArrays.jl for efficient handling of large matrices and Plots.jl for visualization (with gr() used for animations).

3.1. Constants

- h_bar = 0.65821220e-3: Reduced Planck constant in eV·ps.
- m = 9.1093837e-31 * (10e25 / 1.602117662): Mass of an electron, converted to units of $(eV \cdot ps^2)/nm^2$.
- gamma = (h_bar^2)/(2*m): A convenient constant appearing in the kinetic energy term, in units of $eV \cdot nm^2$.

3.2. grid Function

Generates the computational mesh for an L-shaped domain, node coordinates, element connectivity (local-to-global mapping l2g), and identifies boundary nodes. The L-shape is constructed by defining node numbering for three rectangular subdomains: a top-left square (a), a bottom-left square (b), and a bottom-right square (c). max_length is adjusted to ensure that the number of divisions along an axis (domain_length / max_length) is an even integer, which is a requirement for the specific FEM element structuring or subsequent solver logic. Boundary nodes are explicitly collected into a Set for easy application of Dirichlet boundary conditions.

functions.jl > {} Functions > multigrid_vic_hierarchy

```julia
#= Victor Emmanuel Meimaris 23/3/25: Creating Module with all functions used on main.jl, I will be
explaining the numbers in my report. =#
module Functions
using SparseArrays, LinearAlgebra, Arpack, Plots, SparseArrays, IncompleteLU

plotlyjs() # Enable PlotlyJS backend for interactivity
#gr() # Backend for animations change to this for stable animations as well as low memory images when testing

# Electron constants for my domain in nanometers and femtoseconds
const h_bar = 0.65821220e-3 # h_bar actual value in eV * ps
const m = 9.1093837e-31 * (10e25 / 1.602117662 )# mass of electron in kg is 9.1093837e-31 but we want in (eV * ps^2) / nm^2
const gamma = (h_bar^2)/(2*m) # in eV * nm^2

# Creating Coordinate/Mesh function, returns grid ,l2g Matrix and number of elements / nodes.
function grid(domain::Tuple{Real, Real}, max_length::Float64) # num_of_squares_x::UInt16, num_of_squares_y::UInt16

    # Define domain length in any axis since x and y have the same domain
    domain_length = domain[2] - domain[1]

    # Filtering input so that it's valid

    # This division has to be a multiple of 2 in order for FEM to work so we modify max_length accordingly
    temp = domain_length / max_length
    if temp < 2
        max_length = domain_length / 2
    elseif temp % 2 != 0
        temp = round(Int, temp)  # Round to the nearest integer
        if temp % 2 != 0
            temp += 1
        elseif temp == 2 # Added this for occasions where temp rounds to 2 so it needs +2
            temp += 2
        end
        max_length = domain_length / temp
    end
    temp = round(Int, temp) # Case for if temp = 2.0

    # Length of xg and yg
    lengthr = Int64(domain_length / max_length + 1)

    # Deriving number of squares based on length provided
    num_of_squares_y = temp
    num_of_squares_x = temp # This could be different to y if multiple of 2

    # Number of elements and nodes
    noe = Int((num_of_squares_y * num_of_squares_x) * 0.75) # Number of elements
    nop = num_of_squares_x + num_of_squares_y + noe + 1# Number of nodes

    nx_half = (num_of_squares_x ÷ 2) + 1
    ny_half = (num_of_squares_y ÷ 2) + 1
    # a->top left square
    a = reshape(1:nx_half * ny_half, nx_half, ny_half)
    a = transpose(a)

    # b->bottom big rectnagle
    b = reshape(a[end,1]:nop, num_of_squares_x + 1, (num_of_squares_y ÷ 2) + 1)
    b = transpose(b)

    # c->bottom right square
    c = b[:, nx_half:end]
    # b -> becomes bottom left square
    b = b[:, 1:nx_half]

    # Keep boundary nodes for boundary conditions in the solution later without any order
    boundary_nodes = Set{Int}() # Initialise empty set for boundary nodes for fast look ups to detect number of boundary nodes in each elemer
    union!(boundary_nodes, @view a[1, :])
    union!(boundary_nodes, @view a[:, 1])
    union!(boundary_nodes, @view a[:, end])
    union!(boundary_nodes, @view b[:, 1])
    union!(boundary_nodes, @view b[end, :])
    union!(boundary_nodes, @view c[1, :])
    union!(boundary_nodes, @view c[end, :])
    union!(boundary_nodes, @view c[:, end])
```

```
    # Generate coordinate range
    x = range(domain[1], domain[2], num_of_squares_x + 1)# need boundary nodes for dirichlet, elements with two sides with boundary nodes and elem with 1 side
    y = range(domain[2], domain[1], num_of_squares_y + 1)#if boundary nodes in l2g ==2 then 1 side elseif ==3 2 sides else no sides create flag or smthing to not do ifs all the time

    # Generate coordinates
    xc = [xi for xi in x, _ in y]  # Each row is xi, number of rows = length of y
    yc = [yi for _ in x, yi in y]  # Each column is yi, number of collumns = length of x

    # Setting coordinates on a grid
    coords = zeros(nop, 2)
    temp = 1
    index = 1
    for i = 1:(num_of_squares_y ÷ 2)
        @inbounds for j = 1:(num_of_squares_x ÷ 2 + 1)
            coords[index, 1] = xc[temp]
            coords[index, 2] = yc[temp]
            temp += 1
            index += 1
        end
        temp += (num_of_squares_y ÷ 2) # I am not adding +1 to temp or index because in last iteration of j loop it was already done
    end

    coords[index:end, 1] = xc[temp:length(xc)]
    coords[index:end, 2] = yc[temp:length(yc)]

    # Define local to global map " l2g "
    l2g = zeros(Int, noe, 4) # Each element has 4 nodes
    index = 1
    # For top square
    for i = 1:num_of_squares_x ÷ 2
        @inbounds for j = 1:num_of_squares_y ÷ 2
            l2g[index, :] = [a[i + 1, j], a[i + 1, j + 1], a[i, j + 1], a[i, j]]
            index +=1
        end
    end
    # For bottom left square
    for i = 1:(num_of_squares_y ÷ 2)
        @inbounds for j = 1:(num_of_squares_x  ÷ 2)
            l2g[index, :] = [b[i + 1, j], b[i + 1, j + 1], b[i, j + 1], b[i, j]]
            index += 1
        end
    end
    # For bottom right square overlapping elements
    for i = 1:(num_of_squares_y ÷ 2)
        @inbounds for j = 1:(num_of_squares_x  ÷ 2)
            l2g[index, :] = [c[i + 1, j], c[i + 1, j + 1], c[i, j + 1], c[i, j]]
            index += 1
        end
    end
    return coords, l2g, noe, nop, boundary_nodes, max_length, lengthr, a, b, c, nx_half, ny_half
```

3.3. fem_matrices Function

Assembles the FEM matrices A and B required for the Crank-Nicolson scheme.

Defines the four bilinear shape functions for a quadrilateral element in local coordinates (ξ, η). Iterates over each element (e = 1:noe):

- Retrieves element node coordinates (xe, ye).
- Defines coordinate transformation $x(\xi, \eta)$, $y(\xi, \eta)$.
- Calculates the Jacobian matrix components (J11, J12, J21, J22) and its determinant detJ.
- Calculates derivatives of shape functions with respect to global coordinates (θNx, θNy).
- Computes local element matrices:
  - Stiffness & Potential part (KV):

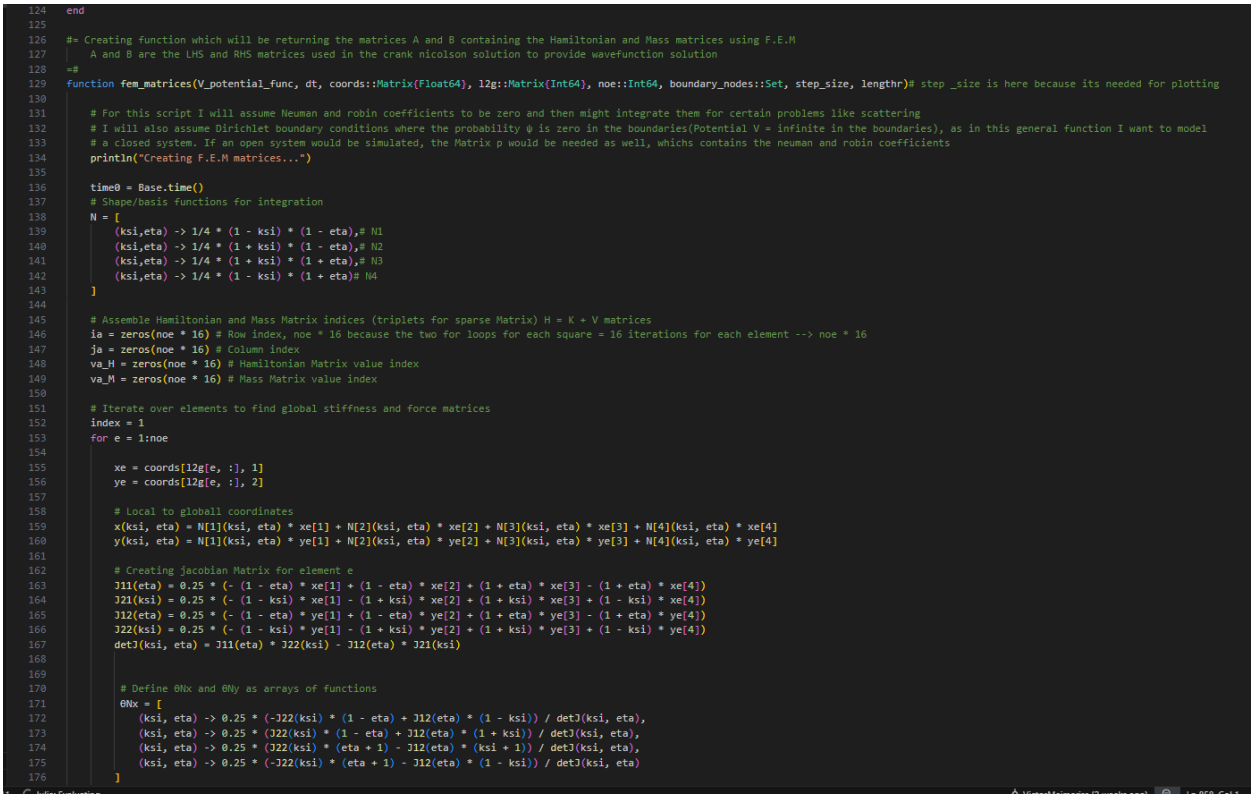
    $$\int_e (\gamma(\nabla N_i \cdot \nabla N_j) + N_i V N_j)\,|detJ| d\xi d\eta$$

  - Mass part (L):

    $$\int_e N_i N_j |detJ| d\xi d\eta$$

These integrals are evaluated using gauss_quad_2D(integrand, 3) (3-point Gaussian quadrature).

