Mathematical modelling and numerical simulation of driving buses and vibrations door

Student:

Anouchka Desmettre

Supervisors:

N. Barral - ENSEIRB MATMECA

D. Lahaye - TU DELFT

Second Year Internship - 2023/2024

Abstract

This report presents the work carried out during a three-month internship at TU Delft in collaboration with Ventura, a leading manufacturer in the bus door industry. The objective of the internship was to develop numerical models that simulate the behavior of bus doors under various operational stresses.

The project began by exploring the fundamental principles of mechanical modeling using single and double-mass systems to represent the behavior of the door components. By understanding these simple systems, I was able to extend the models to more complex structures, ultimately applying them to bus doors, which require the simulation of multiple interacting masses. The transition from one mass to multi-mass systems allowed for a more accurate representation of the dynamic forces experienced by the door in real-world conditions.

The second phase of the project involved the application of the diffusion equation to model how stress and deformation propagate through the door over time. Both stationary and time-dependent formulations were examined, providing insight into how doors react under impacts. The solution of the 1D and 2D diffusion equations laid the groundwork for more complex simulations.

The final part of the report focuses on the solution of the biharmonic equation, which models the bending and deformation of elastic structures. Both analytical and numerical methods were employed to solve the biharmonic equation in 1D and 2D, providing a comprehensive understanding of how the bus door deforms under various loads. The project culminated in the successful application of these methods to simulate a time-dependent 2D biharmonic problem, allowing for accurate predictions of the door's structural behavior under repeated stress scenarios.

The report concludes with reflections on the potential for further research, particularly in the areas of fatigue and fracture mechanics, and industrial design optimization. These upcoming studies will build upon the foundations laid during the internship, offering opportunities to refine the models and explore new approaches to sustainable engineering in transportation systems.

Table of Contents

1 Introduction

As cities grow and prioritize sustainability, public transportation must evolve to become more efficient and environmentally friendly. Leading manufacturers such as Volvo, VDL, and Scania are at the forefront of this transition, focusing on bus door mechanisms as critical components for safety, operational efficiency, and reduced environmental impact. These systems must be adaptable to modern demands, ensuring both robustness and sustainability as cities increase their reliance on public transit.

TU Delf is a university renowned for its pioneering research in engineering and environmental sustainability. Conducted in collaboration with Ventura, a leading manufacturer of bus doors, the project aimed to improve door mechanisms using advanced numerical modeling techniques. The partnership between Ventura and TU Delft provided a unique opportunity to merge practical industry challenges with cutting-edge academic research.

Throughout the internship, the project focused on enhancing the understanding of how bus doors behave under various operational stresses in order to explore ways to reduce the environmental impact of these systems.

This was achieved through the development of numerical simulations based on both mathematical modeling and physical principles. These simulations examined a primary operational scenario: the impact of falling objects on door mechanisms, modeled as a inpulsive force.

Ventura Bus Door

The report is structured progressively, starting with fundamental models and advancing toward complex simulations relevant to bus door dynamics. Section 2 introduces the single mass system, where both analytical and numerical methods are applied to establish a foundation for understanding mechanical responses. Section 3 expands this to a two-mass system, critical for modeling the interconnected masses found in bus doors. In Section 4, we explore the diffusion equation, starting with stationary models and gradually incorporating time-dependency in 1D and 2D. This section forms the basis for understanding stress distribution, essential for bus door analysis. Section 5 culminates with the biharmonic equation, where the groundwork laid in previous sections allows us to simulate how the bus door bends under real-world conditions.

This internship report thus outlines the mathematical and engineering approaches used to tackle these complex industry challenges, contributing to the broader goal of sustainable public transportation.

2 Modeling a Single Point Mass System

2.1 Case Study and Mathematical Model

The aim of this section is to model the vibration dynamics of the bus door using a *mass-spring-damper system* [1], as shown in Figure 1. This system simulates the behavior of the door when it is subjected to an impulsive external force $F(t)$, such as an object falling onto the door. We model the door (leaf and frame) as a point mass m, assuming that its physical dimensions are negligible. The *spring* represents the elastic force that restores the door to its equilibrium position after displacement, with stiffness characterized by the spring constant k. The *damper* accounts for the resistive forces that dissipate the energy of oscillations, where the damping rate is governed by the coefficient β. The displacement of the mass is denoted by $x(t)$.

This system is characterized by the following ordinary differential equation (ODE):

$$m\ddot{x}(t) + \beta\dot{x}(t) + kx(t) = F(t) \quad (1)$$

Where $\dot{x}(t)$ represents the velocity and $\ddot{x}(t)$ the acceleration.The system starts from rest with the initial conditions:

$$\begin{cases} x(0) &= 0 \quad \text{(initial position)}, \\ \dot{x}(0) &= 0 \quad \text{(initial velocity)} \end{cases} \quad (2)$$

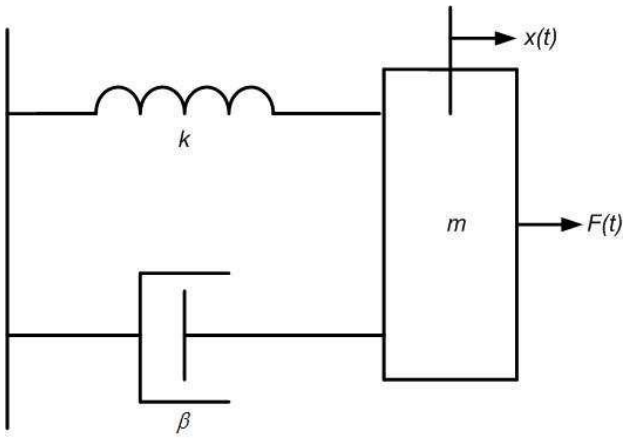


Figure 1: Single Point Mass System

2.2 Analytical Approaches

To study the dynamic behavior of the bus door system, we begin by solving the second-order differential equation for a Dirac delta function which can be approximated in the code, with the appropriate parameters, by a Gaussian pulse:

$$F(t) = F_0\delta(t - t_0) \approx \frac{F_0}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(t - t_0)^2}{2\sigma^2}\right)$$

We chose two different methods to find the analytical solution to this equation:

- **The method of homogeneous and particular solutions**, which involves solving for both the general and particular solutions to the equation.
- **The Laplace Transform**, which offers a more systematic way of solving differential equations, particularly for systems subject to impulsive forces like the Dirac delta function.

After solving the equation using both methods, we found that the calculations for the first method (homogeneous and particular solutions) are more laborious, especially when dealing with complex external forces.

2.2.1 Undamped Solution for a Dirac delta Excitation

When the damping coefficient $\beta = 0$, the system behaves as a pure harmonic oscillator. The solution for the Dirac delta force is given by:

$$x(t) = \frac{F_0}{m\omega_0^2} \sin(\omega_0(t - t_0))u(t - t_0) \tag{3}$$

This represents undamped oscillations at the natural frequency $\omega_0 = \sqrt{\frac{k}{m}}$. The amplitude depends on the magnitude of the applied force F_0 and the mass. By first solving the undamped case, we can validate the correctness of our numerical implementation in a simpler scenario.

2.2.2 Damped Solution for a Dirac delta Excitation

When the damping coefficient $\beta \neq 0$, the system's energy dissipates over time. The solution for the damped case is:

$$x(t) = \frac{F_0}{m\gamma^2} e^{-\alpha(t-t_0)} \sin(\gamma(t - t_0))u(t - t_0) \tag{4}$$

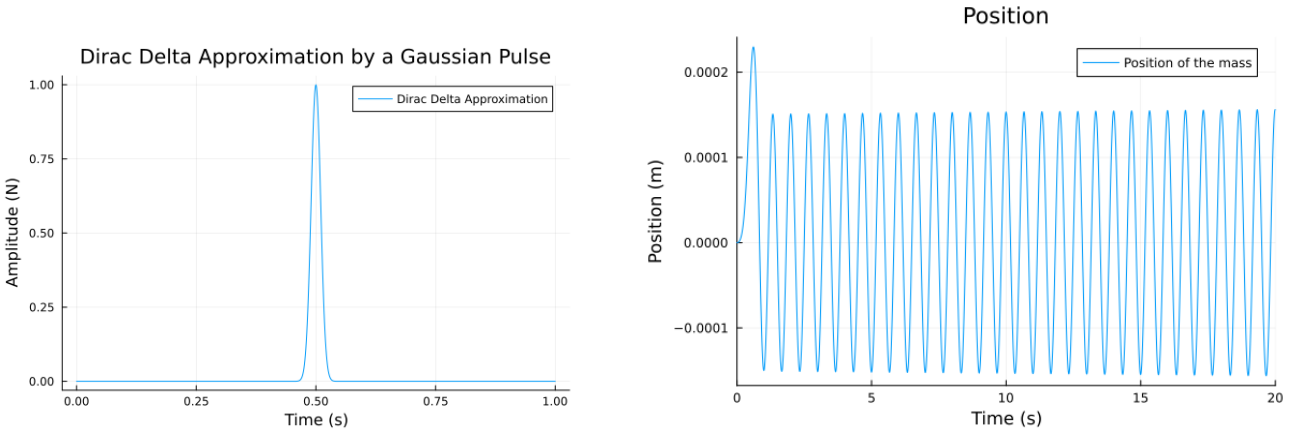
Where $\alpha = \frac{\beta}{2m}$ and $\gamma = \sqrt{\omega_0^2 - \alpha^2}$. Here, the oscillations gradually decay due to the damping effect, and the system oscillates at a lower frequency γ.

In both cases, the analytical solutions allow us to compare the results with those obtained from numerical simulations, ensuring consistency and accuracy. This step-by-step comparison is crucial in verifying the validity of our numerical models.

2.3 Numerical Validation

2.3.1 Code Implementation

Two approaches were used to solve the mass-spring-damper system subjected to a Dirac impulse:



(a) Dirac Delta Approximation

(b) Mass Position over Time

Figure 2: Dirac Delta Approximation and Solution of the Single Mass System

- The solver *Tsit5*, a fifth-order explicit Runge-Kutta method known as the Tsitouras 5/4 method and suitable for non-stiff problems, was employed.

Initially, we solved the problem using *SecondOrderODEProblem*.

- We transformed the second-order ODE into a first-order system by introducing the velocity variable $v(t) = \dot{x}(t)$, leading to the following system:

$$\begin{cases} \dot{x}(t) = v(t) \\ \dot{v}(t) = \frac{1}{m}\left(F_{\text{external}}(t) - \beta v(t) - k u(t)\right) \end{cases}$$

Both methods yield identical results, as shown in Figure 2, providing an initial confirmation of the correctness of our code.

2.3.2 Comparison with Analytical Solution

We validated the previous numerical solution by comparing it to the analytical Solution 4 and analyzing the error. We computed the absolute and relative errors and plotted their logarithms (base 10) to visualize the error evolution over time. We use tolerance to set the allowable limits for the two errors, ensuring that the solution remains accurate and stable over time.

As can be seen in Figure 3, the absolute error oscillates 10^{-9}, while the relative error ranges from 10^{-3} to 10^{-6}. Between 0s and 1s, both errors remain very low, indicating that the Dirac impulse at 0.5s is accurately captured in the same way by both the analytical and numerical solutions.

These results confirm the accuracy of the numerical solution, showing that the method maintains stability and precision throughout the simulation.

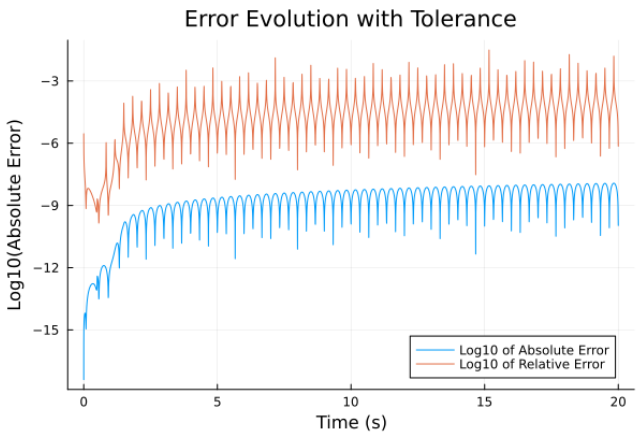


Figure 3: Validation of the numerical study for the single point mass system

3 Modeling a Two-Mass System

As before, we first verified the system without damping to ensure accuracy, using a step-by-step approach to adopt an optimized and efficient method.

3.1 Two-Mass System without Damping

3.1.1 Mathematical Model

The two-mass system (Figure 4 and EDO 5) is described by the following equation of motion, where the same Dirac impulse from Figure 2 is applied to m_1:

$$\begin{cases} m_1\ddot{x}_1 + k_1x_1 + k_2(x_1 - x_2) = F_0\delta(t - t_0) \\ m_2\ddot{x}_2 + k_3x_2 + k_2(x_2 - x_1) = 0 \end{cases} \tag{5}$$

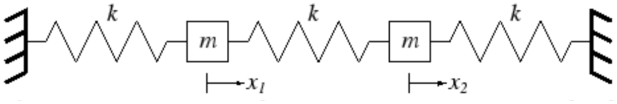


Figure 4: Two-Point Mass System without damping

3.1.2 Analytical solution

Here are the different steps we went through to determine the analytical solution:

1. Definition of the connectivity matrix B, the diagonal spring constants matrix D, and the diagonal mass matrix M to describe the system's kinematics and dynamics.

2. Calculation of the weighted Laplacian matrix $K = B^\top DB$, which captures the interactions between masses and springs.

3. Extraction of the submatrix K_0 by removing the rows and columns corresponding to fixed points, focusing on the mobile nodes.

4. Writing the system equation of motion for a dirac impulse applied to one the first mass: $M\ddot{\vec{x}}(t) + K_0\vec{x}(t) = \vec{F}(t)$.

5. Transforming the second-order system into a first-order state-space form and solving for the state vector z: $\dot{z} = Az + F \iff \dot{z} = \begin{pmatrix} 0 & I \\ -M^{-1}K_0 & 0 \end{pmatrix} z + \begin{pmatrix} 0 \\ M^{-1}F \end{pmatrix}$

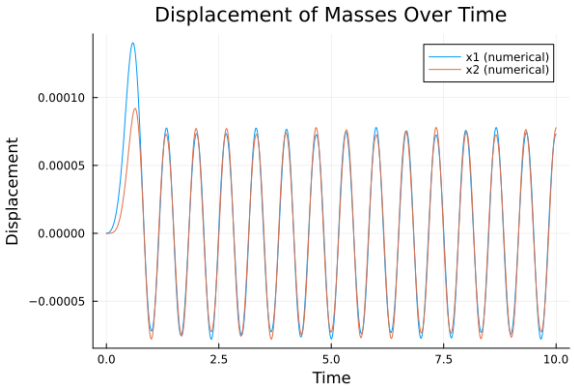
6. After facing many complex calculations, we simplified the system by setting $k_1 = k_2 = k_3 = k$, which allowed us to compute the four eigenvalues λ_i and eigenvectors V_i of the system matrix A more easily : $\lambda_i = \pm\sqrt{-\frac{k}{m_1} - \frac{k}{m_2} \pm \frac{k\sqrt{m_1^2 - m_1m_2 + m_2^2}}{m_1m_2}}$

7. We transform the system into modal coordinates by defining $x = Vq$. The system simplifies to: $\ddot{q}_i + \lambda_i q_i = F_{modal,i}$.

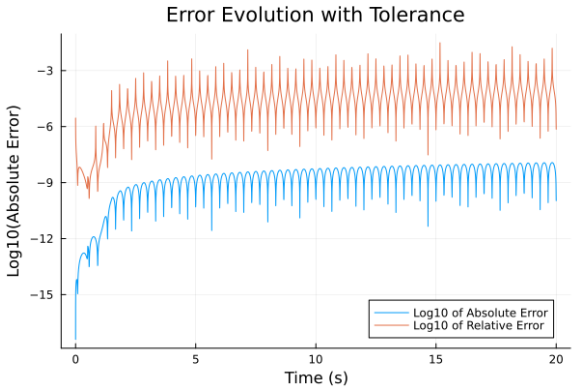
8. Converting back to the original coordinates to obtain the displacements and velocities $x_i(t)$.

3.1.3 Numerical validation

As before, we plot the numerical solution and calculate the error using the specified tolerances (Figure 5). The error is of the same order of magnitude as in the single mass system, which is consistent given the similar numerical methods and tolerances applied. This suggests that our approach remains accurate and reliable for the two-mass system.



(a) Solutions of the two-mass systems.



(b) Numerical vs analytical solutions.

Figure 5: Validation of the numerical study for the two-mass system without damping

3.2 Two-Mass System with Damping

3.2.1 Mathematical Model

We consider the same system as before, but three dampers c_1, c_2, and c_3 connected along with the springs have been added (Figure 6 and EDO 6).

$$\begin{cases} m_1\ddot{x}_1 + c_1\dot{x}_1 + c_2(\dot{x}_1 - \dot{x}_2) + k_1x_1 + k_2(x_1 - x_2) & = F_0\delta(t - t_0) \\ m_2\ddot{x}_2 + c_3\dot{x}_2 + c_2(\dot{x}_2 - \dot{x}_1) + k_3x_2 + k_2(x_2 - x_1) & = 0 \end{cases} \tag{6}$$

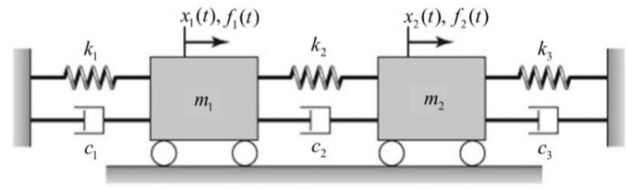
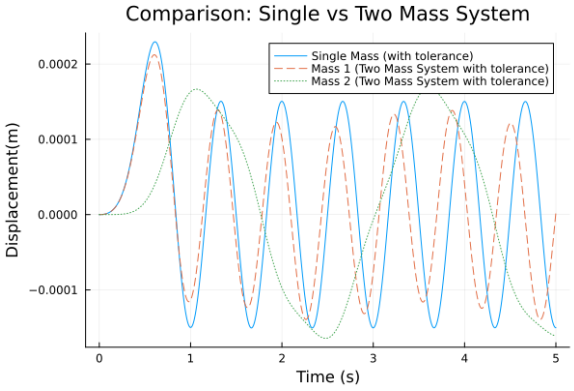


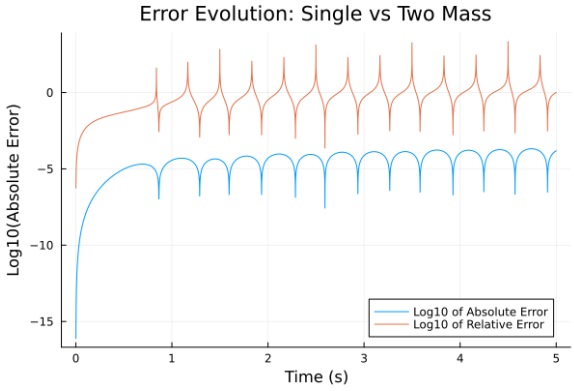
Figure 6: Two-Point Mass System with damping.

3.2.2 Numerical Validation

Given the complexity of analytical calculations for the two-mass system, we validated the model by replicating the simpler first-order system. To do this, we set the parameters: $k_3 = 0$, making the second spring inactive, k_2 large to couple the two masses, and $m_1 = m_2$ with $c_2 = c_3 = 0$ and $c_1 = \beta$, effectively simplifying the system to behave like a single mass-spring-damper system. This configuration allows us to verify the model's accuracy before moving to more complex cases.



(a) Displacement comparison between the single mass and two-mass systems.



(b) Error analysis: Single mass vs mass1 two-mass systems.

Figure 7: Validation of the two-mass system with damping.

As shown in Figure 7, mass 1 behaves similarly to the single-mass system, as it directly experiences the external force. Mass 2 follows the motion of mass 1 due to the strong coupling spring but does not receive the direct impulse.

The results indicate that the relative error between the single mass of the single-mass system and mass 1 of the two-mass system is approximately 10%, reflecting only minor differences between the systems. Meanwhile, the absolute error remains low, around 10^{-5}, confirming that both systems exhibit similar behavior and validating our approach.

3.3 Numerical Stiffness Study

We analyze the effects of numerical stiffness (i.e., the varying time scales in the ODE system) on the dynamic behavior of a mechanical system by significantly varying the mass values (Hooke's Law: $k = m \cdot \omega^2$) [2].

In the simulation results (Figure 8), as stiffness increases (through greater mass), the system's natural frequencies increase [3].
Consequently, numerical solvers must reduce the time step (Δt) to accurately capture these faster oscillations. The average time steps for different mass combinations were:

- **Scenario 1**: $m_2 = 50\,\text{kg}$: $\Delta t = 0.233\,\text{s}$
- **Scenario 2**: $m_2 = 100\,\text{kg}$: $\Delta t = 0.222\,\text{s}$
- **Scenario 3**: $m_2 = 500\,\text{kg}$: $\Delta t = 0.227\,\text{s}$

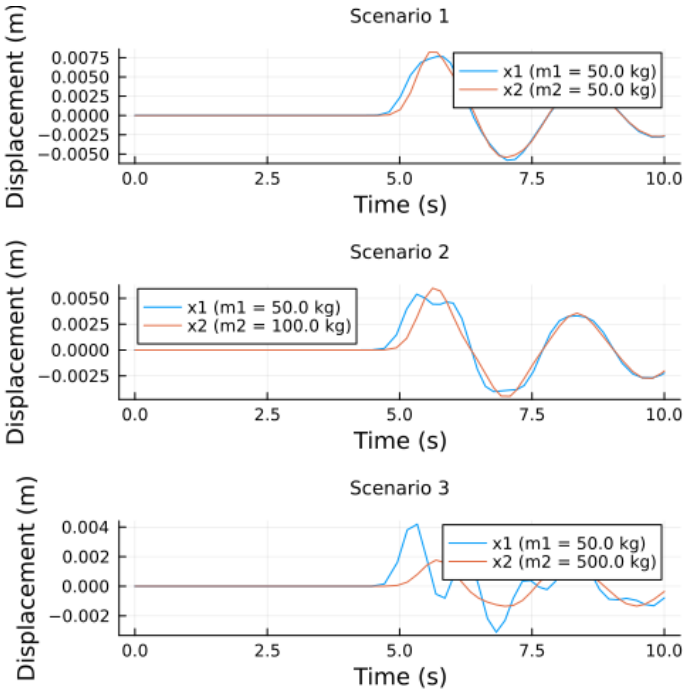


Figure 8: Effect of different masses on system stiffness

In systems where mass significantly increases (Scenario 3), the dynamics shift from stiffness-dominated to inertia-dominated, allowing for slightly larger time steps due to slower system responses. If damping is not proportionally increased with mass, its relative influence decreases, leading to feasible larger Δt values as the system's oscillations slow down.

4 Diffusion Equation

We are ready now to work on the Diffusion Equation 7:

$$\frac{\partial T(x,y,t)}{\partial t} = D\nabla^2 T(x,y,t) + S(x,y) \tag{7}$$

where T represents the diffusing property (like temperature), D is the diffusion coefficient and we use a Gaussian function as the source term S.