1. **Assembly:** The local matrices KV and L are assembled into global sparse matrices H_global (referred to as H initially, then tempo) and M_global (referred to as M).



```
end

#= Creating function which will be returning the matrices A and B containing the Hamiltonian and Mass matrices using F.E.M
    A and B are the LHS and RHS matrices used in the crank nicolson solution to provide wavefunction solution
=#
function fem_matrices(V_potential_func, dt, coords::Matrix{Float64}, l2g::Matrix{Int64}, noe::Int64, boundary_nodes::Set, step_size, lengthr)# step _size is here because its needed for plotting

    # For this script I will assume Neuman and robin coefficients to be zero and then might integrate them for certain problems like scattering
    # I will also assume Dirichlet boundary conditions where the probability ψ is zero in the boundaries(Potential V = infinite in the boundaries), as in this general function I want to model
    # a closed system. If an open system would be simulated, the Matrix p would be needed as well, whichs contains the neuman and robin coefficients
    println("Creating F.E.M matrices...")

    time0 = Base.time()
    # Shape/basis functions for integration
    N = [
        (ksi,eta) -> 1/4 * (1 - ksi) * (1 - eta),# N1
        (ksi,eta) -> 1/4 * (1 + ksi) * (1 - eta),# N2
        (ksi,eta) -> 1/4 * (1 + ksi) * (1 + eta),# N3
        (ksi,eta) -> 1/4 * (1 - ksi) * (1 + eta)# N4
    ]

    # Assemble Hamiltonian and Mass Matrix indices (triplets for sparse Matrix) H = K + V matrices
    ia = zeros(noe * 16) # Row index, noe * 16 because the two for loops for each square = 16 iterations for each element --> noe * 16
    ja = zeros(noe * 16) # Column index
    va_H = zeros(noe * 16) # Hamiltonian Matrix value index
    va_M = zeros(noe * 16) # Mass Matrix value index

    # Iterate over elements to find global stiffness and force matrices
    index = 1
    for e = 1:noe

        xe = coords[l2g[e, :], 1]
        ye = coords[l2g[e, :], 2]

        # Local to globall coordinates
        x(ksi, eta) = N[1](ksi, eta) * xe[1] + N[2](ksi, eta) * xe[2] + N[3](ksi, eta) * xe[3] + N[4](ksi, eta) * xe[4]
        y(ksi, eta) = N[1](ksi, eta) * ye[1] + N[2](ksi, eta) * ye[2] + N[3](ksi, eta) * ye[3] + N[4](ksi, eta) * ye[4]

        # Creating jacobian Matrix for element e
        J11(eta) = 0.25 * (- (1 - eta) * xe[1] + (1 - eta) * xe[2] + (1 + eta) * xe[3] - (1 + eta) * xe[4])
        J21(ksi) = 0.25 * (- (1 - ksi) * xe[1] - (1 + ksi) * xe[2] + (1 + ksi) * xe[3] + (1 - ksi) * xe[4])
        J12(eta) = 0.25 * (- (1 - eta) * ye[1] + (1 - eta) * ye[2] + (1 + eta) * ye[3] - (1 + eta) * ye[4])
        J22(ksi) = 0.25 * (- (1 - ksi) * ye[1] - (1 + ksi) * ye[2] + (1 + ksi) * ye[3] + (1 - ksi) * ye[4])
        detJ(ksi, eta) = J11(eta) * J22(ksi) - J12(eta) * J21(ksi)


        # Define θNx and θNy as arrays of functions
        θNx = [
            (ksi, eta) -> 0.25 * (-J22(ksi) * (1 - eta) + J12(eta) * (1 - ksi)) / detJ(ksi, eta),
            (ksi, eta) -> 0.25 * (J22(ksi) * (1 - eta) + J12(eta) * (1 + ksi)) / detJ(ksi, eta),
            (ksi, eta) -> 0.25 * (J22(ksi) * (eta + 1) - J12(eta) * (ksi + 1)) / detJ(ksi, eta),
            (ksi, eta) -> 0.25 * (-J22(ksi) * (eta + 1) - J12(eta) * (1 - ksi)) / detJ(ksi, eta)
        ]
```

```
            (ksi, eta) -> 0.25 * (J21(ksi) * (1 - eta) - J11(eta) * (1 - ksi)) / detJ(ksi, eta),
            (ksi, eta) -> 0.25 * (-J21(ksi) * (1 - eta) - J11(eta) * (1 + ksi)) / detJ(ksi, eta),
            (ksi, eta) -> 0.25 * (-J21(ksi) * (1 + eta) + J11(eta) * (1 + ksi)) / detJ(ksi, eta),
            (ksi, eta) -> 0.25 * (J21(ksi) * (1 + eta) + J11(eta) * (1 - ksi)) / detJ(ksi, eta)
        ]
        # Local Hamiltonian and Mass matrices
        KV = zeros(4, 4)
        L = zeros(4, 4) # Local Mass Matrix

        # Precompute integrands for less allocations
        # Local potential Matrix V integrals
        integrands_V = [
            (ksi, eta) -> N[i](ksi, eta) * N[j](ksi, eta) * V_potential_func(x(ksi, eta), y(ksi, eta)) * abs(detJ(ksi, eta))
            for i in 1:4, j in 1:4
        ]

        # Compute local stiffness Matrix K integrals with αx and αy equal to 1 for the Shrodinger equation
        integrands_K = [
            (ksi, eta) -> gamma * (∂Nx[i](ksi, eta) * ∂Nx[j](ksi, eta) + ∂Ny[i](ksi, eta) * ∂Ny[j](ksi, eta)) * abs(detJ(ksi, eta))
            for i in 1:4, j in 1:4
        ]

        # Compute local Mass Matrix integrals
        integrands_L = [
            (ksi, eta) -> N[i](ksi, eta) * N[j](ksi, eta) * abs(detJ(ksi, eta))
            for i in 1:4, j in 1:4
        ]

        # Add results to local Hamiltonian and Mass Matrix
        for i = 1:4
            @inbounds for j = 1:4
                KV[i, j] = gauss_quad_2D(integrands_K[i, j], 3) + gauss_quad_2D(integrands_V[i, j], 3)
                L[i, j] = gauss_quad_2D(integrands_L[i, j], 3)
            end
        end

        # Assemble global stiffness and Mass matrices indices from the local element matrices
        for i = 1:4
            @inbounds for j = 1:4
                ia[index] = 12g[e, i] # global node index at row i of local matrices
                ja[index] = 12g[e, j] # global node index at collumn j
                va_H[index] = KV[i, j] # global value index at i, j of local Hamiltonian Matrix
                va_M[index] = L[i, j] # global value index at i, j of local Mass Matrix
                index = index + 1;
            end
        end
    end
    # Assembling global Hamiltonian Matrix = global stiff+ global potential, and Mass matrices using indices from loop
    H = sparse(ia, ja, va_H)
    M = sparse(ia, ja, va_M)

    # Enforce dirichlet boundary conditions
    boundary_nodes = collect(boundary_nodes)  # Convert Set to Vector
```

2. **Dirichlet Boundary Conditions:** Rows and columns corresponding to boundary_nodes in H_global and M_global are zeroed out, and the diagonal entry is set to 1. This effectively enforces $\Psi = 0$ at these nodes and ensures the matrices remain well-conditioned.
3. **Crank-Nicolson Matrices:**
    - The Hamiltonian H_global (stored in tempo) is scaled: $H_{\text{scaled}} = H_{\text{global}} \cdot i \cdot \Delta t/(2\hbar)$.
    - $A = M_{\text{global}} + H_{\text{scaled}}$
    - $B = M_{\text{global}} - H_{\text{scaled}}$

- The variable tempo stores the original Hamiltonian matrix $\int (\gamma \nabla N_i \cdot \nabla N_j + N_i V N_j)$, which is used later for energy calculations.

```
    boundary_nodes = collect(boundary_nodes)  # Convert Set to Vector
    for node in boundary_nodes
        H[node, :] .= 0  # Zero row
        H[:, node] .= 0  # Zero column
        M[node, :] .= 0
        M[:, node] .= 0
        H[node, node] = 1  # Set diagonal
        M[node, node] = 1
    end
    #= Tests
    println("M symmetric?",norm(M-M')<1e-10)
    println("H symmetric?",norm(H-H')<1e-10)
    println("Is H hermittian?", ishermitian(H))

    # Test H for positive semi-definiteness
    try
        lambda_min = eigs(H, nev=1, which=:SR)[1][1]
        lambda_max = eigs(H, nev=1, which=:LR)[1][1]
        if real(lambda_min) >= -1e-10
            println("H is positive semi-definite, smallest eigenvalue: ", lambda_min)
            println("H highest eigenvalue: ",lambda_max)
        else
            println("H is not positive semi-definite, smallest eigenvalue: ", lambda_min)
        end
    catch e
        println("Error computing eigenvalues of H: ", e)
    end
    try
        chol_M = cholesky(M)
        println("M is positive definite")
    catch e
        println("M is not positive definite: ", e)
    end=#

    # Multiply H by i * dt / (2*h_bar) to include everything in H and hold previous value in tempo for latter use
    tempo = H
    H = H * im * dt / (2 * h_bar)

    # Matrices A and B for RHS and LHS
    A = H + M
    B = M - H

    time1 = Base.time()

    time = time1 - time0
    time = round(time, digits = 2)
    println("Finished creating matrices from $noe elements in $time seconds")

    # Now our system looks like this: Aψ(n+1) = Bψn, almost ready for the solution with crank nicolson
    return A, B, lengthr, dt, boundary_nodes, tempo, step_size
end
```

3.4. wavefunction Function

Defines the initial wavefunction $\Psi(x, y, t = 0)$. Forms Gaussian wave packet modulated by a plane wave:

$$\Psi(x, y; x_0, y_0, \sigma, k_x, k_y) = \left(e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}} \right) \cdot \left(e^{-i(k_x x + k_y y)} \right)$$

- **Parameters:**
  - x_0, y_0: Initial center coordinates of the Gaussian.
  - σ: Standard deviation, controlling the initial spatial spread of the packet.
  - k_x, k_y: Wavevector components, determining the initial momentum and direction of propagation.

```
# Creating inital wavefunction equation Ψ0 to solve with function solution
function wavefunction(x, y ; x0, y0, sigma, kx::Float64, ky::Float64)
    gaussian = exp(- ((x - x0)^2 + (y - y0)^2 ) / (2 * sigma^2))
    plane_wave = exp(-im * (kx * x + ky * y))
    return gaussian * plane_wave
end
```

3.5. V_function Function

Provides a selection of potential energy landscapes $V(x, y)$.

- **Options (V_flag):**
  1. V_flag == 1: Infinite potential well (constant potential, default $V_0 = 0.0$). The "infinite" walls are handled by Dirichlet boundary conditions.
  2. V_flag == 2: Circular potential barrier of height V0 centered at (x0, y0) with radius r.
  3. V_flag == 3: Circular potential well of depth $-V_0$ centered at (x0, y0) with radius r.

```
# Creating function for potential V and its flags
function V_function(V_flag::Int64, V0 = 0.0, x0 = 0.0, y0 = -0.5, r = 0.2)
    if V_flag == 1 # Infinite potential well with bottom at 0.0 or at selected V0
        return (x,y) -> V0 # Which equals 0.0 at default
    elseif V_flag == 2 # Barrier of potential V0 at area around x0 and y0"default center at (-0.3, -0.5) with radius 0.1"
        return (x, y) -> ((x - x0)^2 + (y - y0)^2 <= r^2 ? V0 : 0.0)
    elseif V_flag == 3 # Well of potential V0
        return (x, y) -> ((x - x0)^2 + (y - y0)^2 <= r^2 ? -V0 : 0.0)
    end
end
```

3.6. solution Function

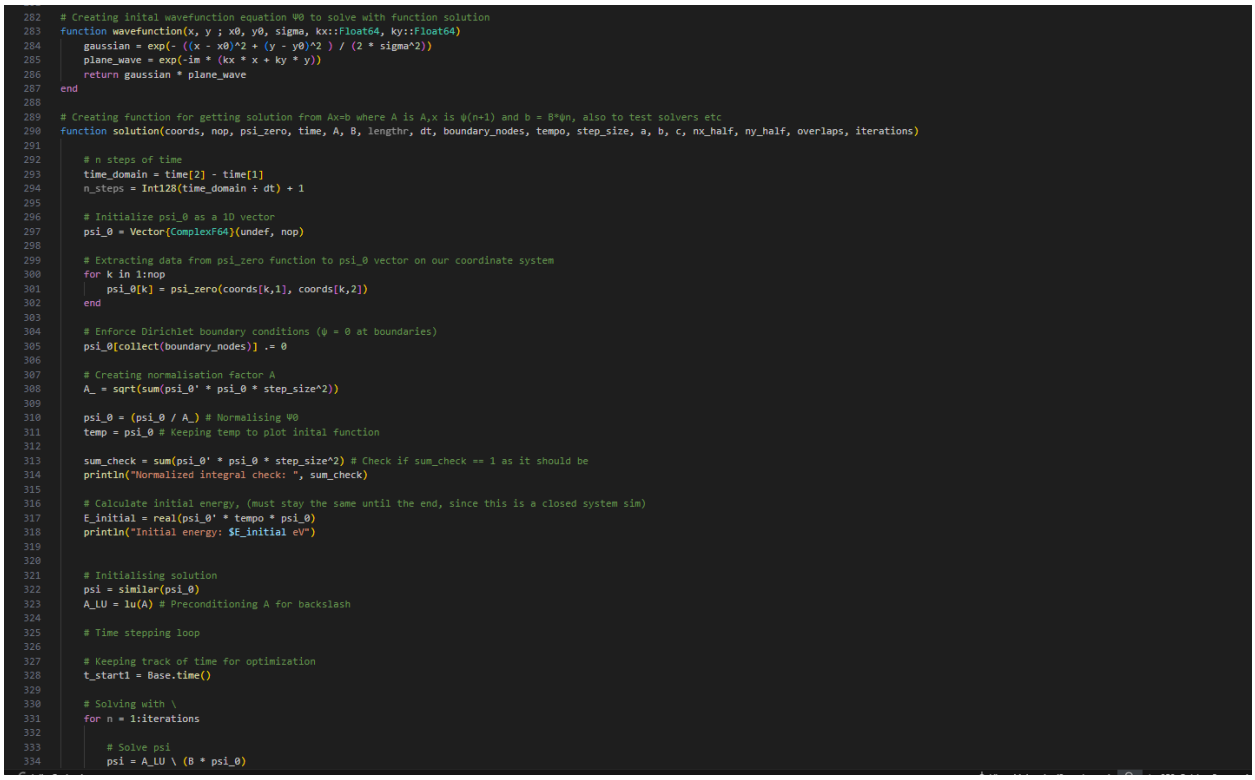
Solves the TDSE over a specified number of iterations or up to a certain time, yielding the final wavefunction and testing each solver.