4.1 Stationnary Diffusion Equation in 1D

We now simplify Equation 7:

$$\frac{\partial^2 T(x)}{\partial x^2} = A \exp\left(-\frac{(x-x_0)^2}{2\sigma^2}\right) \tag{8}$$

With N segments in the domain, the points are spaced uniformly with h between them. The second derivative is approximated using finite differences: $T_i'' \approx \frac{T_{i-1} - 2T_i + T_{i+1}}{h^2}$

For Dirichlet boundary conditions: $T_1 = T_a = 100$ °C and $T_{N+1} = T_b = 0$ °C. This leads to a system of equations : $A_{1D}X = S$, where S is the source vector, and A_{1D} is the Laplacian matrix:

$$A_{1D} = \frac{1}{h^2}\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 1 & -2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 1 \\ 0 & \cdots & 0 & 1 \end{bmatrix}, \quad S = \begin{bmatrix} T_a * h^2 \\ S(x_2) \\ \vdots \\ T_b * h^2 \end{bmatrix}$$

Since A_{1D} is tridiagonal and symmetric, we can efficiently perform an LU decomposition, which leads to the coherent result shown in Figure 15.

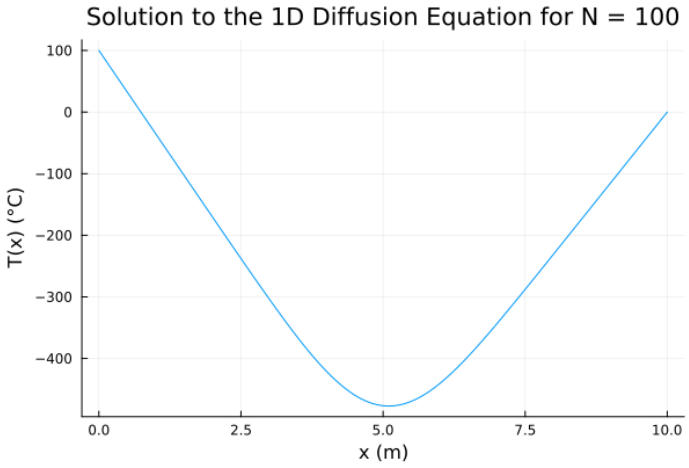


Figure 9: 1D Diffusion Solution with Boundary Conditions

4.2 Stationnary Diffusion Equation in 2D

Having successfully solved the 1D diffusion equation, we are now ready to extend our approach to a more complex 2D scenario, we are working on :

$$\nabla^2 T = S(x, y) \tag{9}$$

4.2.1 Numerical Solution

To solve the diffusion equation over a 2D domain, we use the **Kronecker Product Approach**.

The matrix A_{xx} applies the 1D diffusion operation along the x-direction for each fixed y:

$$A_{xx} = A_{1Dx} \otimes I_{N_y+1}$$

Here, A_{1Dx} is the tridiagonal matrix from the 1D case, and I_{N_y+1} is the identity matrix of size $(N_y + 1) \times (N_y + 1)$. This results in a block diagonal matrix where each block corresponds to a row of grid points in the y-direction. Similarly, A_{yy} handles diffusion along the y-direction for each fixed x:

$$A_{yy} = I_{N_x+1} \otimes A_{1Dy}$$

This configuration also results in a block diagonal matrix, but each block corresponds to a column of grid points in the x-direction.

The combined system matrix A_{2D} for the 2D scenario, which effectively models diffusion in both dimensions, is given by the sum of the two matrices:

$$A_{2D} = A_{xx} + A_{yy}$$

The linear system can be expressed as:

$$A_{2D}X = S$$

where X is the solution vector representing the flattened matrix of temperature, and S deals with the Gaussian source term (Figure 10):

$$S(x, y) = A \exp\left(-\frac{(x - x_0)^2 + (y - y_0)^2}{2\sigma^2}\right) \tag{10}$$

By solving this system, we obtain the temperature values over the entire domain (Figure 11) that match the input parameters.

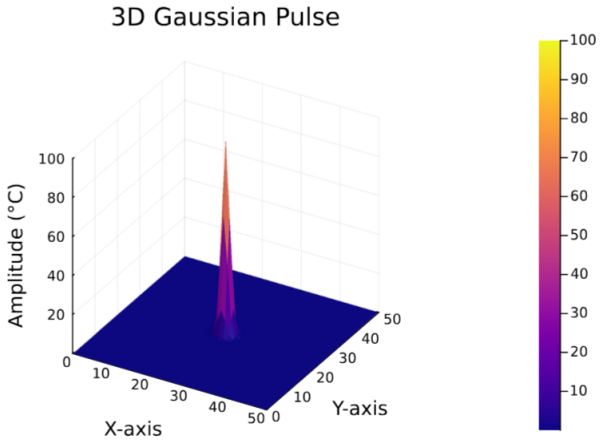


Figure 10: 3D Gaussian Pulse

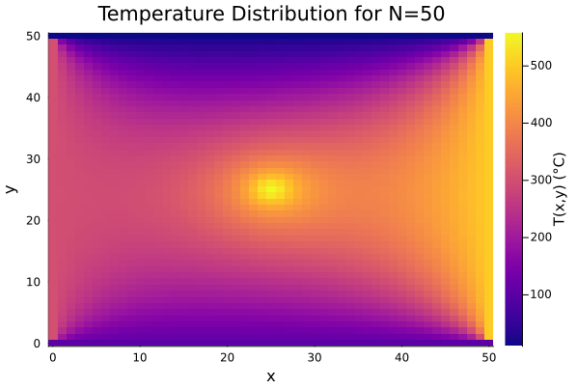


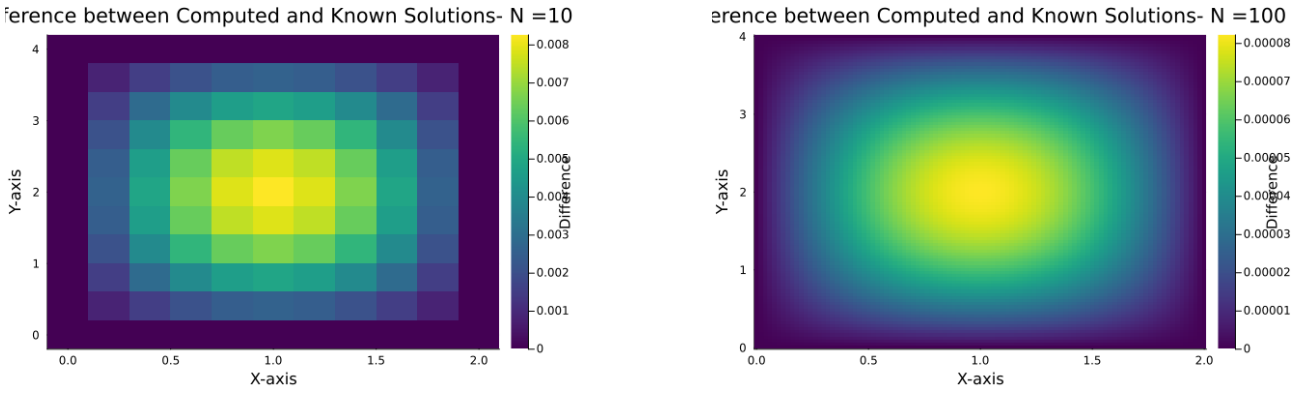
Figure 11: Numerical Solution of the 2D Diffusion Equation for $N = 50$

4.2.2 Verification Test for the Stationary Diffusion Equation using an analytical solution

We solve the stationary 2D diffusion equation using the method of separation of variables, assuming the solution can be written as $u(x, y) = X(x)Y(y)$.

$$u(x, y) = \sin\left(\frac{n\pi x}{L_x}\right) \sin\left(\frac{m\pi y}{L_y}\right) \iff u(x, y) = \sin(kx)\sin(\ell y)$$

Where m and n are integers, and L_x and L_y are the domain lengths, so k and ℓ are constants representing the wave numbers in the x- and y-directions.
In finite difference methods, the error, e, scales with the grid spacing h : $e \propto h^2$. Indeed, Figures 12 validate our simulation: - For $N = 10$: The maximum error is around 10^{-3}. With a coarser grid, the numerical solution has lower resolution, leading to larger errors. - For $N = 100$: The grid spacing is reduced by a factor of 10, and since the error scales with h^2, the error is reduced by a factor of 100, yielding an error of approximately 10^{-5}.



(a) For $N = 10$

(b) for $N = 100$

Figure 12: Error between Analytical and Numerical Solutions

As grid resolution increases, the numerical solution becomes more accurate. However, there is a trade-off: higher accuracy requires finer grids, which increase computational cost. Therefore, the choice of grid size balances accuracy with efficiency.

4.3 Time-Dependent Diffusion Equation in 2D

4.3.1 Numerical Solution of the Time-Dependent 2D Diffusion Equation

To solve the Time-Dependent 2D Eiffusion Equation (7), we tested three different numerical techniques: a custom time loop, the Explicit Euler method, and the Implicit Euler method. Each method is presented below, along with the reasoning behind why we progressed from one method to the next.

1. Time Loop Method

We began by using a simple time loop approach, where the solution is updated iteratively at each time step. The temperature distribution T^{n+1} is calculated by solving the linear system:

$$(I - \Delta t\, D\, A)T^{n+1} = T^n + \Delta t \cdot S$$

In this formulation, I is the identity matrix, and the time step Δt controls the evolution of the solution. This approach offers flexibility and is straightforward to implement, making it a good initial method. However, because it relies on direct matrix inversion at each time step, the method quickly becomes computationally expensive as the grid size increases. This limitation in efficiency prompted us to explore more advanced methods.

2. Explicit Euler Method

Next, we implemented the Explicit Euler method, which avoids the expensive matrix inversion by directly computing the solution at the next time step using:

$$T^{n+1} = T^n + \Delta t \cdot F(T^n) \iff T^{n+1} = T^n + \Delta t \cdot (DAT^n + S)$$

This method is simple to implement and computationally cheaper per time step compared to the time loop approach. However, the Explicit Euler method is limited by the Courant-Friedrichs-Lewy (CFL) condition: $\Delta t \leq \frac{h^2}{2D}$.
This condition requires extremely small time steps, especially when the diffusion coefficient D or grid resolution is high, making the method inefficient for larger simulations. Given this limitation, we adopted the Implicit Euler method.

3. Implicit Euler Method

Finally, we applied the Implicit Euler method which addresses the stability issues of the explicit method. In the implicit scheme, the temperature field at the next time step T^{n+1} is determined by solving:

$$T^{n+1} = T^n + \Delta t \cdot F(T^{n+1}) \iff T^{n+1} = T^n + \Delta t \cdot (DAT^{n+1} + S)$$

This method is unconditionally stable, meaning that larger time steps can be used without compromising stability. We used Julia's `TRBDF2()` solver, which is designed for stiff problems like diffusion. Although the Implicit Euler method requires solving a system of equations at each step (which increases the computational complexity), its ability to handle larger time steps efficiently made it the most suitable approach for this problem.

4.3.2 Verification Test for the Time-Dependent Diffusion Equation using an Analytical Solution

Based on Marc Buffat's course [4], we derive the analytical solution for the time-dependent diffusion equation:

$$u(x, y, t) = \sin(kx)\sin(\ell y)\exp(-D(k^2 + \ell^2)t)$$

This solution includes both oscillatory and diffusive components, ensuring the solution exhibits proper temporal decay. Dirichlet boundary conditions are applied, with $u(x, y) = 0$ on all boundaries, which is satisfied because $\sin(kx)$ and $\sin(\ell y)$ vanish at the edges of the domain. The source term is constructed to match the diffusion equation:

$$-\nabla^2 u(x, y, t) + \frac{\partial u(x, y, t)}{\partial t} = f(x, y, t)$$

We numerically solve the system using the three different methods and compare each numerical solutions with the analytical solution at every time step to assess the accuracy of the time integration.

Below is a comparison of the Implicit Euler method with the analytical solution:

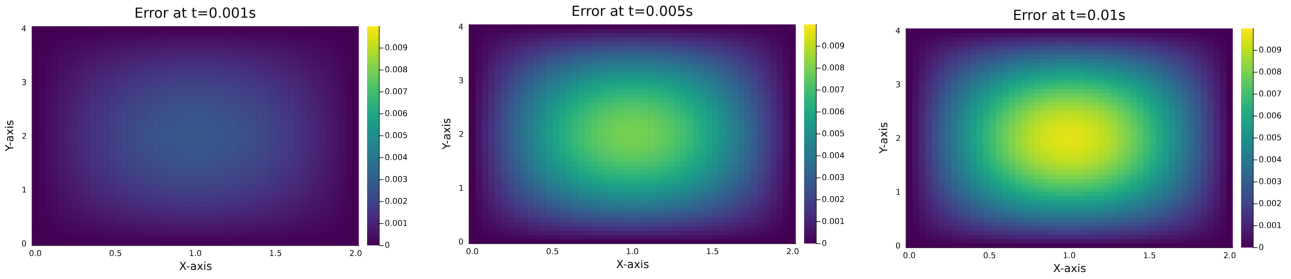


Figure 13: Error between the analytical solution and the Implicit Euler numerical solution with $\Delta t = 0.0001$.

As seen in Figure 13, the error increases as time progresses, but remains on the order of 10^{-3}, which is satisfactory. The center is where the temperature changes the most over time, leading to larger numerical errors compared to the boundaries, where Dirichlet conditions impose a strict $u(x, y) = 0$. The lower error near the boundaries suggests that the boundary conditions are correctly enforced.

5 Biharmonic Equation

Solving second-order equations prepared us for more complex problems, where higher-order discretization methods and nested system solutions are required. Each step builds our ability to handle advanced PDEs. Our goal now is to solve the time-dependent 2D biharmonic equation [5], a fourth-order partial differential equation:

$$\nabla^2 \left(EI(x, y) \times \nabla^2 u(t, x, y) \right) = \frac{\partial^2 u(t, x, y)}{\partial t^2} + S(x, y) \tag{11}$$

Here, $u(x, y, t)$ represents the displacement field of the door, E is the Young's modulus, I is the moment of inertia, $EI(x, y)$ forms the flexural rigidity, accounts for the varying material properties of the structure (e.g., frame and glass of a door), and $S(x, y)$ accounts for the external forces applied to the door. This equation helps us understand how the door bends and deforms under different loads.

5.1 Analytical Solution of the Biharmonic Equation

5.1.1 Analytical Solution of the 1D Biharmonic Equation

Before tackling the time-dependent 2D biharmonic equation, we begin by solving the simpler stationary 1D case (12). This allows us to familiarize ourselves with the concepts and boundary conditions involved in higher-order partial differential equations.

$$\frac{d^4u(x)}{dx^4} = 0 \tag{12}$$

$$\begin{cases} u(0) = 0, \quad u(1) = 0 \quad \text{(deflection)} \\ u''(0) = 0, \quad u''(1) = 0 \quad \text{(bending moment)} \end{cases} \tag{13}$$

To solve this, we use the separation of variables approach, which introduces a separation constant λ (based on [6]), an eigenvalue that depends on the boundary conditions of the system. The equation becomes:

$$u''''(x) = \lambda u(x)$$

For $\lambda > 0$ The characteristic equation is:

$$r^4 = \lambda.$$

The four roots are:

$$r_1 = \lambda^{1/4}, \quad r_2 = -\lambda^{1/4},$$
$$r_3 = i\lambda^{1/4}, \quad r_4 = -i\lambda^{1/4}$$

The general solution is a combination of trigonometric and hyperbolic functions:

$$u(x) = C_1\cos(\lambda^{1/4}x) + C_2\sin(\lambda^{1/4}x) + C_3\cosh(\lambda^{1/4}x) + C_4\sinh(\lambda^{1/4}x)$$

First Condition: $u(0) = 0$ and Third Condition: $u''(0) = 0$ lead to $C_3 = -C_1$

Second Condition: $u(1) = 0$ gives:

$$C_1\left(\cos(\lambda^{1/4}) - \cosh(\lambda^{1/4})\right) + C_2\sin(\lambda^{1/4}) + C_4\sinh(\lambda^{1/4}) = 0.$$

Fourth Condition: $u''(1) = 0$ gives:

$$-C_1\lambda^{1/2}\left(\cos(\lambda^{1/4}) + \cosh(\lambda^{1/4})\right) - C_2\lambda^{1/2}\sin(\lambda^{1/4}) + C_4\lambda^{1/2}\sinh(\lambda^{1/4}) = 0.$$

For $\lambda < 0$ We write $\lambda = -\mu^2$, where μ is a positive real number. The biharmonic equation becomes:

$$u''''(x) = -\mu^2 u(x),$$

and the characteristic equation is:

$$r^4 = -\mu^2.$$

The four roots are complex and can be written as:

$$r = \frac{\sqrt{2}}{2}(\pm 1 \pm i)\mu^{1/2}.$$

The general solution for $u(x)$ is a combination of oscillatory and exponential terms:

$$u(x) = C_1 e^{\frac{\sqrt{2}}{2}\mu x} + C_2 e^{-\frac{\sqrt{2}}{2}\mu x} + C_3 e^{i\frac{\sqrt{2}}{2}\mu x} + C_4 e^{-i\frac{\sqrt{2}}{2}\mu x}$$

This can also be rewritten in terms of sine and cosine functions:

$$u(x) = C_1\cos(\mu x) + C_2\sin(\mu x) + C_3\cosh(\mu x) + C_4\sinh(\mu x)$$

Applying Boundary Conditions
To solve for the constants C_1, C_2, and C_4, we derive two equations from the boundary conditions at $x = 1$. These two equations form a system of linear equations leading to the complete analytical solution for $u(x)$.

Applying Boundary Conditions
The process of applying boundary conditions follows similarly to the case of $\lambda > 0$, but the constants C_1, C_2, C_3, and C_4 will be adjusted to account for the complex roots and their behavior at the boundaries.

5.1.2 Analytical Solution of the 2D Biharmonic Equation

To extend this method in order to solve (14), we use a separation of variables approach with two spatial dimensions, x and y:

$$\nabla^4 u(x, y) = \lambda u(x, y), \tag{14}$$

Here, $u(x, y) = X(x)Y(y)$, and λ remains the separation constant. The general solution is constructed similarly to the 1D case, using trigonometric and hyperbolic functions in both dimensions. However, solving analytically in 2D becomes more complex due to the higher order of the equation and the need to apply boundary conditions in both the x- and y-directions, making manual calculations impractical. Instead, we focus on numerical methods, such as finite difference techniques, which allow us to efficiently approximate the solution.

Given that we've already validated our numerical approach in solving mass and diffusion equations (in 1D, 2D, stationary, and time-dependent forms), we are confident in relying on numerical solutions for the biharmonic equation without spending excessive time on its analytical resolution.

5.2 Numerical Solution sationary biharmonic Equation in 2D

By adapting Equation 11, the stationary biharmonic equation is given by:

$$\nabla^2 \left(EI(x, y) \cdot \nabla^2 u(x, y) \right) = S(x, y) \tag{15}$$

We used two methods to check whether we found identical and therefore consistent results.

5.2.1 Decomposition into Two Second-Order Problems

The biharmonic equation is decomposed into two sequential second-order Laplace equations:

$$\begin{cases} \nabla^2 m(x, y) = S(x, y), \\ EI(x, y)\nabla^2 u(x, y) = m(x, y), \end{cases}$$

Where $m(x, y)$ represents the bending moment. This decomposition simplifies the problem by allowing us to solve two second-order equations instead of directly solving the fourth-order biharmonic equation.

We begin by solving $\nabla^2 m(x, y) = S(x, y)$. This step uses the same matrix A_{2D}, which approximates the Laplacian operator via finite differences. Dirichlet boundary conditions are applied, with $m = 0$ on the boundaries, ensuring that the door's edges remain fixed. The source term $S(x, y)$ is defined by the same Gaussian pulse described in Equation 10.
Next, we solve the linear system $A_{2D} \cdot m = S$ to compute the bending moment $m(x, y)$. Without access to VenturA's construction data, we use open-source parameters. The moment of inertia I is calculated using the formula [7]:

$$I = \frac{bh^3}{12},$$

Where b is the width and h is the height of the cross-section. The Young's modulus E, which varies depending on the material, is obtained from [8].

Together, these give the flexural rigidity $EI(x, y)$, which varies across the structure depending on its material properties:

- **Frame**: Typically made from a stiffer material, the frame has a higher $EI(x, y)$, resulting in greater resistance to bending.
- **Glass**: Being thinner and less stiff, the glass has a lower $EI(x, y)$, reflecting its reduced resistance to deformation.

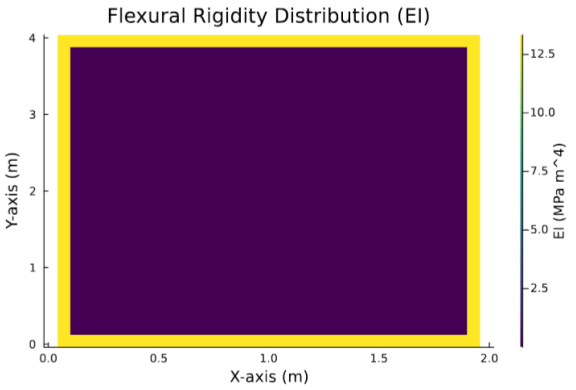


Figure 14: Flexural rigidity of the door

Finally, by solving the second equation, we compute the displacement field $u(x, y)$, taking into account the varying material properties and geometric characteristics of the structure.

2. Fourth-order problem

In this method, we directly solve the stationary biharmonic equation in its fourth-order form:

$$-A_{2D}\left(EI(x, y) \cdot A_{2D}u(x, y)\right) = S(x, y) \tag{16}$$

First, Dirichlet boundary conditions are applied at the domain boundaries to represent a fixed structure. Once the rigidity is computed, the matrix A_{2D} is adjusted by multiplying each row by the corresponding $EI(x, y)$ to account for material variations. Finally, the biharmonic equation is solved by inverting the system $A_{2D}(EI \cdot A_{2D})u = S$, yielding the displacement field $u(x, y)$, which illustrates the deformation of the structure under the applied load.

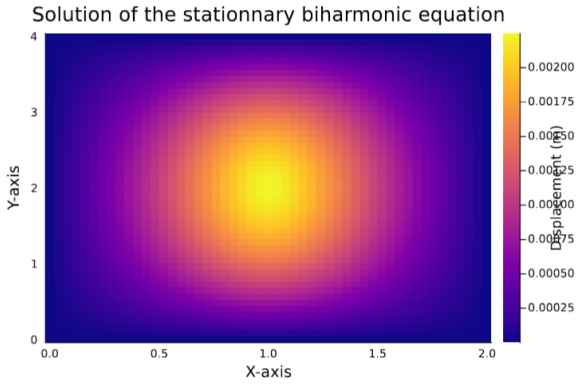


Figure 15: Stationary Biharmonic Solution for $N = 50$

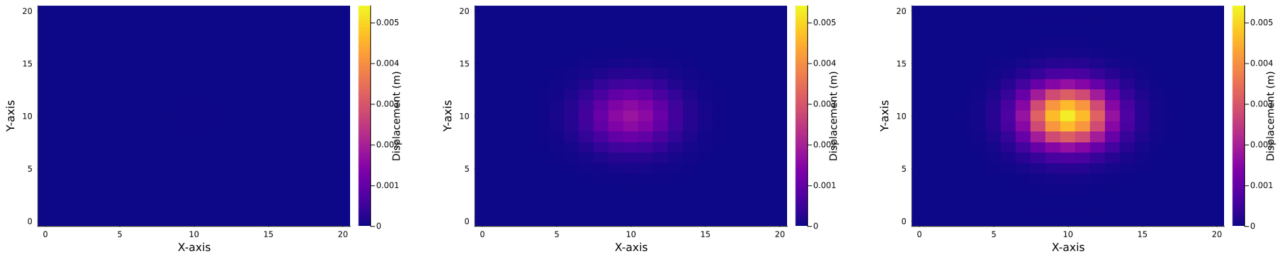
The consistency of results obtained from both methods strengthens our confidence that the code is functioning correctly.