- **Key Steps:**
  1. **Initialization:** The initial wavefunction vector $\vec{\psi}_0$ is populated using the psi_zero function evaluated at nodal coordinates. Dirichlet boundary conditions are enforced on $\vec{\psi}_0$.
  2. **Normalization:** $\vec{\psi}_0$ is normalized such that $\int |\Psi_0|^2 dxdy = 1$. The normalization constant A_ is calculated as sqrt(sum(psi_0' * psi_0 * step_size^2)). A check sum_check verifies this.
  3. **Initial Energy:** The initial energy $E_{\text{initial}} = \text{real}(\vec{\psi}_0^{\dagger} \cdot \text{tempo} \cdot \vec{\psi}_0)$ is calculated. tempo is the true Hamiltonian matrix.
  4. **Time Stepping and Solver Benchmarking:** The system $A\vec{\psi}^{n+1} = B\vec{\psi}^n$ is solved iteratively for iterations steps using four different approaches, and their execution times are recorded:

     a. Direct Solver (\): A_LU = lu(A) precomputes the LU factorization.
     b. BiCGSTAB Alone.
     c. Domain Decomposition with BiCGSTAB.

d. Domain Decomposition with Multigrid.

5. **Final Energy:** The energy is recalculated using the final wavefunction. For a closed system with a time-independent Hamiltonian, energy should be conserved.
6. **Performance Reporting:** Timings for each solver approach are printed



```
# Creating inital wavefunction equation Ψ0 to solve with function solution
function wavefunction(x, y ; x0, y0, sigma, kx::Float64, ky::Float64)
    gaussian = exp(- ((x - x0)^2 + (y - y0)^2 ) / (2 * sigma^2))
    plane_wave = exp(-im * (kx * x + ky * y))
    return gaussian * plane_wave
end

# Creating function for getting solution from Ax=b where A is A,x is ψ(n+1) and b = B*ψn, also to test solvers etc
function solution(coords, nop, psi_zero, time, A, B, lengthr, dt, boundary_nodes, tempo, step_size, a, b, c, nx_half, ny_half, overlaps, iterations)

    # n steps of time
    time_domain = time[2] - time[1]
    n_steps = Int128(time_domain ÷ dt) + 1

    # Initialize psi_0 as a 1D vector
    psi_0 = Vector{ComplexF64}(undef, nop)

    # Extracting data from psi_zero function to psi_0 vector on our coordinate system
    for k in 1:nop
        psi_0[k] = psi_zero(coords[k,1], coords[k,2])
    end

    # Enforce Dirichlet boundary conditions (ψ = 0 at boundaries)
    psi_0[collect(boundary_nodes)] .= 0

    # Creating normalisation factor A
    A_ = sqrt(sum(psi_0' * psi_0 * step_size^2))

    psi_0 = (psi_0 / A_) # Normalising Ψ0
    temp = psi_0 # Keeping temp to plot inital function

    sum_check = sum(psi_0' * psi_0 * step_size^2) # Check if sum_check == 1 as it should be
    println("Normalized integral check: ", sum_check)

    # Calculate initial energy, (must stay the same until the end, since this is a closed system sim)
    E_initial = real(psi_0' * tempo * psi_0)
    println("Initial energy: $E_initial eV")


    # Initialising solution
    psi = similar(psi_0)
    A_LU = lu(A) # Preconditioning A for backslash

    # Time stepping loop

    # Keeping track of time for optimization
    t_start1 = Base.time()

    # Solving with \
    for n = 1:iterations

        # Solve psi
        psi = A_LU \ (B * psi_0)
```

```
        psi = A_LU \ (B * psi_0)
        #println(sum(psi'*psi*step_size^2)) # Checking if integral stays the same
        # Asigning value to psi_0
        psi_0 = psi

    end
    temp1 = psi
    t_end1 = Base.time()

    # Reusing old solution to test my own solver
    psi_0 = temp

    println("Multigrid with domain decomposition method using $overlaps overlaps")
    t_start2 = Base.time()

    for n = 1:iterations

        # Solve psi
        psi = bicgstab_vic(A, B * psi_0, 1e-10, 150) # Bicgstab alone

        # Asigning value to psi_0
        psi_0 = psi

    end
    t_end2 = Base.time()

    # Reusing old solution to test solver
    psi_0 = temp

    t_start3 = Base.time()

    for n = 1:iterations

        # Solve psi
        psi = domain_decomposition(A, B * psi_0, a, b, c, nx_half, ny_half, overlaps, 1)

        # Asigning value to psi_0
        psi_0 = psi

    end
    t_end3 = Base.time()

    # Reusing old solution to test solver
    psi_0 = temp

    t_start4 = Base.time()

    for n = 1:iterations

        # Solve psi
        psi = domain_decomposition(A, B * psi_0, a, b, c, nx_half, ny_half, overlaps, 2)

```

```
        # Asigning value to psi_0
        psi_0 = psi

    end
    t_end4 = Base.time()

    # Final calculations

    final_E = real(psi_0' * tempo * psi_0)
    println("Energy: $final_E eV")

    time__ = t_end1 - t_start1
    time__ = round(time__, digits = 4)
    println("Time elapsed using backslash : $time__ seconds")

    time_2 = t_end2 - t_start2
    time_2 = round(time_2, digits = 4)
    println("Time elapsed using bicgstab solver alone: $time_2 seconds")

    time_3 = t_end3 - t_start3
    time_3 = round(time_3, digits = 4)
    println("Time elapsed using dd and bicgstab solver : $time_3 seconds")

    time4 = t_end4 - t_start4
    time4 = round(time4, digits = 4)
    println("Time elapsed using dd, multigrid and bicgstab solver : $time4 seconds")

    #println("Compare norms of psi solved with backslash $(norm(temp1)) vs multigrid $(norm(psi))")
    return psi, temp # return psi and the initial state 1 #
end

```

3.7. animated_solution Function

Generates an animation of the probability density $|\Psi(x, y, t)|^2$ over time using the solution function's logic but without recording the benchmarks. It calculates global_max ensure consistent z-axis scaling in plots, the user gives an input of the iterations wanted and the function captures a frame each no_afr.

```
# Creating animated solution over desired time
function animated_solution(coords, nop, psi_zero, time, A, B, lengthr, dt, boundary_nodes, tempo, step_size, no_fr, no_afr)

    println("Iterating with Crank-Nicolson...")
    time0 = Base.time()
    # Initialize psi_0 as a 1D vector with a solution at each point so "nop"
    psi_0 = Vector{ComplexF64}(undef, nop)

    # Extracting data from psi_zero function to psi_0 vector on our coordinate system
    for k in 1:nop
        psi_0[k] = psi_zero(coords[k,1], coords[k,2])
    end

    # Enforce Dirichlet boundary conditions (ψ = 0 at boundaries)
    psi_0[collect(boundary_nodes)] .= 0

    # Creating normalisation factor A
    A_ = sqrt(sum(psi_0' * psi_0 * step_size^2))

    # Normalising Ψ0
    psi_0 = (psi_0 / A_)

```

```
    # Creating range for animation
    xg = yg = range(-1, 1, step = step_size)

    # Calculate global maximum for consistent scaling in plotting
    global_max = maximum(abs2.(psi_0)) * 1.8

    # Preconditioning
    A_LU = lu(A)

    # Initialisng frames and solution matrices to plot
    frames = []
    Z = fill(NaN, lengthr, lengthr)

    # Time evolution loop with Crank Nicolson
    for n in 1:no_fr # Produce

        # Solving psi with , right now \
        psi = A_LU \ (B * psi_0)

        # Time variable for plotting
        t = (n - 1) * dt * 1000
        t = round(t, digits = 2)
        # Capture every no_afrth frame
        if n % no_afr == 0 || n == 1 #800
            for k in 1:nop
                i = i = round(Int, (coords[k,:][2] + 1) / step_size) + 1
                j = round(Int, (coords[k,:][1] + 1) / step_size) + 1
                Z[i,j] = abs2(psi[k]) # ||^2 value of psi
            end
            # Plotting variable
            p = plot(
                surface(xg, yg, Z, # Surface for 3d plot
                    colormap = :viridis, # Matlab colormap viridis
                    colorbar = true, # Colorbar at the right to see at values
                    title = "Electron wavefunction |ψ|² at t = $(round(t, digits=3)) femtoseconds",# Printing time as well
                    xlabel="x in nm", ylabel="y in nm", zlabel="|ψ(x, y)|²",
                    zlim = (0, global_max),
                    xlim = extrema(xg),# Extremes for x, y and the colorbar
                    ylim = extrema(yg),
                    clim = (0, global_max),
                    showscale = false,
                    camera = (30, 25, 1.0),
                    showlegend = false,
                    show_boundingbox = false,
                    overwrite_figure = false
                ),
                size = (800, 600),  # Resolution
                dpi = 300  # DPI
            )
            push!(frames, p)# Push p into frames for animation
        end
        psi_0 = psi  # Ψn = Ψ to continue iterative method
```

```
            end
            psi_0 = psi # Ψ0 = Ψ to continue iterative method
        end
        time1 = Base.time()
        time = time1 - time0
        time = round(time, digits = 2)

        println("Finished iterating $no_fr frames in $time seconds")

        lengthf = length(frames)
        # Saving animation into variable
        println("Creating animation...")
        time0 = Base.time()
        anim = @animate for (i, frame) in enumerate(frames)
            plot(frame)  # Plot the current frame
            println("Progress: $i/$lengthf")  # Show current/total
        end

        # Printing energy level
        Energy = real(psi_0' * tempo * psi_0)
        println("Energy of closed system is: $Energy eV")
        time1 = Base.time()
        time = time1 - time0
        time = round(time, digits = 2)
        println("Animation created in $time seconds")
        return anim
    end