5.3 Time-Dependent Biharmonic Equation in 2D

We now focus on 11, reformulating it as a system of two first-order ODEs:

$$\begin{cases} \frac{\partial u}{\partial t} = v, \\ \frac{\partial v}{\partial t} = \nabla^2\left(EI(x, y) \times \nabla^2 u\right) - S(x, y), \end{cases} \tag{17}$$

To handle the second-order time derivative, we introduce the auxiliary variable $v = \frac{\partial u}{\partial t}$, which represents the velocity. This reformulation allows us to decouple the original second-order equation into a system of first-order equations, simplifying both the numerical integration and solution process.



(a) At $t = 0.0$s (b) At $t = 0.005$s (c) At $t = 0.01$s

Figure 16: Solution of the 2D time-dependent biharmonic equation

We apply Euler's implicit method, as we did for the diffusion equation, to iteratively solve this system at each time step. As we said, this method is highly effective for maintaining stability, particularly in stiff systems like the biharmonic equation, enabling us to use larger time steps without sacrificing accuracy.

Given that the implicit Euler integration worked well for the diffusion equation—validated against an analytical solution—this enhances our confidence in its reliability for the biharmonic problem.

Since the stationary version of the solution behaves as expected, we can conclude that the spatial discretization is likely accurate. This, combined with the successful handling of the temporal aspect, suggests that the method provides a valid and robust approach to solving the time-dependent biharmonic equation.

6 General Conclusion

During my internship at TU Delft, I focused on the mechanical modeling and numerical simulation of bus door systems subjected to various stress conditions. This project allowed me to apply advanced numerical techniques, particularly in solving partial differential equations such as the biharmonic and diffusion equations. These methods offered valuable insights into complex structural behaviors and optimization strategies, demonstrating the significance of mathematical rigor in addressing real-world engineering challenges.

By progressing from single-mass systems to more sophisticated, time-dependent models, I developed robust simulation tools capable of predicting how bus doors respond to stress. The success of these models validated their applicability and also could be useful for future exploration in structural mechanics, particularly in optimizing door designs to reduce weight while preserving strength and durability.

Looking ahead, my upcoming coursework, including Fatigue and Fracture and Industrial Structural Design and Optimization, will build on the foundation established during this internship. I believe the fatigue course will deepen my understanding of how repetitive stresses, similar to those experienced by bus doors, contribute to material fatigue and eventual failure. Gaining knowledge about the thresholds of structural integrity under real-world conditions will be crucial for future design enhancements. Additionally, I hope the course on structural optimization will allow me to adopt a more comprehensive approach to designing components that meet strength requirements while optimizing for cost and material efficiency. These courses align with my growing interest in sustainable engineering and my commitment to developing durable, energy-efficient systems.

In conclusion, this internship has not only honed my technical skills but also broadened my perspective on the critical role sustainability plays in modern engineering.

A Bibliography

References

[1] P. Edwards. Mass-spring-damper systems: The theory. Bournemouth University, 2001. Available at https://faculty.washington.edu/seattle/physics227/reading/reading-3b.pdf.

[2] Wikipedia contributors. Loi de hooke. https://fr.wikipedia.org/wiki/Loi_de_Hooke, 2023. Accessed: 2023-09-23.

[3] Henry C. Fu and Kam K. Leang. Teaching the difference between stiffness and damping. University of Utah, 2012. Available at https://hfu.mech.utah.edu/wp-content/uploads/sites/23/2016/06/FuHC_2012a-Teaching-the-difference-between-stiffness-and-damping.pdf.

[4] Marc Buffat. équations de diffusion. Université Claude Bernard Lyon 1, 2014. https://perso.univ-lyon1.fr/marc.buffat/COURS/COURSDF_HTML/node9.html.

[5] Wikipedia contributors. Biharmonic equation. https://en.wikipedia.org/wiki/Biharmonic_equation, 2023. Accessed: 2023-09-23.

[6] JN Reddy. Existence and uniqueness of solutions to a stationary finite element model of the biharmonic equation. Computers & Mathematics with Applications, 3(2):135–147, 1977.

[7] Area Moment of Inertia. Engineering Toolbox. https://www.engineeringtoolbox.com/area-moment-inertia-d_1328.html.

[8] Young's Modulus of Steel. Amesweb. https://amesweb.info/Materials/Youngs-Modulus-of-Steel.aspx.

B Environmental Initiatives and TU Delft's Commitment

B.1 Introduction

Environmental concerns are at the forefront of today's global challenges, and the transportation industry is no exception. Working at TU Delft, renowned for its cutting-edge research and innovation, I have had the opportunity to explore how our efforts can contribute to a more sustainable future.

It is remarkable how universities can play a pivotal role in driving the ecological transition. At TU Delft, the commitment to sustainability goes beyond mere rhetoric and translates into tangible actions. The labs are buzzing with activity as researchers and students collaborate to rethink materials, optimize processes, and reduce carbon footprints.

In the context of our collaboration with Ventura, a leader in bus door manufacturing, we are exploring how technological advancements can address environmental challenges. For instance, using recyclable or low-impact materials in door design could not only reduce waste but also improve vehicle energy efficiency by lightening structures.

B.2 The Role of TU Delft in Promoting Sustainability

Being immersed in the TU Delft environment has shown me firsthand the university's dedication to environmental issues. Researchers are not just focused on theoretical aspects but are deeply engaged in practical solutions that can be implemented in the industry.

One of the most inspiring aspects is the interdisciplinary approach adopted here. Engineers work alongside environmental scientists, economists, and social scientists to develop better solutions. This collaborative spirit ensures that environmental strategies are not only technically sound but also economically viable and socially acceptable.

B.3 Integration with Bus Door Manufacturing

The Ventura project is a good example of this interdisciplinary approach. They are not only focused on making bus doors more efficient, but they are also reimagining them in the context of sustainability. This involves several key initiatives:

- **Material Innovation**: Exploring the use of new, sustainable materials that are lighter and have a smaller environmental footprint.
- **Energy Efficiency**: Redesigning door mechanisms to reduce energy consumption during operation, contributing to the overall efficiency of the bus.
- **Lifecycle Assessment**: Evaluating the environmental impact of the doors from production to end-of-life, ensuring that each stage aligns with sustainability goals.

These initiatives are not without challenges. Introducing new materials or processes often raises complex questions about cost, technical feasibility, and market acceptance.

B.4 Project Management Principles

Critically analyzing investments, assessing full and partial costs, and calculating break-even points are steps in an approach that ensures new environmental projects are not only good for the planet but also economically viable.

For example, when considering a new recyclable material for the door frames, a thorough cost-benefit analysis must be conducted. This includes not only the initial production costs but also potential savings from reduced material waste and improved energy efficiency over the product's lifetime.

B.5 Overcoming Challenges Through Dynamic Project Management

It is important to acknowledge that adopting new materials or processes can face resistance, whether technical, financial, or cultural. This is where dynamic project management comes into play. By staying flexible and adapting our methods, we can overcome these hurdles. Employing agile methodologies allows us to test solutions quickly, learn from failures, and adjust our course accordingly.

For instance, when an initial prototype using a new material did not meet the durability standards, it was not seen as a setback. Instead, it was an opportunity to refine the approach, leading to a more robust solution in subsequent iterations.

B.6 The Human Dimension

Perhaps the most critical insight has been recognizing the importance of the human element. The changes we are striving for depend on the willingness of individuals—engineers designing the products, technicians manufacturing them, and customers using them. Cultivating a culture of innovation and environmental responsibility is therefore paramount.

B.7 Conclusion

In summary, a commitment to the environment is a complex process requiring a blend of technical expertise, economic rigor, and human understanding. By integrating these dimensions, we can make meaningful contributions to building a more sustainable world.

My internship at TU Delft has not only allowed me to apply theoretical knowledge in a practical setting but also to become more aware of the work driving the future of sustainable transportation.

C Julia Code

Listing 1: Parameters Setup

```
using DifferentialEquations
using Plots
using Statistics
using SymPy
using SpecialFunctions
using LinearAlgebra
using SparseArrays
using Printf
using LinearSolve

# Parameters
m = 75.0 # Mass (kg)
w0 = 2 *  * 1.5 # Resonant frequency (rad/s)
k = m * w0^2 # Spring constant
gamma = 0.1 # Damping coefficient
F0 = 1.0 # Amplitude of the impulse (N)
t0 = 0.5 # Time of impact
sigma = 0.18 # Width of Gaussian pulse
epsilon = 0.01 # Width of the Dirac delta approximation
Nx = 10
Ny = 10
h = 1.0 / Nx # Assuming equal spacing for simplicity
hx = h
hy = h;
D = 1.0 # Diffusion coefficient, adjust as needed
delta_t = 0.0001 # Time step size
num_steps = 100; # Number of time steps
k =  / Lx # Wave number for the x-direction
ell =  / Ly ; # Wave number for the y-direction
```

C.1 Single Mass-Spring System Simulations

Listing 2: First Numerical Method

```

# Define the Gaussian pulse which approximates a Dirac delta function
function gaussian_pulse(t, t0, F0, sigma)
    return F0 * exp(-((t - t0)^2) / (2 * sigma^2))
end

# Time interval for plotting the Dirac delta approximation
t = 0:0.001:1

# Calculate the Dirac delta approximation
dirac_approx = [gaussian_pulse(ti, t0, F0, epsilon) for ti in t]

# Plot the Dirac delta approximation
p0 = plot(t, dirac_approx, label="Dirac␣Delta␣Approximation", xlabel="Time␣(s)", ylabel="
    Amplitude␣(N)", title="Dirac␣Delta␣Approximation␣by␣a␣Gaussian␣Pulse")

# Define the system of equations
function mass_spring_damper!(ddu, du, u, p, t)
    ddu[1] = (gaussian_pulse(t, t0, F0, sigma) - gamma * du[1] - k * u[1]) / m
end
```

```
# Initial conditions
u0 = [0.0] # Initial position
v0 = [0.0] # Initial velocity
tspan = (0.0, 20.0) # Time span

# Problem definition
prob = SecondOrderODEProblem(mass_spring_damper!, v0, u0, tspan)

# Solve the problem
sol = solve(prob)

# Plot the solution for position
p1 = plot(sol, idxs=2, label="Position␣of␣the␣mass", xlabel="Time␣(s)", ylabel="Position␣(m)
    ", title="Position") # Here idxs = 2 is the position

# Combine the plots
display(plot(p0, p1, layout = (2, 1)))

# Save the Dirac delta approximation plot
savefig(p0, "dirac_delta_approximation.png");

# Save the position plot
savefig(p1, "mass_position.png");
```

Listing 3: Second Numerical Method

```
# Define the system of equations
function mass_spring_damper!(du, u, p, t)
    F_external = gaussian_pulse(t, t0, F0, sigma)
    du[1] = u[2]
    du[2] = (F_external - gamma * u[2] - k * u[1]) / m
end

# Initial conditions
u0 = [0.0, 0.0] # Initial position and velocity

# Problem definition
prob_one_ODE = ODEProblem(mass_spring_damper!, u0, tspan)

# Solve the problem using the default solver (explicit method Tsit5)
sol_one_ODE = solve(prob_one_ODE, Tsit5())

# Plot the solution for position and velocity
p2 = plot(sol_one_ODE, idxs=1, label="Position", xlabel="Time␣(s)", ylabel="Position␣(m)",
    title="Position")

# Display the plot
display(p2)
```

Listing 4: Comparison Between Analytical and Numerical Solutions

```
# Analytical solution for the damped case
function analytical_solution_damped(t, t0, F0, m, , 0)
     =  / (2m)
     = sqrt(4m*k - ^2) / (2m)
    u = similar(t)
    for i in 1:length(t)
        if t[i] >= t0
            u[i] = (F0 / (m * ^2)) * exp(- * (t[i] - t0)) * sin( * (t[i] - t0))
```

```
        else
            u[i] = 0.0
        end
    end
    return u
end

# Calculate the analytical solutions
t = sol.t
u_analytical_damped = analytical_solution_damped(t, t0, F0, m, gamma, w0)

# Plot the numerical and analytical solutions together for position
p4 = plot(t, sol[2,:], label="Position (numerical - 2nd order ODE)", xlabel="Time (s)",
    ylabel="Position (m)", title="Position: Numerical vs Analytical")
plot!(p4, t, u_analytical_damped, label="Position (analytical)")

# Display the plots
display(p4)

# Calculate the errors
absolute_error = abs.(sol[2,:] .- u_analytical_damped)
relative_error = abs.((sol[2,:] .- u_analytical_damped) ./ u_analytical_damped)

# Plot the errors
p66 = plot(t, log10.(absolute_error), label="Log10 of Absolute Error", xlabel="Time (s)",
    ylabel="Log10(Absolute Error)", title="Error Evolution with Tolerance")
plot!(p66, t, log10.(relative_error), label="Log10 of Relative Error")

# Display the plot
display(p66)

# Save the position plot
savefig(p66, "error_tolerance");
```

C.2 Two Point Mass System

Listing 5: Code to Calculate and Plot the Analytical Solution

```
# Define symbols
@syms  k m1 m2  t F0

# Define symbolic matrix A
A = sympy.Matrix( 4, 4,
    [
    0, 0, 1, 0,
    0, 0, 0, 1,
    -(2*k / m1), k / m1, 0, 0,
    k / m2, -2*k / m2, 0, 0
])

# Calculate symbolic eigenvalues of A
 = A.eigenvals()
_values = collect(keys())
display()

V = A.eigenvects()
V_matrix = []
```

```
# Iterate over each tuple (eigenvalue, list of eigenvectors)
for item in V
    eigenvalue, _, eigenvectors_list = item
    for eigenvector_matrix in eigenvectors_list
        # Each 'eigenvector_matrix' contains an eigenvector as a matrix
        # Extract the eigenvector, which is the only element in the list
        push!(V_matrix, eigenvector_matrix)
    end
end

# Concatenate all eigenvectors into columns to form matrix V
V_matrix = hcat(V_matrix...)
display(V_matrix)

# Define the cosine function as the external force
function cosine_force(t, , F0)
    return F0 * cos( * t)
end

# Create the force vector F
F = [0; 0; cosine_force(t, , F0)/m1; 0]

# Transform A and F into modal coordinates
V_inv = V_matrix.inv()
A_modal = V_inv * A * V_matrix
F_modal = V_inv * F

# Solve the system in each modal coordinate, including the force term
q_solutions = []
q = SymFunction("q") # Define a symbolic function q for modal coordinates

for i in 1:length(_values)
    # Set up the differential equation for each modal coordinate
    eq = sympy.Eq(sympy.Derivative(q(t), t, t) + _values[i] * q(t), F_modal[i])

    # Solve the differential equation with specified initial conditions
    ics = Dict(q(0) => 0, q(t).diff(t).subs(t, 0) => 0)
    println("Starting equation resolution for i = ", i)
    sol = dsolve(eq, q(t), ics=ics)
    println("Equation solved for i = ", i)
    push!(q_solutions, sol)
end

# Convert q(t) solutions to a column matrix
q_solutions_matrix = [sol.rhs for sol in q_solutions] # Extract the right-hand side (rhs) of
    each equation
q_solutions_matrix = sympy.Matrix(q_solutions_matrix) # Convert to a SymPy matrix

# Display the solutions for modal coordinates
display(q_solutions_matrix)

# Multiply by the eigenvector matrix to obtain z(t)
z_solutions = V_matrix * q_solutions_matrix

# Extract the displacements u1(t) and u2(t)
u1_t = z_solutions[1, :] # Assume that u1(t) is on the first row
u2_t = z_solutions[2, :] # Assume that u2(t) is on the second row

display(u1_t)
```

```
display(u2_t)

# Parameters
m1_val = 75.0 # Mass of object 1 (in kg)
m2_val = 85.0 # Mass of object 2 (in kg)
_val = 2.0 # Resonant frequency for mass 1
k_val = m1_val * _val^2 # Spring constant k derived from mass and frequency
F0_val = 5 # Amplitude of the external force

# Substitute the parameters into the symbolic expressions
u1_t_subs = u1_t.subs([(m1, m1_val), (m2, m2_val), (k, k_val), (F0, F0_val), (, _val)])
u2_t_subs = u2_t.subs([(m1, m1_val), (m2, m2_val), (k, k_val), (F0, F0_val), (, _val)])

# Evaluate the expressions over the time interval
t_values = 0:0.011:10 # Time interval from 0 to 10 with a step of 0.011
u1_values = [float(subs(u1_t_subs[i], t => val)) for val in t_values, i in 1:size(u1_t_subs,
    1)] # Compute u1(t) for each t in t_values
u2_values = [float(subs(u2_t_subs[i], t => val)) for val in t_values, i in 1:size(u2_t_subs,
    1)] # Compute u2(t) for each t in t_values

# Extract the real part of the computed values
u1_values_real = [real(u1) for u1 in u1_values]
u2_values_real = [real(u2) for u2 in u2_values]

# Plot the results
plot(t_values, u1_values_real, label="x1 (analytical)", title="Displacement u1(t) and u2(t)"
    , xlabel="Time (t)", ylabel="Displacement")
plot!(t_values, u2_values_real, label="u2 (analytical)")

# Save the plot to a file
savefig("displacement_masses.png")
```

Listing 6: Numerical Solution with No Dampers

```
# Define the first-order ODE system
function system!(du, u, p, t)
    x1, v1, x2, v2 = u # positions and velocities
    du[1] = v1
    du[2] = (gaussian_pulse(t, t0_val, F0_val, sigma_val) - k_val*x1 - k_val*(x1 - x2)) /
        m1_val
    du[3] = v2
    du[4] = (0.0 - k_val*x2 - k_val*(x2 - x1)) / m2_val
end

# Initial conditions: zero initial positions and velocities
u0 = [0.0, 0.0, 0.0, 0.0]

# Time span for the simulation
tspan = (0.0, 10.0)

# Define the first-order ODE problem
prob = ODEProblem(system!, u0, tspan)

# Solve the problem using Tsit5 solver with lower tolerances
sol = solve(prob, Tsit5(), reltol=1e-8, abstol=1e-10) # Lowered tolerances

# Extract the numerical solutions
u1_num = sol[1, :]
u2_num = sol[3, :]
```

```
# Plot the results of the displacements
plot(sol.t, u1_num, label="x1 (numerical)", xlabel="Time", ylabel="Displacement", title="
    Displacement of Masses Over Time")
display(plot!(sol.t, u2_num, label="x2 (numerical)"))

# Save the plot to a file
savefig("displacement_masses.png")
```

Listing 7: Comparison Between Single and Two-Mass Systems

```
# First order ODE for the single mass system
function mass_spring_damper!(du, u, p, t)
    du[1] = u[2]
    du[2] = (gaussian_pulse(t, t0, F0, sigma) - gamma * u[2] - k * u[1]) / m
end

# Initial conditions
u0_single = [0.0, 0.0] # Initial position and velocity
tspan_single = (0.0, 5.0)

# Problem definition for the single mass
prob_single = ODEProblem(mass_spring_damper!, u0_single, tspan_single)

# Solve the single mass problem with tolerance
sol_single = solve(prob_single, Tsit5(), reltol=1e-12, abstol=1e-12)

# Adjusted parameters for two mass system
m1 = 75.0 # Mass 1 (kg)
m2 = 75.0 # Mass 2 (set equal to m1 to mimic single mass system)
k1 = k # Spring constant for mass 1
k2 = 500.0 # Moderate spring constant to couple the two masses
k3 = 0.0 # No spring constant for the second spring
1 = gamma # Damping coefficient for mass 1
2 = 0.0 # No damping for mass 2
3 = 0.0 # No damping for the second spring

# Function for external force on mass 2 (no external force here)
F_external2(t) = 0.0

# First order ODE for two mass system
function system!(du, u, p, t)
    x1, v1, x2, v2 = u # positions and velocities

    du[1] = v1
    du[2] = (gaussian_pulse(t, t0, F0, sigma) - 1*v1 - 2*(v1 - v2) - k1*x1 - k2*(x1 - x2)) /
        m1
    du[3] = v2
    du[4] = (F_external2(t) - 3*v2 - 2*(v2 - v1) - k3*x2 - k2*(x2 - x1)) / m2
end

# Initial conditions: zero initial positions and velocities
u0_two = [0.0, 0.0, 0.0, 0.0]
tspan_two = (0.0, 5.0)

# Problem definition for two masses
prob_two = ODEProblem(system!, u0_two, tspan_two)

# Solve the two mass problem with tolerance
sol_two = solve(prob_two, Tsit5(), reltol=1e-12, abstol=1e-12)
```

```
# Plot the comparison
p_comparison = plot(sol_single.t, sol_single[1,:], label="Single Mass (with tolerance)",
    xlabel="Time (s)", ylabel="Displacement (m)", title="Comparison: Single vs Two Mass
    System")
plot!(sol_two.t, sol_two[1,:], label="Mass 1 (Two Mass System with tolerance)", linestyle=:
    dash)
plot!(sol_two.t, sol_two[3,:], label="Mass 2 (Two Mass System with tolerance)", linestyle=:
    dot)

# Display the plot
display(p_comparison)

# Save the plot
savefig("comparison_single_two_masses.png")
```

Listing 8: Numerical Study with Dampers

```
# Define parameters
m1 = 75.0 # Mass 1 (between 50 and 100 kg)
m2 = 85.0 # Mass 2 (between 50 and 100 kg)
01 = 2.0 # Resonance frequency for mass 1
02 = 2.5 # Resonance frequency for mass 2

k1 = m1 * 01^2 # Spring constant k1
k3 = m2 * 02^2 # Spring constant k3
k2 = 1500.0 # Spring constant k2 (chosen to create coupling between the masses)

1 = 20.0 # Damping coefficient 1
2 = 25.0 # Damping coefficient 2
3 = 30.0 # Damping coefficient 3

# Function for external force on mass 2 (no external force here)
F_external2(t) = 0.0

# Definition of the first-order ODE system
function system!(du, u, p, t)
    x1, v1, x2, v2 = u # positions and velocities

    du[1] = v1
    du[2] = (gaussian_pulse(t, t0, F0, sigma) - 1*v1 - 2*(v1 - v2) - k1*x1 - k2*(x1 - x2)) /
        m1
    du[3] = v2
    du[4] = (F_external2(t) - 3*v2 - 2*(v2 - v1) - k3*x2 - k2*(x2 - x1)) / m2
end

# Initial conditions: zero initial positions and velocities
u0 = [0.0, 0.0, 0.0, 0.0]

# Time interval for plotting the Dirac delta approximation
t = 0:0.001:10

# Calculate the Dirac delta approximation
dirac_approx = [gaussian_pulse(ti, t0, F0, epsilon) for ti in t]

# Time span for the simulation
tspan = (0.0, 10.0)

# Plot the Dirac delta approximation
```

```
p0 = plot(t, dirac_approx, label="Dirac Delta Approximation", xlabel="Time (s)", ylabel="
    Amplitude", title="Dirac Delta Approximation by a Gaussian Pulse")

# Define the first-order ODE problem
prob = ODEProblem(system!, u0, tspan)

# Solve the problem using the Tsit5() method
sol = solve(prob, Tsit5())

# Plot the results of the displacements
p1 = plot(sol.t, sol[1,:], label="u1 (displacement of mass 1)", xlabel="Time", ylabel="
    Displacement", title="Displacement of Masses Over Time")
plot!(sol.t, sol[3,:], label="u2 (displacement of mass 2)")

plot(p0, p1, layout = (2, 1))
```