```

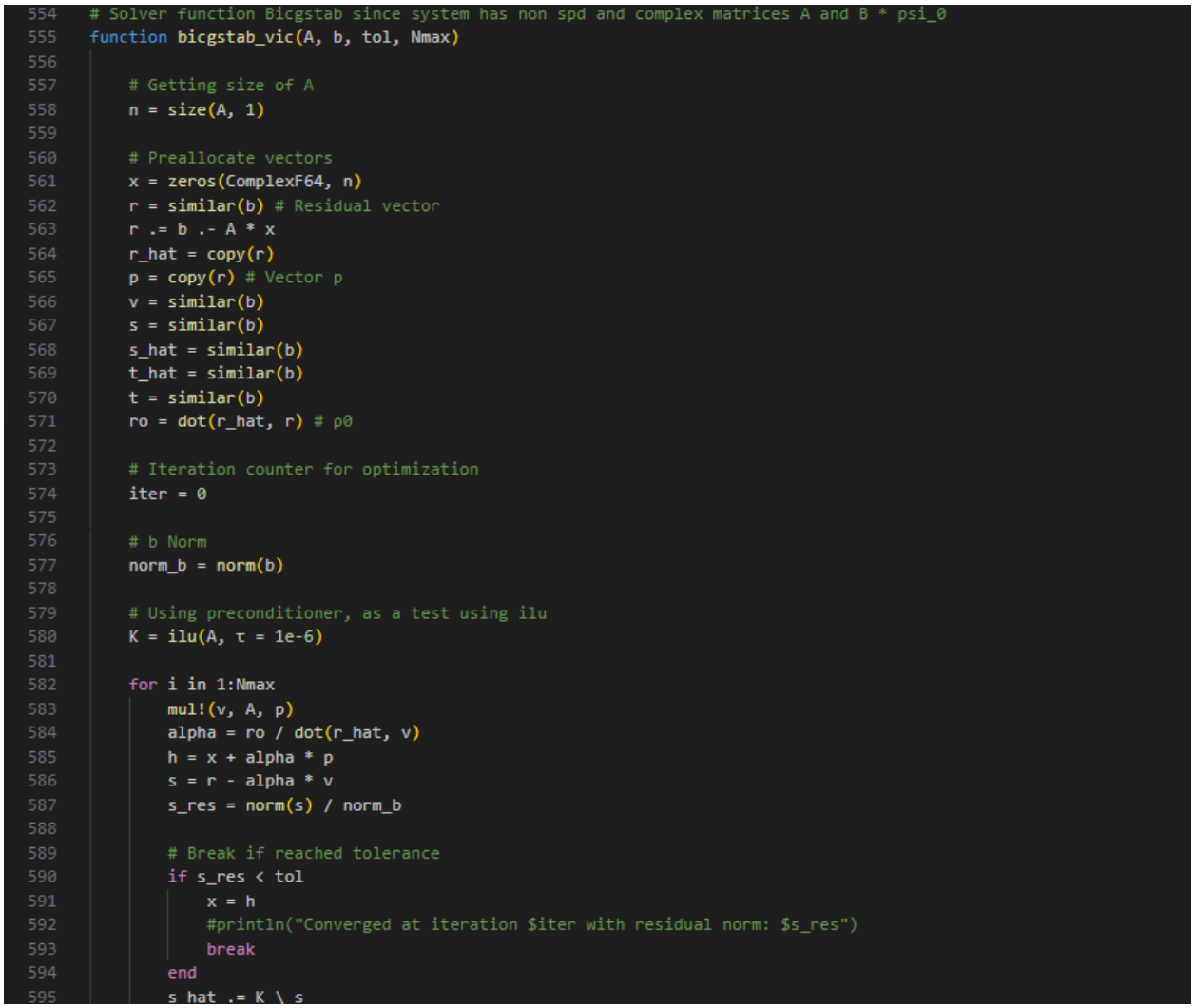
3.8. Helper Functions

- **gauss_quad_2D(funct, n):** Implements 2D Gaussian quadrature for numerical integration over a square domain $[-1, 1] \times [-1, 1]$. It supports $n = 2, 3, 4$ integration points in each direction. This is crucial for accurately computing the element matrices in fem_matrices.

```
# Creating 2 dimensional gauss_quadrature function to use to solve the integrals needed in FEM integrating from -1 to 1
function gauss_quad_2D(funct, n)
    if n == 2
        ksi = (-0.5773502692, 0.5773502692)
        eta = (-0.5773502692, 0.5773502692)
        w = ones(2)
    elseif n == 3
        ksi = (0, -0.7745966692, 0.7745966692)
        eta = (0, -0.7745966692, 0.7745966692)
        w = (0.8888888889, 0.5555555556, 0.5555555556)
    elseif n == 4
        ksi = (-0.8611363116, -0.3399810436, 0.3399810436, 0.8611363116)
        eta = (-0.8611363116, -0.3399810436, 0.3399810436, 0.8611363116)
        w = (0.3478548451, 0.6521451549, 0.6521451549, 0.3478548451)
    else
        error("Only n = 2, 3, 4 are supported")
    end
    result = 0.0
    for i = 1:n
        @inbounds for j = 1:n
            result += w[i] * w[j] * funct(ksi[i], eta[j])
        end
    end
    return result
end
```

- **bicgstab_vic(A, b, tol, Nmax):**

  Implements the BiConjugate Gradient Stabilized (BiCGSTAB) method, an iterative algorithm to solve $Ax = b$ on a non spd system. BiCGSTAB is a Krylov subspace method that iteratively refines a solution. It uses an Incomplete LU factorization ($K \approx A$) as a preconditioner (K^{-1} applied via K \ vector) to accelerate convergence. Each iteration involves computing step lengths (α, ω) and search directions (p) to minimize the residual $r = b - Ax$.



```julia
# Solver function Bicgstab since system has non spd and complex matrices A and B * psi_0
function bicgstab_vic(A, b, tol, Nmax)

    # Getting size of A
    n = size(A, 1)

    # Preallocate vectors
    x = zeros(ComplexF64, n)
    r = similar(b) # Residual vector
    r .= b .- A * x
    r_hat = copy(r)
    p = copy(r) # Vector p
    v = similar(b)
    s = similar(b)
    s_hat = similar(b)
    t_hat = similar(b)
    t = similar(b)
    ro = dot(r_hat, r) # ρ0

    # Iteration counter for optimization
    iter = 0

    # b Norm
    norm_b = norm(b)

    # Using preconditioner, as a test using ilu
    K = ilu(A, τ = 1e-6)

    for i in 1:Nmax
        mul!(v, A, p)
        alpha = ro / dot(r_hat, v)
        h = x + alpha * p
        s = r - alpha * v
        s_res = norm(s) / norm_b

        # Break if reached tolerance
        if s_res < tol
            x = h
            #println("Converged at iteration $iter with residual norm: $s_res")
            break
        end
        s_hat .= K \ s
```

```
596             t = A * s_hat
597             t_hat .= K \ t
598             omega = dot(t_hat, s_hat) / dot(t_hat, t_hat)
599             x = h + omega * s_hat
600             r = s - omega * t
601 
602             rel_res = norm(r) / norm_b
603 
604             if rel_res < tol
605                 #println("Converged at iteration $iter with residual norm: $rel_res")
606                 break
607             end
608             temp = ro
609             ro = dot(r_hat, r)
610             beta = (ro / temp) * (alpha / omega)
611             p = r + beta * (p - omega * v)
612             iter += 1
613             if i == Nmax
614                 #println("Exiting with max iterations..")
615             end
616         end
617         return x
618     end
```

- **domain_decomposition(A, b, s1, s2, s3, nx_half, ny_half, overlap, flag):**

  Implements an Additive Schwarz domain decomposition method to iteratively solve $Ax = b$.

  Script Implementation Details:

  1. **Setup:**
     - Validates overlap.
     - Initializes global solution x to zero.
     - Defines inner1, inner3 (core non-overlapping parts of subdomains 1 and 3) and overlap matrices (over1, over21, etc.) based on s1, s2, s3 and overlap.
     - Constructs global node index vectors (inds1, inds2, inds3) for the three extended (overlapping) subdomains.
     - Extracts subdomain matrices A_1, A_2, A_3 from the global A.
     - Creates mapping index vectors d and c to extract solutions from the core regions of extended subdomains 2 and 3, respectively.

```
621 function domain_decomposition(A, b, s1, s2, s3, nx_half, ny_half, overlap, flag) # Matrix A and sub matrices from grid A, b,
622
623     # error handling
624     if overlap > nx_half - 1 || overlap > ny_half - 1 # n_half = nx / 2 + 1, thats why -1 is there
625         println("Overlap needs to  less than or equal to n / 2 ...")
626         return 0
627     end
628
629     # Getting size of A
630     n = size(A, 1)
631
632     # Preallocate solution and residual vectors
633     x = zeros(ComplexF64, n, 1)
634     r = similar(b)
635
636     # s1, s2, s3 modification to get inner and boundary nodes, inner2 = s2
637     inner1 = s1[1: end - 1, :] # Last line is the boundary line same as first line of s2
638     inner3 = s3[:, 2:end] # First collumn is the boundary collumn aligning with s2
639
640     # Overlap matrices
641     over1 = s2[1:overlap, :]
642     over21 = inner1[end - overlap + 1:end, :]
643     over23 = inner3[:, 1:overlap]
644     over3 = s2[:, end - overlap + 1:end]
645
646     # Indices for submatrices
647     inds1 = vec(transpose(vcat(inner1, over1)))
```

```
646     # Indices for submatrices
647     inds1 = vec(transpose(vcat(inner1, over1)))
648     inds2 = vec(hcat(transpose(vcat(over21, s2)), over23))
649     inds2 = sort(inds2)
650     inds3 = vec(transpose(hcat(over3, inner3)))
651
652     # Sumbatrices of A
653     A1 = A[inds1, inds1]
654     A2 = A[inds2, inds2]
655     A3 = A[inds3, inds3]
656     # Resulting matrices will be of size nop x nop where nop is the number of nodes within their subdomain, they will be square
657
658     # Creating index Matrix that gets wanted solution from x2 after solving, which is where s2 lies, meaning, we dont want the soluti
659     # that is on over21 and over23 but on top of s2
660     d = zeros(Int, length(vec(s2)), 1)
661     index = length(vec(over21)) + 1 # index starts from where over21 ends, also used to give value to d
662     indexx = 1
663     for k = 1:size(s2)[1] # Row loop
664         for i = 1: size(s2)[2]
665             d[indexx] = index
666             index += 1
667             indexx +=1
668         end
669         index +=  size(over23)[2] # skip over23
670     end
671
672     # Same for s3 to not include the first collumn of the solution
673     c = zeros(Int, length(vec(inner3)), 1)
674     index = overlap + 1 # skips first collumn
675     indexx = 1
676     for k = 1:size(inner3)[1] # Row loop
677         for i = 1: size(inner3)[2]
678             c[indexx] = index
679             index += 1
680             indexx +=1
681         end
682         index += overlap # skips first collumn
683     end
684
685     # Max iterations and tolerance
686     Nmax = 200;
687     tolDD = 10e-10
688     tolMG = 10e-4
689     tolBic = 10e-5
690     it = 20
691
```

2. **Solver Branching (flag):**
   - **If flag == 1 (BiCGSTAB for subdomains):**
     - In each DD iteration, solves $A_k \delta\tilde{x}_k = r_k^{(m)}$ using bicgstab_vic for

each of the three subdomains.

```
692     if flag == 1 # Solving with bicgstab
693         for i = 1:Nmax
694             # Residual
695             r = b - A * x;
696 
697             # Convergence criterion and fail print
```

```
697             # Convergence criterion and fail print
698             if(norm(r)/norm(b)<tolDD)
699                 #println("Converged in $i iterations with residual ", norm(r)/norm(b))
700                 break
701             end
702             if i == Nmax
703                 println("Reached maximum iterations without convergence")
704             end
705 
706             # Solve all systems with residual in selected indices
707 
708             # Solve first system
709 
710             x1 = bicgstab_vic(A1, r[inds1], tolBic, it) # multigrid_vic(nx_1, ny_1, overlap, A1, r[inds1], 3, 2, tol, 1)
711             x1 = x1[1:(length(inner1))] # Stays the same since indexing is the same
712 
713             # Solve second system
714             x2 = bicgstab_vic(A2, r[inds2], tolBic, it) # multigrid_vic(nx_2, ny_2, overlap, A2, r[inds2], 3, 2, tol, 2)
715             x2 = x2[d]
716 
717             # Solve the third system
718             x3 = bicgstab_vic(A3, r[inds3], tolBic, it) # multigrid_vic(nx_3, ny_3, overlap, A3, r[inds3], 3, 2, tol, 1)
719             x3 = x3[c]
720 
721             # Update solution
722             x[sort!(vec(inner1))] += x1
723             x[sort!(vec(s2))] += x2
724             x[sort!(vec(inner3))] += x3
725     end
```