Listing 9: Numerical Stiffness Study

```
function system!(du, u, p, t)
    m1, m2, 01, 02, k2, 1, 2, 3, t0, F0, sigma = p
    k1 = m1 * 01^2
    k3 = m2 * 02^2
    x1, v1, x2, v2 = u

    du[1] = v1
    du[2] = (gaussian_pulse(t, t0, F0, sigma) - 1*v1 - 2*(v1 - v2) - k1*x1 - k2*(x1 - x2)) /
         m1
    du[3] = v2
    du[4] = (F_external2(t) - 3*v2 - 2*(v2 - v1) - k3*x2 - k2*(x2 - x1)) / m2
end

# Simulate and plot for different masses, and calculate delta t
function simulate_mass_system(mass_cases)
    01 = 2.0
    02 = 2.5
    k2 = 1500.0
    1 = 20.0
    2 = 25.0
    3 = 30.0
    t0 = 5.0
    F0 = 5
    sigma = 0.18
    tspan = (0.0, 10.0)
    u0 = [0.0, 0.0, 0.0, 0.0]

    plots = []
    for (i, (m1, m2)) in enumerate(mass_cases)
        p = [m1, m2, 01, 02, k2, 1, 2, 3, t0, F0, sigma]
        prob = ODEProblem(system!, u0, tspan, p)
        sol = solve(prob, Tsit5())

        # Create plot for each scenario with proper spacing
        plt = plot(sol.t, sol[1, :], label="x1 (m1 = $m1 kg)", xlabel="Time (s)", ylabel="
            Displacement (m)", title="Scenario $i")
        plot!(plt, sol.t, sol[3, :], label="x2 (m2 = $m2 kg)")
        push!(plots, plt)

        # Calculate delta t for each solution
        delta_ts = diff(sol.t)
        println("Delta t for m1 = $m1 kg and m2 = $m2 kg: ", mean(delta_ts))
```

```
    end

    # Adjust layout to avoid overlap and space plots
    p_combined = plot(plots..., layout = grid(length(mass_cases), 1), size=(500, 500),
        titlefont=9, labelfont=5)
    return p_combined
end

# Define mass cases for simulation
mass_cases = [(50.0, 50.0), (50.0, 100.0), (50.0, 500.0)]

# Call the simulation function and display the plot
p_combined = simulate_mass_system(mass_cases)
display(p_combined)

# Save the final plot with titles
savefig(p_combined, "mass_system_displacement_scenarios.png")
```

C.3 Diffusion Equation

C.3.1 1D Diffusion Equation

Listing 10: Resolution of the 1D Diffusion Equation

```
# Mesh parameters
N = 100 # Number of interior points
L = 10.0 # Length of the domain
h = L / N # Mesh spacing

# Grid point positions
x = range(0, L, length=N+1)

# Boundary conditions
T_a = 100.0
T_b = 0.0

# Construct the sparse matrix A using Tridiagonal
A = Tridiagonal(1/h^2 * ones(N), -2/h^2 * ones(N+1), 1/h^2 * ones(N))

# Apply boundary conditions directly
A[1, 1] = 1.0
A[N+1, N+1] = 1.0
A[1, 2] = 0.0
A[N+1, N] = 0.0

println("This␣is␣the␣A␣matrix:")
display(A)

# ------------------------------------------------------------------------
# Construct the source vector S
# Define the function for the Gaussian pulse
function gaussian_pulse(x, p)
    Amp, x0, sigma = p
    return Amp * exp(-0.5 * ((x - x0) / sigma)^2)
end

# Parameters of the pulse
Amp = 100 # Amplitude of the impulse
```

```
x0 = L / 2 # Center of the Gaussian impulse
sigma = L / 10 # Standard deviation of the impulse

# Preparation of the parameter vector
p = [Amp, x0, sigma]

# Constructing the source vector S using broadcasting
S = gaussian_pulse.(x, Ref(p))

# Apply boundary conditions
S[1] = T_a
S[N+1] = T_b

# LU decomposition (sparse matrix)
lu_decomp = lu(A)

# Solving the system using the LU factors
T = lu_decomp \ S

# Visualization of the solution
display(plot(x, T, title="Solution to the 1D Diffusion Equation for N = $N", xlabel="x (m)",
     ylabel="T(x) (C)", legend=false))

savefig("1D_Diffusion.png")
```

C.3.2 2D Stationnary Diffusion Equation

Listing 11: Laplacian Matrix Construction

```
# Create the 1D diffusion matrix without any boundary conditions
function create_1D_diffusion_matrix(N, h)
    A_1D = spzeros(N+1, N+1)
    for i in 2:N
        A_1D[i, i-1] = 1 / h^2
        A_1D[i, i] = -2 / h^2
        A_1D[i, i+1] = 1 / h^2
    end
    return A_1D
end

# Create A_1D for both dimensions
A_1Dx = create_1D_diffusion_matrix(Nx, h)
A_1Dy = create_1D_diffusion_matrix(Ny, h)

# Create Axx and Ayy using Kronecker products
Axx = kron(A_1Dx, sparse(I, Ny+1, Ny+1))
Ayy = kron(sparse(I, Nx+1, Nx+1), A_1Dy)

# Combine Axx and Ayy to form the full 2D diffusion matrix A
A = Axx + Ayy

# Display the matrix A
println("Matrix A_1Dy:")
display(A_1Dy)
println("Matrix Axx:")
display(Axx)
println("Matrix Ayy:")
display(Ayy)
```

```
println("Matrix A without boundary conditions:")
display(A)

# Implement boundary conditions

# Construct the mesh indicator matrix IG
IG = ones(Int, Nx+1, Ny+1)
IG[2:end-1, 2:end-1] .= 0 # Interior points
IGvec = reshape(IG, (Nx+1)*(Ny+1)) # Vectorize

# Linear indices for interior and boundary nodes
L = LinearIndices(IGvec)
interior_indices = findall(x -> x == 0, IGvec)
boundary_indices = findall(x -> x == 1, IGvec)

# Handle Dirichlet boundary conditions in the matrix and the right-hand side vector
A[boundary_indices, :] .= 0.0
A[boundary_indices, boundary_indices] .= I(length(boundary_indices))

println("Final matrix A:")
display(A)
```

Listing 12: Resolution Without Gaussian Pulse

```
# Define the boundary conditions
T_north = 100.0 # Example temperature for the north boundary
T_south = 50.0 # Example temperature for the south boundary
T_west = 75.0 # Example temperature for the west boundary
T_east = 25.0 # Example temperature for the east boundary

# Create the right-hand side vector
S = zeros(Nx+1, Ny+1)

# North and South boundaries
for i in 1:Nx+1
    S[i, Ny+1] = T_north
    S[i, 1] = T_south
end

# West and East boundaries
for j in 1:Ny+1
    S[1, j] = T_west
    S[Nx+1, j] = T_east
end

# Reshape F into a vector
s = reshape(S, (Nx+1)*(Ny+1))

# Handle Dirichlet boundary conditions in the matrix and the right-hand side vector
s[boundary_indices] .= S[boundary_indices]

println("Right-hand side vector f:")
display(s)

# Solve the linear system
u = A \ s

# Reshape the solution vector into a 2D array for plotting
U = reshape(u, Nx+1, Ny+1)
```

```
# Manually create the mesh grid using broadcasting
x = 0:h:Nx*h
y = 0:h:Ny*h
x_grid = repeat(x', Ny+1, 1)
y_grid = repeat(y, 1, Nx+1)

# Create a 2D heatmap plot
p2d = heatmap(x, y, U,
              title = "Solution to the 2D Diffusion Equation ",
              xlabel = "x(m)",
              ylabel = "y(m)",
              color = :plasma) # Choose a colormap
              legend="Temperature (C)"

savefig(p2d, "temperature_distribution_2D.png")

# Display the 2D plot
display(p2d)
```

Listing 13: Resolution with Gaussian Pulse

```
# Initialize the source vector F
F = zeros(Nx+1, Ny+1) # Zero matrix initially

# Parameters for the Gaussian pulse
x0, y0 = 0.5 * Nx, 0.5 * Ny # Center of the Gaussian
Amp = 100 # Amplitude of the Gaussian
sigma = 1.0 # Spread of the Gaussian

# Use broadcasting to fill the source vector F with the Gaussian function
x = (0:Nx) * h
y = (0:Ny) * h
F .= Amp .* exp.(-((x .- x0).^2 .+ (y' .- y0).^2) ./ (2 * sigma^2))

# Apply boundary conditions for the north and south
F[:, Ny+1] .= T_north
F[:, 1] .= T_south

# Apply boundary conditions for the west and east
F[1, :] .= T_west
F[Nx+1, :] .= T_east

# Reshape F into a vector
f = reshape(-F, (Nx+1)*(Ny+1))

# Handle Dirichlet boundary conditions in the matrix and the right-hand side vector
IG = ones(Int, Nx+1, Ny+1)
IG[2:end-1, 2:end-1] .= 0 # Interior points
IGvec = reshape(IG, (Nx+1)*(Ny+1)) # Vectorize

boundary_indices = findall(x -> x == 1, IGvec)

A[boundary_indices, :] .= 0.0
A[boundary_indices, boundary_indices] .= I(length(boundary_indices))
f[boundary_indices] .= F[boundary_indices]

# Solve the linear system
u = A \ f

# Reshape the solution vector into a 2D array for plotting
```

```
U = reshape(u, Nx+1, Ny+1)

# Manually create the mesh grid using broadcasting
x_grid = repeat(x', Ny+1, 1)
y_grid = repeat(y, 1, Nx+1)

# Reshape the source vector F for 3D plotting
F_2D = reshape(F, Nx+1, Ny+1)

# Create a 3D surface plot for the source vector F
p1 = surface(x_grid, y_grid, F_2D,
             title = "Source Vector F",
             xlabel = "x",
             ylabel = "y",
             zlabel = "Source Value",
             color = :plasma, # Choose a colormap
             camera = (40, 30)) # Adjust the camera angle for better viewing

savefig(p1, "source_vector_F_3D.png")

# Display the 3D plot
display(p1)

# Create a 2D heatmap plot for the temperature distribution
p2d = heatmap(x, y, U,
              title = "Temperature Distribution for N=$N",
              xlabel = "x",
              ylabel = "y",
              color = :plasma, # Choose a colormap
              colorbar = true) # Show colorbar

# Manually add a vertical colorbar label using `colorbar_title`
plot!(p2d, colorbar_title = "T(x,y) (C)")

savefig(p2d, "temperature_distribution_2D.png")

# Display the 2D plot
display(p2d)
```

Listing 14: Comparison Between Analytical and Numerical Solutions

```
# KNOWN SOLUTION u(x, y) = sin(kx) sin(ell y)

# Compute the known solution
u_known = [sin(k * xi) * sin(ell * yi) for yi in y, xi in x]

# VISUALIZATION

# Manually create the mesh grid using broadcasting
x_grid = repeat(x', Ny+1, 1)
y_grid = repeat(y, 1, Nx+1)

# 3D Plot for the computed solution
p1 = surface(x_grid, y_grid, u_reshaped,
             title = "Computed Solution - N=$Nx",
             xlabel = "x",
             ylabel = "y",
             zlabel = "Displacement (m)",
             color = :plasma,
```

```
            camera = (40, 30))

savefig(p1, "computed_solution_3D.png")
display(p1)

# 2D Heatmap for the known solution
p2 = heatmap(x, y, u_known',
            title="Known Solution u(x,y) - N=$Nx",
            xlabel="X-axis", ylabel="Y-axis",
            color=:plasma, colorbar_title="Displacement (m)")

savefig(p2, "known_solution_2D_10.png")
display(p2)

# 2D Heatmap for the computed solution
p3 = heatmap(x, y, u_reshaped',
            title="Computed Solution from Diffusion Equation - N=$Nx",
            xlabel="X-axis", ylabel="Y-axis",
            color=:plasma, colorbar_title="Displacement (m)")

savefig(p3, "computed_solution_2D_10.png")
display(p3)

# Compare computed solution with the known solution
difference = u_reshaped - u_known
p4 = heatmap(x, y, difference',
            title="Difference between Computed and Known Solutions- N =$Nx",
            xlabel="X-axis", ylabel="Y-axis",
            color=:viridis, colorbar_title="Difference")

savefig(p4, "solution_difference_2D_10.png")
display(p4)
```