- **If flag == 2 (Multigrid for subdomains):**
  - **Crucially, calls multigrid_vic_hierarchy *once before the DD iteration loop*** for each subdomain (A1, A2, A3) to precompute their respective multigrid hierarchies (A1_, R1, P1, levels1, etc.). The time for this setup is printed.

```
726     elseif flag == 2 # Using multigrid and bicgstab as a smoother
727
728         # Parameters
729         n1 = 6
730         n2 = 4
731
732         # Getting sizes for multigrid
733
734         nx_1 = size(inner1)[2]
735         ny_1 = size(inner1)[1] + overlap
736
737         nx_2 = size(s2)[1] + overlap
738         ny_2 = nx_2
739
740         nx_3 = size(inner3)[2] + overlap
741         ny_3 = size(inner3)[1]
742
743         time_start = Base.time()
744         # Preallocating matrices for multigrid
745         A1_, R1, P1, levels1 = multigrid_vic_hierarchy(nx_1, ny_1, overlap, A1)
746         A2_, R2, P2, levels2 = multigrid_vic_hierarchy(nx_2, ny_2, overlap, A2, 2)
747         A3_, R3, P3, levels3 = multigrid_vic_hierarchy(nx_3, ny_3, overlap, A3)
748
749         time_end = Base.time()
```

- In each DD iteration, solves $A_k \delta\tilde{x}_k = r_k^{(m)}$ using multigrid_vic for each subdomain, passing the precomputed hierarchies.

```
        time = time_end - time_start
        time = round(time, digits = 4)
        println("Time elapsed creating matrices for multigrid before loop:", time)

        for i = 1:Nmax
            # Residual
            r = b - A * x;

            # Convergence criterion and fail print
            if(norm(r)/norm(b)<tolDD)
                #println("Converged in $i iterations with residual ", norm(r)/norm(b))
                break
            end
            if i == Nmax
                println("Reached maximum iterations without convergence")
            end

            # Solve all systems with residual in selected indices

            # Solve first system

            x1 = multigrid_vic(A1_, R1, P1, levels1, r[inds1], n1, n2, tolMG)
            x1 = x1[1:(length(inner1))] # Stays the same since indexing is the same

            # Solve second system
            x2 = multigrid_vic(A2_, R2, P2, levels2, r[inds2], n1, n2, tolMG)
            x2 = x2[d]

            # Solve the third system
            x3 = multigrid_vic(A3_, R3, P3, levels3, r[inds3], n1, n2, tolMG)
            x3 = x3[c]

            # Update solution
            x[sort!(vec(inner1))] += x1
            x[sort!(vec(s2))] += x2
            x[sort!(vec(inner3))] += x3
        end
    end
    return x
end
```

3. DD Iteration:
   a. Calculates global residual $r = b - Ax$.
   b. Checks for convergence (norm(r)/norm(b) < tolDD).
   c. Solves for local corrections x1, x2, x3 on subdomains using the method selected by flag.
   d. Extracts relevant parts of x1, x2, x3 (using d and c for x2, x3).
   e. Updates global solution x additively.

The L-shaped domain Ω is divided into three overlapping subdomains Ω_k. The global matrix A is restricted to each extended (overlapping) subdomain as $A_k = A[\text{inds}_k, \text{inds}_k]$.

The iterative process is:

1. Initialize solution $x^{(0)}$.
2. For iteration $m = 0, 1, \ldots$

a. Global residual: $r^{(m)} = b - Ax^{(m)}$

b. For each subdomain k, solve a local correction problem on the extended subdomain using the restricted residual $r_k^{(m)} = r^{(m)}[\text{inds}_k]$: $A_k \delta\tilde{x}_k = r_k^{(m)}$

(Solved via bicgstab_vic in the code). $\delta\tilde{x}_k$ is the correction on the extended subdomain.

c. Map relevant parts of $\delta\tilde{x}_k$ to a global correction that applies to the core (non-overlapping) part of subdomain k. (The code uses index vectors d and c for this mapping for subdomains 2 and 3).

d. Update global solution additively: `x(m+1)=x(m)+Σk(mapped core correction for δx~k).

3. Convergence is checked by $\|r^{(m)}\|/\|b\| < \text{tol}$.

- **Parameters Affecting Domain Decomposition Convergence and Performance:**
  - overlap: The number of shared node layers between subdomains. Generally, a larger overlap improves the convergence rate of the Schwarz iterations (as more information is exchanged per iteration) but also increases the size and cost of solving each subdomain problem. An optimal overlap balances these factors. Too small an overlap can lead to slow convergence or divergence.
  - Tolerance (tol) and Max Iterations (Nmax): These control the accuracy of the final DD solution and the maximum computational effort for the outer Schwarz loop.
  - Problem Size and Condition Number: Larger, more ill-conditioned global systems $Ax = b$ will generally be harder for any iterative method, including DD, to solve. The properties of the subdomain matrices A_k also play a role.

- **multigrid_vic_hierarchy(nox, noy, overlap, A1, flag_grid_type = 1):**

  Precomputes and returns the components of a geometric multigrid hierarchy for a given fine-grid subdomain operator A1. Builds restriction and prolongation matrices $R^{(l-1)}$, $P^{(l-1)} = 4(R^{(l-1)})^T$ for each level using build_Restriction_matrix. Builds coarse grid operators using Galerkin projection for $l > 1$, $A^{(l)} = R^{(l-1)} A^{(l-1)} P^{(l-1)}$.

```
function multigrid_vic_hierarchy(nox, noy, overlap, A1, flag = 1)

    #  First I will be creating the restriction and prolongation matrices by using the weighted average of nine fine grid values to create
    # for non_f = fine grid number of nodes in a direction, to find the coarse nodes non_c = floor((non_f - 1) / 2) + 1 so that if the interval number is odd it becomes even
    # example 5x x 6y nodes -> 4x x 5y intervals, non_cx = 3 and non_cy = 3 also, containing the information of all the previous finer nodes


    # Getting number of nodes for each coarser level

    levels = 1
    nx = zeros(Int, 21, 1) # A size of 21 is sufficient to have as many as 2 million nodes, set number less expensive than calculating the levels
    ny = similar(nx)
```

```
    ny = similar(nx)

    nx[1] = nox # Fine grid values at level 1
    ny[1] = noy

    #println("x: $(nx[1]), y: $(ny[1]) at level: $levels")

    for i = 2: 21
        nx[i] = floor((nx[i - 1] - 1) / 2) + 1
        ny[i] = floor((ny[i - 1] - 1) / 2) + 1
        levels += 1
        #println("x: $(nx[i]), y: $(ny[i]) at level: $levels")
        if nx[i] <= 3 || ny[i] <= 3
            break
        end
    end
    levelss = 1
    if flag == 2
        n_inner = zeros(Int, 21, 1)
        n_inner[1] = nox - overlap
        #println("Inner nodes: $(n_inner[1]) at level: $levelss")
        for i = 2: 21
            n_inner[i] = floor((n_inner[i - 1] - 1) / 2) + 1
            levelss += 1
            #println("Inner nodes: $(n_inner[i]) at level: $levels")
            if nx[i] <= 3
                break
            elseif n_inner[i] <= 3
                for j = i + 1: levels
                    n_inner[j] = n_inner[i]
                    levelss += 1
                    #println("Inner nodes: $(n_inner[j]) at level: $levelss")
                end
                break
            end
        end
    end
    # Create restriction and prolongation matrices which are for levels 2-total number of levels since the first level doesnt need them

    R = Vector{SparseMatrixCSC{Float64, Int}}(undef, levels - 1) # Vectors of sparse matrices
    P = Vector{SparseMatrixCSC{Float64, Int}}(undef, levels - 1)

    for i = 2:levels
        if flag == 1 # For A1 and A3 which are rectangles
            R[i - 1] = build_Restriction_matrix(flag, nx[i - 1], nx[i], ny[i - 1], ny[i])
        else # If its A2 and it has an L shape
            R[i - 1] = build_Restriction_matrix(flag, nx[i - 1], nx[i], ny[i - 1], ny[i], n_inner[i - 1], n_inner[i])
        end
        # Create prolongation matrix from the restriction
        P[i - 1] = 4 * R[i - 1]'
    end
    # Inniate A
```

```
    # Inniate A

    A = Vector{SparseMatrixCSC{ComplexF64, Int}}(undef, levels) # Vector of sparse matrices

    # Build A's top level (finest)
    A[1] = A1

    # Use Galerkin method to build A on each level
    for i = 2: levels
        A[i] = R[i - 1] * A[i - 1] * P[i - 1]
    end

    return A, R, P, levels
end
```

- **multigrid_vic(A_hierarchy, R_hierarchy, P_hierarchy, levels, B_fine, n1, n2,**

tol):

Performs a V-cycle multigrid solve using a precomputed hierarchy.

Script Implementation Details:

1. Initializes solution vectors $x^{(l)}$ and RHS vectors $b^{(l)}$ for all levels. Sets $b[1] = B_fine$.
2. Iterates for Nmax = 17 V-cycles or until convergence.
3. V-Cycle Implementation:
   a. Down-stroke: For levels
4. Convergence is checked on the finest level residual: norm(b[1] - A_hierarchy[1] * x[1]) < norm_b * tol.

```
# Creating multigrid function with V cycles for my 2D grid, as it the most efficient, to use as preconditioner along with domain decomposition
#so this multigrid will be tailored for rectangle grids (not L shape since we use decomposition)
function multigrid_vic(A, R, P, levels, B, n1, n2, tol)

    # Parameters
    Nmax = 17

    # Iniate b and x vectors
    b = Vector{Vector{ComplexF64}}(undef, levels)
    x = similar(b)

    for i = 1: levels
        b[i] = zeros(ComplexF64, size(A[i])[1])
        x[i] = similar(b[i])
    end
    # Build the RHS's top level
    b[1] = B

    # Norm of RHS
    norm_b = norm(b[1])

    # Iterate over all levels
    for it = 1: Nmax
        for i = 2: levels - 1
            x[i] = bicgstab_vic(A[i], b[i], 10e-10, n1)
            b[i + 1] = R[i] * (b[i] - A[i] * x[i])
            x[i + 1][:].= 0
        end
        x[end] = A[end] \ b[end]

        for i = levels - 1 : -1: 1
            x[i] = x[i] + P[i] * x[i+1]
            x[i] = bicgstab_vic(A[i], b[i], 10e-10, n2)
        end

        nrm = norm(b[1] - A[1] * x[1])
        if nrm < norm_b * tol
            break;
```

```
        end

        nrm = norm(b[1] - A[1] * x[1])
        if nrm < norm_b * tol
            break;
        end
        if it == Nmax
            println("Exiting with max iterations on multigrid")
            break;
        end
    end
    return x[1]
end

# Map function to get active node mappings for L-shape and not get the top right corner.
function map(flag, nx::Int, n_inner::Int = 0, ny::Int = nx)

    map_index = zeros(Int, ny, nx) # Saves the indicies of the active nodes
    map_coords = Tuple{Int, Int}[] # Stores the coordinates of active node index

    n_act = 0 # Active number of nodes

    # Loop to check if the node is active and get the indices and coordinate
    for y = 1:ny
        for x = 1:nx
            if ((x <= n_inner && y <= ny) || (y <= n_inner && x <= nx)) && flag == 2
                n_act += 1
                map_index[y, x] = n_act
                push!(map_coords, (y, x))
            elseif flag == 1
                n_act += 1
                map_index[y, x] = n_act
                push!(map_coords, (y, x))
            end
        end
    end
    return n_act, map_index, map_coords
end

# Function to build the L-shaped Restriction Matrix using fine and coarse grid bounding box and inner box parameters
function build_Restriction_matrix(flag, nfx_bb::Int, ncx_bb::Int, nfy_bb::Int = nfx_bb, ncy_bb::Int = ncx_bb, nf_inner::Int = 0, nc_inner::Int = 0)
    if flag == 2
        nf_act, f_map_index, _ = map(flag, nfx_bb, nf_inner) # Getting the fine active nodes as well as the fine map indices
        nc_act, _, c_map_coords = map(flag, ncx_bb, nc_inner) # Getting the coarse active nodes as well as the coordinates for the coarse map
    else
        nf_act, f_map_index, _ = map(flag, nfx_bb, 0, nfy_bb) # Getting the fine active nodes as well as the fine map indices
        nc_act, _, c_map_coords = map(flag, ncx_bb, 0, ncy_bb) # Getting the coarse active nodes as well as the coordinates for the coarse map
    end
    if nc_act == 0 || nf_act == 0
        return spzeros(Float64, nc_act, nf_act)
    end

    I = Int[] # Row indices (active coarse node L-indices)
```

```
    end

    I = Int[] # Row indices (active coarse node L-indices)
    J = Int[] # Collumn indices (active fine node L-indices)
    V = Float64[] # Stencil weights

    stencil = 1/16 * [1 2 1; 2 4 2; 1 2 1] # 9 node stencil, diagonal 1/16, horizontal and vertical 1/8 and central 1/4 adding up to 1

    for  kcL = 1: nc_act # Iterate over each active coarse node
        yc_bb, xc_bb = c_map_coords[kcL] # 2D bounding box coords of this coarse node on y, x

        # Getting central fine grid point correspoing to where the coarse node is
        yf_ctr = 2 * yc_bb - 1 # Fine grid center y
        xf_ctr = 2 * xc_bb - 1 # Fine grid center x

        for s_y = 1:3 # Stencil in the y-dim (1,2,3)
            for s_x = 1:3 # Stencil in x-dim

                weight = stencil[s_y, s_x]
                if weight == 0.0 continue end # If there is no node continue to the next node

                off_y = s_y - 2 # Stencil offset (-1,0,1)
                off_x = s_x - 2 # Stencil offset (-1,0,1)

                tfy = yf_ctr + off_y # Target fine y
                tfx = xf_ctr + off_x # Target fine x

                if (1 <= tfy <= nfy_bb) && (1 <= tfx <= nfx_bb)
                    kfL = f_map_index[tfy, tfx]
                    if kfL>0
                        push!(I, kcL)
                        push!(J, kfL)
                        push!(V, weight)
                    end
                end
            end
        end
    end

    if isempty(I)
        return spzeros(Float64, nc_act, nf_act)
    end

    R = sparse(I, J, V, nc_act, nf_act)
    #=row_sums = sum(R, dims=2)
    for i in 1:nc_act
        if row_sums[i] != 0
            R[i, :] = R[i, :] ./ row_sums[i]
        end
    end=#
    return R
end
end # Module end
```

Mathematical Idea (Geometric Multigrid V-Cycle):

1. **Hierarchy of Grids:** A sequence of coarser grids is defined, starting from the fine grid (dimensions nox, noy). Node counts are approximately halved in each dimension for coarser levels until a minimum size is reached. For L-shaped subdomains (flag_grid_type=2), n_inner helps manage the geometry during coarsening.
2. **Grid Operators ($A^{(l)}$):** The operator $A^{(0)} = A1$ is given on the finest level. For coarser levels $l > 0$, the operator is formed using the Galerkin approach:
   $A^{(l)} = R^{(l-1)} A^{(l-1)} P^{(l-1)}$

   where $R^{(l-1)}$ is the restriction operator from level $l-1$ to l, and $P^{(l-1)}$ is the prolongation (interpolation) operator from level l to $l-1$.
3. **Restriction (R):** Transfers a vector (e.g., residual) from a finer grid to a

coarser grid. The build_Restriction_matrix function constructs R based on weighted averaging (using a 9-point stencil for interior points).

4. **Prolongation (P):** Interpolates a vector (e.g., correction) from a coarser grid to a finer grid. In this code, $P = 4R^T$.
5. **Smoothing:** On each grid level l (except the coarsest), a "smoother" is applied to reduce high-frequency error components of the current approximate solution $x^{(l)}$ to $A^{(l)}x^{(l)} = b^{(l)}$. This implementation uses bicgstab_vic for n1 iterations (pre-smoothing) before going to a coarser grid, and for n2 iterations (post-smoothing) after returning from a coarser grid. The reason bicgstab_vic is used and not a smoother like damped Jacobi, is because A is not necessarily diagonally dominant as the smoother requires, as such not guaranteeing convergence.
6. V-Cycle Algorithm (Recursive View for solving $A^{(l)}x^{(l)} = b^{(l)}$):
   a. Coarsest Level: If on the coarsest grid, solve $A^{(\text{coarsest})}x^{(\text{coarsest})} = b^{(\text{coarsest})}$ directly (e.g., A[end] \ b[end]).
   b. Pre-Smoothing (Down-stroke): Perform n1 smoothing steps (e.g., bicgstab_vic) on $A^{(l)}x^{(l)} = b^{(l)}$ to get an improved $x^{(l)}$.
   c. Compute Residual: $d^{(l)} = b^{(l)} - A^{(l)}x^{(l)}$.
   d. Restrict Residual: Transfer residual to coarser grid: $d^{(l+1)} = R^{(l)}d^{(l)}$.
   e. Recursive Solve: Solve the coarse grid correction equation $A^{(l+1)}e^{(l+1)} = d^{(l+1)}$ by performing a V-cycle starting from level $l + 1$. Set initial guess for $e^{(l+1)}$ to zero.
   f. Prolongate Correction: Interpolate coarse grid correction $e^{(l+1)}$ back to fine grid: $e^{(l)} = P^{(l)}e^{(l+1)}$.
   g. Correct Solution: Update solution: $x^{(l)} = x^{(l)} + e^{(l)}$.
   h. Post-Smoothing (Up-stroke): Perform n2 smoothing steps on $A^{(l)}x^{(l)} = b^{(l)}$.

3.9. Solver Benchmarking and Discussion

The solution function in the script systematically benchmarks four different approaches to solve the linear system $A\vec{\psi}^{n+1} = B\vec{\psi}^{n}$ arising at each time step of the Crank-Nicolson scheme.

Observed Benchmark Ranking:

- **1.** Direct Solver **(\):**

For sparse matrices that are not excessively large (i.e., the number of unknowns nop is within a certain range), Julia's backslash operator (\) is highly optimized.

- **2. domain_decomposition with multigrid_vic (flag=2):**

  multigrid_vic method, when applied as a solver for the three smaller subdomain problems ($A_k \delta \tilde{x}_k = r_k^{(m)}$), is performing very effectively. Multigrid methods, when well-tuned, can achieve convergence rates that are nearly independent of the problem size (or depend very weakly), aiming for $O(N)$ complexity for a problem with N unknowns however the current implementation needs work.

- **3. domain_decomposition with bicgstab_vic (flag=1):**

  Subdomain Solver Efficiency: bicgstab_vic (with its ILU preconditioner) is less efficient at solving the subdomain problems than multigrid_vic if there are many elements, but for less elements due to bicgstab, this is very fast.

- **4. bicgstab_vic (alone):**

  bicgstab_vic applied directly to the global matrix A is the fastest for less elements as expected and is close to backslash.

4. main.jl

Simulates the time-evolution of a 2D quantum mechanical wavefunction by solving the time-dependent Schrödinger equation. It sets up the physical problem by defining an initial wave packet, a potential energy landscape, and the simulation domain. The script then utilizes the Finite Element Method (FEM), numerical solvers and domain decomposition techniques, to compute the wavefunction at a later time. Finally, it either visualizes the initial and final probability densities of the wavefunction or animates the solution using a chosen number of frames as well as a frame interval.

main.jl > ...

```julia
using Plots
using SparseArrays
#plotlyjs() # Enable PlotlyJS backend for interactivity
include("functions.jl")
import .Functions

# Victor Emmanuel Meimaris 23/3/25: Created main.jl for project in advanced scientific calc, where I will be solving the time evolving shrodinger equation
# in a 2D L shaped domain, creating my own solves and combining them (bicgstab, domain decomposition and multigrid).

# Input variables
domain = (-1, 1) # In nanometers
time = (0, 1)
domain_min = domain[1]
max_length = 0.025
overlaps = 20 # For domain decomposition
iterations = 1 #3000 #crank nicolson frames

# Wavefunction parameters
sigma = 0.15
y0 =  - 0.5
x0 = - 0.5
ky =  0.0 # + goes to the negative direction
kx = -20.0 # - 5.0

# Create mesh and extract parameters
mesh = Functions.grid(domain, max_length)
coords = mesh[1]
l2g = mesh[2]
noe = mesh[3]
nop = mesh[4]
boundary_nodes = mesh[5]
step_size = mesh[6]
lengthr = mesh[7]

dt = 1e-6 # Time step dt<(dx)^2 for optimal results for ex step_size^2 / 4


# Domain decomposition matrices
a = mesh[8]
b = mesh[9]
c = mesh[10]
nx_half = mesh[11]
ny_half = mesh[12]

# Potential function
V_flag = 3 # Potential V flag, 1 = box, 2 = box with circle barrier, 3 = box with circle well
V0 = 10 # Only needed for flags>1 is in eV!!
x_0 = 0.0 # Circle parameters
y_0 = - 0.5
r_0 = 0.5
V_potential_func = Functions.V_function(V_flag, V0, x_0, y_0, r_0)

# Get matrices
matrices = Functions.fem_matrices(V_potential_func, dt, coords, l2g, noe, boundary_nodes, step_size, lengthr)

```

```
println("Number of elements: ", noe)
println("Time step: $dt picoseconds")

# Create wavefunction
psi_0(x, y) = Functions.wavefunction(x, y; x0, y0, sigma, kx, ky)

# Currently commenting out lines i dont need to test my script!

# Save as mp4
# n steps of time
time_domain = time[2] - time[1]
n_steps = Int128(time_domain ÷ dt) + 1

no_frames = 7200
frame_inter = 80

anim = Functions.animated_solution(coords, nop, psi_0, time, matrices..., no_frames, frame_inter)# or n_steps
mp4(anim, "Animations/Electron/well_10ev.mp4", fps=15)
println("Done")


#= Create two plots for testing
psi = Functions.solution(coords, nop, psi_0, time, matrices..., a, b, c, nx_half, ny_half, overlaps, iterations)
#
psi_final = abs2.(psi[1])
psi_initial = abs2.(psi[2])

# Define grid ranges and Z
xg = -1:step_size:1
yg = -1:step_size:1
Z = fill(NaN, lengthr, lengthr) # Fill with NaN so that no space is occupied
F = fill(NaN, lengthr, lengthr)
# Map solution vector to grid
for k in 1:nop # Length of solution vector
    x, y = coords[k, 1], coords[k, 2]
    i = round(Int, (x - domain_min) / step_size) + 1  # +1 because Julia is 1-based
    j = round(Int, (y - domain_min) / step_size) + 1
    Z[i, j] = psi_final[k]
    F[i, j] = psi_initial[k]
end

# Create plots
p2 = surface(xg, yg, F,
        title="Initial Wavefunction |ψ0|^2",
        xlabel="y", ylabel="x", zlabel="|ψ0|^2",
        camera=(30, 40),
        colorbar=true,
        colorscale="Viridis",
        showscale=true)
display(p2)
p1 = surface(xg, yg, Z,
        title="Final Wavefunction |ψ0|^2",
        xlabel="y", ylabel="x", zlabel="|ψf|^2",
        camera=(30, 40),
```

Script Implementation Details:

- **Parameter Initialization:**

  Domain & Time: Defines the spatial domain from (-1, 1) nm and the simulation time interval.

  Numerical Parameters: Sets the maximum element length (max_length) for the mesh, the overlap size for domain decomposition, and the number of iterations for the solver.

  Wavefunction: Specifies the parameters for the initial Gaussian wave packet, including its center (x0, y0), width (sigma), and momentum (kx, ky).

  Potential: Configures a potential function V, which can be selected via V_flag (e.g., a box with a circular well).

- **Mesh & FEM Setup:**

  Grid Generation: Calls Functions.grid to create a 2D mesh over the domain, extracting coordinates, node counts, boundary information, and step sizes.

  Matrix Assembly: Calls Functions.fem_matrices to assemble the necessary system matrices (e.g., mass and stiffness matrices for the Crank-Nicolson method) based on the mesh and the defined potential function.

- **Wavefunction & Time Evolution:**

  Initial State: Creates the initial wavefunction psi_0 at time t=0 using the Functions.wavefunction helper.

  Solver: The primary computation is performed by Functions.solution, which takes the assembled matrices, initial wavefunction, and domain decomposition parameters to solve for the final state of the system.

  Animation (Optional): Includes a call to Functions.animated_solution to generate an .mp4 video of the wavefunction's evolution over time.

- **Visualization:**

  Data Mapping: The initial and final 1D solution vectors (psi_initial, psi_final) are mapped from the node list onto a 2D grid for plotting.

  Plotting: Uses the Plots.jl library to generate and display 3D surface plots of the initial and final probability densities (|ψ|^2).

5. Influence of Parameters on Simulation Output

The behavior and accuracy of the simulation are highly dependent on several key parameters:

- **Mesh Parameters (domain, max_length):**
  - domain: Defines the physical extent of the simulation area (e.g., -1 nm to 1 nm).
  - max_length: Controls the element size and thus the mesh density. A smaller max_length results in a finer mesh, leading to higher spatial accuracy but significantly increases computational cost (more nodes and elements, larger and denser matrices, longer solution times). The choice of max_length should be small enough to resolve the shortest wavelength components of the wavefunction and the features of the potential.
- **Time Step (dt):**

  Affects the temporal accuracy and stability of the simulation. The Crank-Nicolson scheme is unconditionally stable in theory for this problem,

but a very large Δt will still lead to poor accuracy, failing to capture the dynamics correctly. A smaller Δt provides better temporal resolution but increases the total number of steps required to simulate a given physical time, thus increasing overall computation time.

- **Initial Wavefunction Parameters ($x_0, y_0, \sigma, k_x, k_y$ in wavefunction):**
  - x_0, y_0: The initial (mean) position of the wavepacket.
  - σ: The initial spatial spread (standard deviation) of the wavepacket. A smaller σ means the particle is more localized initially, which, by the uncertainty principle, implies a wider spread in momentum and thus faster spreading of the packet over time.
  - k_x, k_y: Components of the initial wavevector, determining the initial average momentum $\mathbf{p} = \hbar\mathbf{k}$. These dictate the initial direction and group velocity of the wavepacket. Higher magnitudes of k mean higher initial kinetic energy.
- **Potential Parameters (V_flag, V_0, x_0, y_0, r in V_function):**
  - V_flag: Selects the type of potential landscape (e.g., free particle, barrier, well).
  - V_0: The strength (height or depth) of the potential feature.
    - For a barrier (V_flag == 2): A higher V_0 will lead to more reflection and less transmission (tunneling). The energy of the incident wavepacket relative to V_0 is critical.
    - For a well (V_flag == 3): A deeper well (larger positive V_0 for a potential $-V_0$) can lead to bound states or trapping of the wavepacket.
  - x_0, y_0, r: Define the geometry (center and radius/extent) of the potential feature. These determine how the wavepacket interacts spatially with the potential (e.g., head-on collision, glancing interaction).

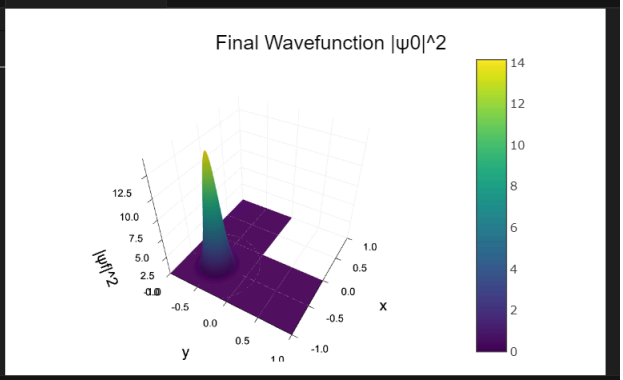
6. Results

main.jl M functions.jl M

main.jl > ...

```
6
7   # Victor Emmanuel Meimaris 23/3/25: Created main.jl for project
8   # in a 2D L shaped domain, creating my own solves and combining
9
10  # Input variables
11  domain = (-1, 1) # In nanometers
12  time = (0, 1)
13  domain_min = domain[1]
14  max_length = 0.005
15  overlaps = 40 # For domain decomposition
16  iterations = 1 #3000 #crank nicolson frames
17
18  # Wavefunction parameters
19  sigma = 0.15
20  y0 =  - 0.5
21  x0 = - 0.5
22  ky =  0.0 # + goes to the negative direction
23  kx = -20.0 # - 5.0
```

Julia Plots (18/18)

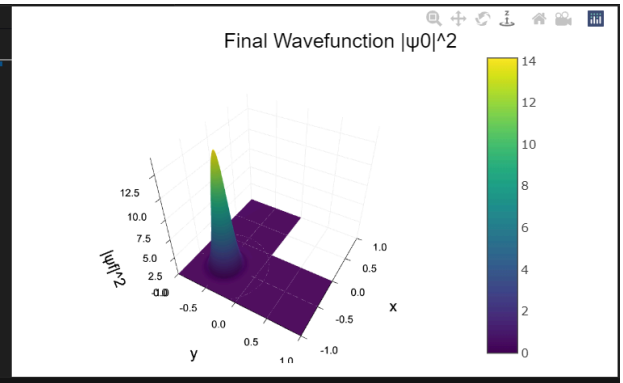


PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

```
WARNING: replacing module Functions.
Creating F.E.M matrices...
Finished creating matrices from 120000 elements in 34.56 seconds
Number of elements: 120000
Time step: 1.0e-6 picoseconds
Normalized integral check: 1.0000000000000002 + 0.0im
Initial energy: 5.514892441305801 eV
Multigrid with domain decomposition method using 40 overlaps
Time elapsed creating matrices for multigrid before loop:0.053
Energy: [5.51489244129943;;] eV
Time elapsed using backslash : 0.045 seconds
Time elapsed using bicgstab solver alone: 0.177 seconds
Time elapsed using dd and bicgstab solver : 0.489 seconds
Time elapsed using dd, multigrid and bicgstab solver : 0.828 seconds
0

julia>
```

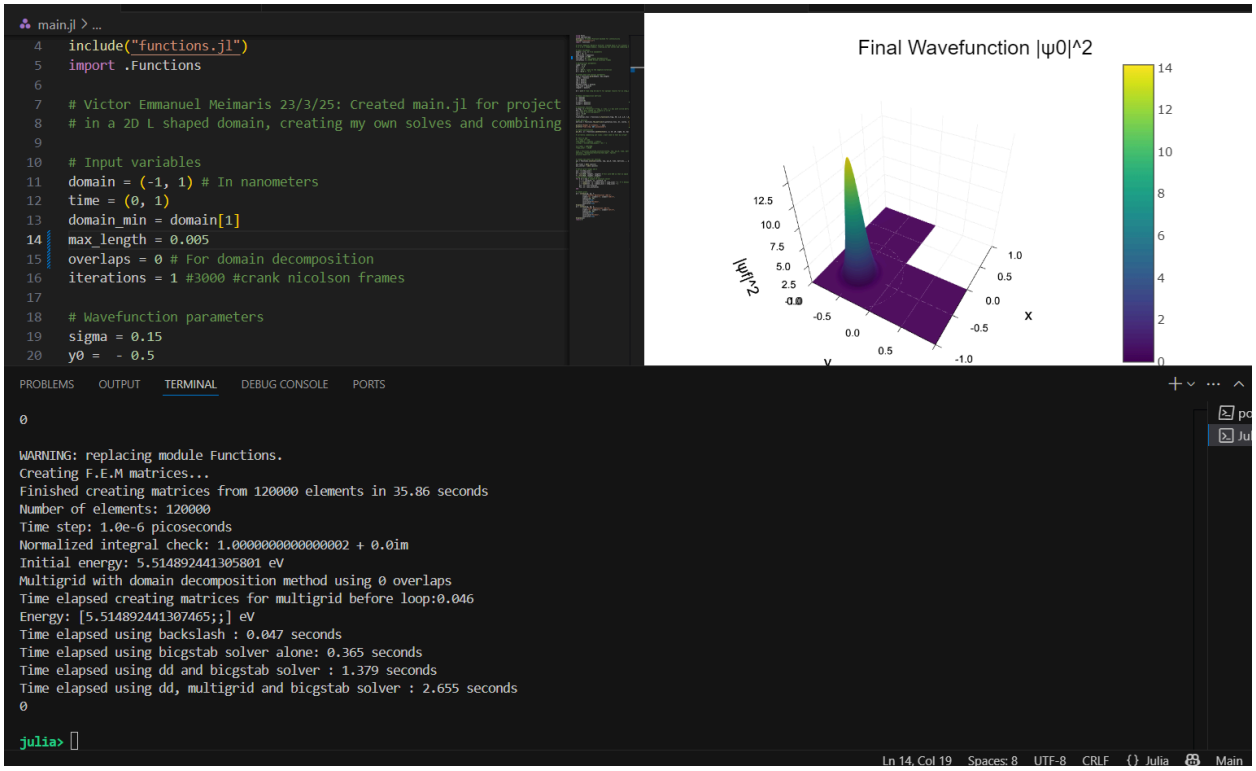
main.jl > ...

```
6
7   # Victor Emmanuel Meimaris 23/3/25: Created main.jl for project
8   # in a 2D L shaped domain, creating my own solves and combining
9
10  # Input variables
11  domain = (-1, 1) # In nanometers
12  time = (0, 1)
13  domain_min = domain[1]
14  max_length = 0.005
15  overlaps = 20 # For domain decomposition
16  iterations = 1 #3000 #crank nicolson frames
17
18  # Wavefunction parameters
19  sigma = 0.15
20  y0 =  - 0.5
21  x0 = - 0.5
22  ky =  0.0 # + goes to the negative direction
23  kx = -20.0 # - 5.0
```



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

```
WARNING: replacing module Functions.
Creating F.E.M matrices...
Finished creating matrices from 120000 elements in 35.72 seconds
Number of elements: 120000
Time step: 1.0e-6 picoseconds
Normalized integral check: 1.0000000000000002 + 0.0im
Initial energy: 5.514892441305801 eV
Multigrid with domain decomposition method using 20 overlaps
Time elapsed creating matrices for multigrid before loop:0.053
Energy: [5.514892441298984;;] eV
Time elapsed using backslash : 0.047 seconds
Time elapsed using bicgstab solver alone: 0.187 seconds
Time elapsed using dd and bicgstab solver : 0.497 seconds
Time elapsed using dd, multigrid and bicgstab solver : 0.734 seconds
0

julia>
```

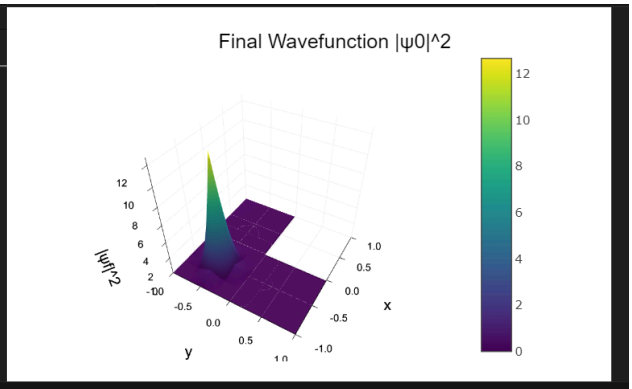


As you can see with less overlap the results were slightly better for the exact same parameters, but for even less the results were significantly worse.

Now for different max_length values going from 0.25 to 0.005 the energy as well as the wavefunction itself converge towards the actual solution and it takes more and more time due to the number of elements significantly increasing from 48 up to 120.000 elements.

main.jl > ...

```
6
7   # Victor Emmanuel Meimaris 23/3/25: Created main.jl for project
8   # in a 2D L shaped domain, creating my own solves and combining
9
10  # Input variables
11  domain = (-1, 1) # In nanometers
12  time = (0, 1)
13  domain_min = domain[1]
14  max_length = 0.25
15  overlaps = 2 # For domain decomposition
16  iterations = 1 #3000 #crank nicolson frames
17
18  # Wavefunction parameters
19  sigma = 0.15
20  y0 =  - 0.5
21  x0 = - 0.5
22  ky =  0.0 # + goes to the negative direction
23  kx = -20.0 # - 5.0
```

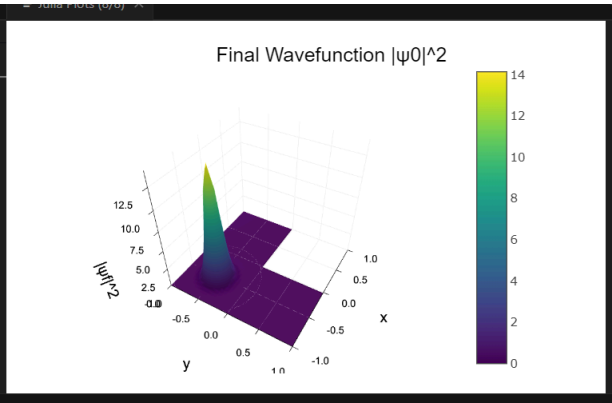


PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

```
WARNING: replacing module Functions.
Creating F.E.M matrices...
Finished creating matrices from 48 elements in 0.55 seconds
Number of elements: 48
Time step: 1.0e-6 picoseconds
Normalized integral check: 1.0000000000000004 + 0.0im
Initial energy: 2.258934770240092 eV
Multigrid with domain decomposition method using 2 overlaps
Time elapsed creating matrices for multigrid before loop:0.0
Energy: [2.2589347701563067;;] eV
Time elapsed using backslash : 0.0 seconds
Time elapsed using bicgstab solver alone: 0.0 seconds
Time elapsed using dd and bicgstab solver : 0.0 seconds
Time elapsed using dd, multigrid and bicgstab solver : 0.001 seconds
0

julia>
```

main.jl > ...

```
6
7   # Victor Emmanuel Meimaris 23/3/25: Created main.jl for project
8   # in a 2D L shaped domain, creating my own solves and combining
9
10  # Input variables
11  domain = (-1, 1) # In nanometers
12  time = (0, 1)
13  domain_min = domain[1]
14  max_length = 0.05
15  overlaps = 5 # For domain decomposition
16  iterations = 1 #3000 #crank nicolson frames
17
18  # Wavefunction parameters
19  sigma = 0.15
20  y0 =  - 0.5
21  x0 = - 0.5
22  ky =  0.0 # + goes to the negative direction
23  kx = -20.0 # - 5.0
```



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

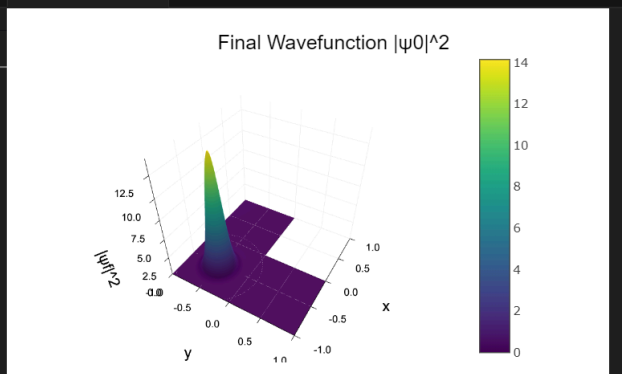
```
WARNING: replacing module Functions.
Creating F.E.M matrices...
Finished creating matrices from 1200 elements in 0.91 seconds
Number of elements: 1200
Time step: 1.0e-6 picoseconds
Normalized integral check: 0.9999999999999998 + 0.0im
Initial energy: 4.770457997971262 eV
Multigrid with domain decomposition method using 5 overlaps
Time elapsed creating matrices for multigrid before loop:0.001
Energy: [4.770457997963259;;] eV
Time elapsed using backslash : 0.0 seconds
Time elapsed using bicgstab solver alone: 0.011 seconds
Time elapsed using dd and bicgstab solver : 0.002 seconds
Time elapsed using dd, multigrid and bicgstab solver : 0.061 seconds
0

julia>
```

main.jl M ● functions.jl M

main.jl > ...

```
6
7   # Victor Emmanuel Meimaris 23/3/25: Created main.jl for project
8   # in a 2D L shaped domain, creating my own solves and combining
9
10  # Input variables
11  domain = (-1, 1) # In nanometers
12  time = (0, 1)
13  domain_min = domain[1]
14  max_length = 0.01
15  overlaps = 20 # For domain decomposition
16  iterations = 1 #3000 #crank nicolson frames
17
18  # Wavefunction parameters
19  sigma = 0.15
20  y0 =  - 0.5
21  x0 = - 0.5
22  ky =  0.0 # + goes to the negative direction
23  kx = -20.0 # - 5.0
```

Julia Plots (12/12)



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

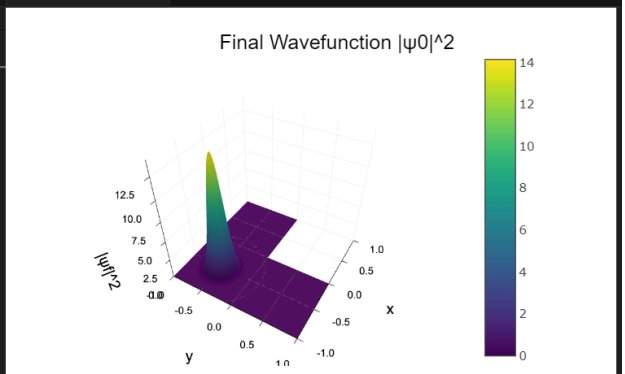
```
WARNING: replacing module Functions.
Creating F.E.M matrices...
Finished creating matrices from 30000 elements in 9.24 seconds
Number of elements: 30000
Time step: 1.0e-6 picoseconds
Normalized integral check: 0.9999999999999999 + 0.0im
Initial energy: 5.490853992905724 eV
Multigrid with domain decomposition method using 20 overlaps
Time elapsed creating matrices for multigrid before loop:0.022
Energy: [5.490853992905723;;] eV
Time elapsed using backslash : 0.01 seconds
Time elapsed using bicgstab solver alone: 0.03 seconds
Time elapsed using dd and bicgstab solver : 0.221 seconds
Time elapsed using dd, multigrid and bicgstab solver : 0.18 seconds
0

julia>
```

main.jl M ● functions.jl M

main.jl > ...

```
6
7   # Victor Emmanuel Meimaris 23/3/25: Created main.jl for project
8   # in a 2D L shaped domain, creating my own solves and combining
9
10  # Input variables
11  domain = (-1, 1) # In nanometers
12  time = (0, 1)
13  domain_min = domain[1]
14  max_length = 0.005
15  overlaps = 20 # For domain decomposition
16  iterations = 1 #3000 #crank nicolson frames
17
18  # Wavefunction parameters
19  sigma = 0.15
20  y0 =  - 0.5
21  x0 = - 0.5
22  ky =  0.0 # + goes to the negative direction
23  kx = -20.0 # - 5.0
```

Julia Plots (14/14)



PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE PORTS

```
WARNING: replacing module Functions.
Creating F.E.M matrices...
Finished creating matrices from 120000 elements in 36.13 seconds
Number of elements: 120000
Time step: 1.0e-6 picoseconds
Normalized integral check: 1.0000000000000002 + 0.0im
Initial energy: 5.514892441305801 eV
Multigrid with domain decomposition method using 20 overlaps
Time elapsed creating matrices for multigrid before loop:0.052
Energy: [5.514892441298984;;] eV
Time elapsed using backslash : 0.052 seconds
Time elapsed using bicgstab solver alone: 0.427 seconds
Time elapsed using dd and bicgstab solver : 0.256 seconds
Time elapsed using dd, multigrid and bicgstab solver : 0.827 seconds
0

julia>
```

7. Conclusion

The provided Julia script offers a capable tool for simulating and visualizing 2D quantum wavepacket dynamics using the Finite Element Method and the Crank-Nicolson scheme. Its modular design allows for easy modification of initial conditions and potential landscapes, facilitating the study of various quantum phenomena such as tunneling, scattering, and particle confinement within an L-shaped geometry just like shown in the animations. The script also provides a platform for exploring and comparing various advanced iterative linear solvers, highlighting the trade-offs between direct methods, standalone iterative methods, and more complex approaches like domain decomposition and multigrid in a serial computing context. These along with the use of sparse matrices and appropriate numerical integration techniques will contribute to the computational feasibility of the simulations in the future when simulating with millions of elements.

Further development could focus on optimizing the iterative solvers, particularly for parallel execution where methods like domain decomposition and multigrid can offer significant advantages, extending to 3D simulations, or incorporating different types of boundary conditions.

8. References and Further Reading

General FEM for the Schrödinger Equation

- **Cancès, E., Defranceschi, M., Kutzelnigg, W., Le Bris, C., & Maday, Y. (2003). Computational Quantum Chemistry: A Primer.** In *Handbook of Numerical Analysis, Vol. X*. North-Holland. (Provides a comprehensive overview of the mathematical and numerical foundations for computational chemistry, including the use of FEM).
- **Puelz, C., & Gubernatis, J. E. (2018). The finite-element method for the Schrödinger equation.** *American Journal of Physics, 86*(6), 441-448. (A pedagogical introduction to applying FEM to the Schrödinger equation, suitable for students).
- **Ram-Mohan, L. R. (2002). *Finite Element and Boundary Element Applications in Quantum Mechanics*.** Oxford University Press. (A book dedicated to the application of FEM in quantum mechanics).

Schrödinger Equation in 2D and Wavepacket Dynamics

- **Goldberg, A., Schey, H. M., & Schwartz, J. L. (1967). Computer-Generated Motion Pictures of One-Dimensional Quantum-Mechanical Transmission and Reflection Phenomena.** *American Journal of Physics, 35*(3), 177-186. (A classic paper,

though 1D, that pioneered the visualization of wavepacket dynamics).

- **Sullivan, D. M. (2012). *Quantum Mechanics for Electrical Engineers*.** John Wiley & Sons. (Chapter 3 provides a practical guide to numerically solving the 2D TDSE using the Finite-Difference Time-Domain (FDTD) method, which shares many concepts with FEM).

1. **Finite Element Method:**
    - Zienkiewicz, O. C., Taylor, R. L., & Zhu, J. Z. (2013). *The Finite Element Method: Its Basis and Fundamentals* (7th ed.). Butterworth-Heinemann.
    - Reddy, J. N. (2019). *An Introduction to the Finite Element Method* (4th ed.). McGraw-Hill Education.
2. **Numerical** Solution of PDEs **& Crank-Nicolson:**
    - LeVeque, R. J. (2007). *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. SIAM.
    - Morton, K. W., & Mayers, D. F. (2005). *Numerical Solution of Partial Differential Equations: An Introduction* (2nd ed.). Cambridge University Press.
3. **Iterative Methods for Linear Systems (including BiCGSTAB):**
    - Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM.
    - Van der Vorst, H. A. (1992). Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing, 13*(2), 631-644.
4. **Domain Decomposition Methods:**
    - Toselli, A., & Widlund, O. (2005). *Domain Decomposition Methods: Algorithms and Theory*. Springer.
    - Smith, B. F., Bjørstad, P. E., & Gropp, W. (1996). *Domain* Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations. Cambridge University Press.
    - **Chan, T. F., & Mathew, T. P. (1994). Domain Decomposition Algorithms.** *Acta Numerica, 3*, 61-143. (A comprehensive survey of domain decomposition methods).
    - **Dryja, M., & Widlund, O. B. (1990). Towards a unified theory of domain decomposition algorithms for elliptic problems.** In T. F. Chan et al. (Eds.), *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*. SIAM. (Discusses the theoretical underpinnings, including Additive Schwarz).
    - **Pavarino, L. F., & Widlund, O. B. (1996). A polylogarithmic bound for an iterative substructuring method for the p-version finite element method in two dimensions.** *SIAM Journal on Numerical Analysis, 33*(4), 1303-1322. (While focused on the p-version, this paper is a good example of advanced analysis of DD methods applied to problems with corner singularities like the

L-shaped domain).

5. **Multigrid Methods:**
    - Briggs, W. L., Henson, V. E., & McCormick, S. F. (2000). *A Multigrid Tutorial* (2nd ed.). SIAM.
    - Trottenberg, U., Oosterlee, C. W., & Schüller, A. (2001). *Multigrid*. Academic Press.
    - **Bank, R. E., & Dupont, T. (1981). An Optimal Order Process for Solving Finite Element Equations.** *Mathematics of Computation, 36*(153), 35-51. (A foundational paper on multigrid methods for FEM).
    - **Apel, T. (1999). *Anisotropic Finite Elements: Local Refinement and Regularity of Solutions*.** Teubner. (Discusses the necessity of anisotropic mesh refinement for problems with singularities, such as on an L-shaped domain, to maintain optimal multigrid performance).
    - **Stevenson, R. (2008). The completion of hierarchical sets of functions.** *Calcolo, 45*(1), 1-21. (Deals with the construction of stable hierarchical bases, which are crucial for multigrid methods on domains with re-entrant corners).