C.3.3 Time-Dependent 2D Diffusion Equation

Listing 15: Method 1 : Time-Dependent 2D Diffusion Equation

```
# TIME DEPENDENCY

# Initial condition as a zero vector
T0 = zeros((Nx+1)*(Ny+1))

# Reshape G to match the correct size
G_reshaped = reshape(G, (Nx+1)*(Ny+1))

# Store the solutions
sol = [T0]

# Time loop to solve the equation at each time step
for n in 1:num_steps
    # Compute the new solution T^{n+1}
    T_new = system_matrix \ (sol[end] + delta_t * G_reshaped)
    push!(sol, T_new)
end

# Convert solutions to 2D format for each time step
sol_reshaped = [reshape(T*1000, Nx+1, Ny+1) for T in sol]
```

```
# Calculate the color scale limits over the entire solution
min_temp = minimum([minimum(T) for T in sol_reshaped])
max_temp = maximum([maximum(T) for T in sol_reshaped])

# Create an animation of the solution
anim = @animate for i in 1:length(sol_reshaped)
    frame = sol_reshaped[i]
    heatmap(x, y, frame', title="Step␣$i", xlabel="X-axis", ylabel="Y-axis", color=:viridis,
        clims=(min_temp, max_temp))
end

# Display the animation in the notebook
display(anim)

# Optional: Save the animation as a GIF
gif(anim, "diffusion_animation.gif", fps=10)
```

Listing 16: Euler Explicit - Resolution for Time-Dependent 2D Diffusion Equation

```
# TIME DEPENDENCY

# Initial conditions
T0 = zeros((Nx+1)*(Ny+1))

# Define the source term S
S_reshaped = reshape(S, (Nx+1)*(Ny+1))

# Define the rhs function F(u)
function F!(du, u, p, t)
    du .= D * (A * u) + S_reshaped
end

# Setup the problem
prob = ODEProblem(F!, T0, (0.0, delta_t*num_steps))

# Solve the problem using Euler's method (an explicit solver)
sol = solve(prob, Euler(), dt=delta_t)

# Convert solutions to 2D format for each time step
sol_reshaped = [reshape(T*10000, Nx+1, Ny+1) for T in sol.u] # Note: sol.u contains the
    solution vectors

# Calculate the color scale limits over the entire solution
min_temp = minimum([minimum(T) for T in sol_reshaped])
max_temp = maximum([maximum(T) for T in sol_reshaped])

# Create an animation of the solution
anim = @animate for i in 1:length(sol_reshaped)
    frame = sol_reshaped[i]
    heatmap(x, y, frame', title="Step␣$i", xlabel="X-axis", ylabel="Y-axis", color=:viridis,
        clims=(min_temp, max_temp))
end

# Display the animation in the notebook
display(anim)

# Optional: Save the animation as a GIF
gif(anim, "diffusion_animation.gif", fps=10)
```

Listing 17: Comparison between Analytical and Numerical Solutions

```
# Create the grid
x = range(0.0, stop=Lx, length=Nx+1)
y = range(0.0, stop=Ly, length=Ny+1)

# INITIALIZE THE NUMERICAL SOLUTION TO MATCH THE ANALYTICAL SOLUTION AT t=0

T0 = [sin(k * xi) * sin(ell * yi) * cos(0) for yi in y, xi in x] # Solution at t=0
T0 = reshape(T0, (Nx+1)*(Ny+1)) # Flatten for the solver

# TIME DEPENDENCY

# Define the rhs function F(u) with time-dependent source term
function F!(du, u, p, t)
    f_t = f_source(x, y, t, k, ell, )
    du .= D * (A * u) + f_t
end

# SOLVE THE SYSTEM IN TIME

# Setup the time grid
times = range(0, stop=delta_t * num_steps, length=num_steps+1)

# Setup the problem
prob = ODEProblem(F!, T0, (0.0, delta_t * num_steps))

# Solve the problem using Euler's method (explicit time-stepping)
sol = solve(prob, Euler(), dt=delta_t)

# KNOWN ANALYTICAL SOLUTION

# Define the known analytical solution with time dependence
function u_analytical_solution(x, y, t, k, ell, , D)
    return [sin(k * xi) * sin(ell * yi) * exp(-D * (k^2 + ell^2) * t) for yi in y, xi in x]
end

# Convert solutions to 2D format for each time step
sol_reshaped = [reshape(T, Nx+1, Ny+1) for T in sol.u]

# Calculate the color scale limits over the entire solution
min_temp = minimum([minimum(T) for T in sol_reshaped])
max_temp = maximum([maximum(T) for T in sol_reshaped])

# Calculate the absolute difference for each time step
differences = [abs.(sol_reshaped[i] .- u_analytical_solution(x, y, times[i], k, ell, , D))
    for i in 1:length(sol_reshaped)]

# Calculate the minimum and maximum error values
min_error = minimum([minimum(diff) for diff in differences])
max_error = maximum([maximum(diff) for diff in differences])

# ANIMATION

# Create an animation comparing computed and analytical solutions
anim = @animate for i in 1:length(sol_reshaped)
    t_current = times[i]

    # Compute the analytical solution at the current time step
    u_analytical = u_analytical_solution(x, y, t_current, k, ell, , D)
```

```
    # Compare the numerical and analytical solutions
    frame_computed = sol_reshaped[i]

    # Calculate the difference between numerical and analytical solutions
    difference = frame_computed - u_analytical

    # Visualization for the numerical solution
    p_computed = heatmap(x, y, frame_computed',
                         title="Computed Solution at t=$(round(t_current, digits=6))s",
                         xlabel="X-axis", ylabel="Y-axis", color=:plasma,
                         clims=(min_temp, max_temp))

    # Visualization for the analytical solution
    p_analytical = heatmap(x, y, u_analytical',
                           title="Analytical Solution at t=$(round(t_current, digits=6))s",
                           xlabel="X-axis", ylabel="Y-axis", color=:plasma,
                           clims=(min_temp, max_temp))

    # Visualization for the difference with fixed error scale
    p_difference = heatmap(x, y, difference',
                           title="Difference between Computed and Analytical",
                           xlabel="X-axis", ylabel="Y-axis", color=:viridis,
                           clims=(min_error, max_error)) # Fixed error scale

    # Combine all three plots in one display
    plot(p_computed, p_analytical, p_difference, layout=(3, 1)) # Display one below the
        other
end

# Display the animation in the notebook
display(anim)

# Save the animation as a GIF (optional)
gif(anim, "biharmonic_time_dependent_comparison_with_fixed_error_scale.gif", fps=10)
```

Listing 18: Comparison between Analytical and Implicit Euler Solutions

```
# Setup the time grid
times = range(0, stop=delta_t * num_steps, length=num_steps+1)

# Setup the problem
prob = ODEProblem(F!, T0, (0.0, delta_t * num_steps))

# Solve the problem using Implicit Euler method (TRBDF2)
sol = solve(prob, TRBDF2(), dt=delta_t, adaptive=false, abstol=1e-12, reltol=1e-12)

# Convert solutions to 2D format for each time step
sol_reshaped = [reshape(T, Nx+1, Ny+1) for T in sol.u]

# Calculate the color scale limits over the entire solution
min_temp = minimum([minimum(T) for T in sol_reshaped])
max_temp = maximum([maximum(T) for T in sol_reshaped])

# Calculate the difference for each time step
differences = [sol_reshaped[i] .- u_analytical_solution(x, y, times[i], k, ell, , D) for i
    in 1:length(sol_reshaped)]

# Calculate the minimum and maximum error values
min_error = minimum([minimum(diff) for diff in differences])
```

```
max_error = maximum([maximum(diff) for diff in differences])

# SELECT SPECIFIC TIME STEPS

time_intervals = [delta_t * 10, delta_t * 50, delta_t * 100] # Display for three specific
    time steps
indices = [findfirst(t -> t  time_interval, times) for time_interval in time_intervals]

# PLOT ERRORS AT SELECTED TIME STEPS

# Create the error plots for the specific time steps
for idx in indices
    t_current = times[idx]

    # Compute the analytical solution at the current time step
    u_analytical = u_analytical_solution(x, y, t_current, k, ell, , D)

    # Compare the numerical and analytical solutions
    frame_computed = sol_reshaped[idx]

    # Calculate the difference between numerical and analytical solutions
    difference = frame_computed - u_analytical

    # Plot the error for the current time step
    p_difference = heatmap(x, y, difference',
                           title="Error at t=$(round(t_current, digits=6))s",
                           xlabel="X-axis", ylabel="Y-axis", color=:viridis,
                           clims=(min_error, max_error),
                           colorbar_ticks=range(min_error, max_error, length=10))

    # Display the plot
    display(p_difference)
end
```

C.4 Biharmonic Equation

C.4.1 Stationnary Biharmonic Equation

Listing 19: Flexural Rigidity Calculation

```
# Frame and glass dimensions (example values)
b_frame = 0.1 # Width of the frame section (in meters)
h_frame = 0.2 # Height of the frame section (in meters)
b_glass = 0.01 # Width of the glass section (in meters)
h_glass = 0.01 # Height of the glass section (in meters)

# Material properties
E_frame = 2.0e11 # Young's modulus of the frame (Pa)
E_glass = 7.0e10 # Young's modulus of the glass (Pa)

# Calculate the moment of inertia for the frame and glass using the rectangular section
    formula
I_frame = (b_frame * h_frame^3) / 12 # Moment of inertia for the frame (m^4)
I_glass = (b_glass * h_glass^3) / 12 # Moment of inertia for the glass (m^4)

# Create the grid
x = range(0, stop=Lx, length=Nx+1)
y = range(0, stop=Ly, length=Ny+1)
```

```
# Initialize E(x,y) and I(x,y) matrices
E_matrix = fill(E_glass, Nx+1, Ny+1)
I_matrix = fill(I_glass, Nx+1, Ny+1)

# Define the frame region
t_c = 0.1 # Thickness of the frame (in meters)
for j in 1:Ny+1
    for i in 1:Nx+1
        if x[i] <= t_c || x[i] >= Lx - t_c || y[j] <= t_c || y[j] >= Ly - t_c
            E_matrix[i, j] = E_frame
            I_matrix[i, j] = I_frame
        end
    end
end

# Compute EI(x,y) matrix
EI_matrix = E_matrix .* I_matrix

# Convert E_matrix from Pa to MPa
E_matrix_MPa = E_matrix ./ 1.0e6 # Convert from Pa to MPa

# Recompute EI(x,y) with E in MPa
EI_matrix_MPa = E_matrix_MPa .* I_matrix # Still in (MPa * m^4)

# Visualize the EI(x,y) distribution in MPa * m^4
heatmap(x, y, EI_matrix_MPa', title="Flexural Rigidity Distribution (EI)", xlabel="X-axis (m
    )", ylabel="Y-axis (m)", color=:viridis, colorbar_title="EI (MPa m^4)")
```

Listing 20: Biharmonic 2D Stationary Method 1

```
# LAPLACIAN MATRIX

# Construct the Laplacian matrix A
function create_1D_diffusion_matrix(N, h)
    A_1D = spzeros(N+1, N+1)
    for i in 2:N
        A_1D[i, i-1] = 1 / h^2
        A_1D[i, i] = -2 / h^2
        A_1D[i, i+1] = 1 / h^2
    end
    return A_1D
end

A_1Dx = create_1D_diffusion_matrix(Nx, hx)
A_1Dy = create_1D_diffusion_matrix(Ny, hy)

Axx = kron(A_1Dx, sparse(I, Ny+1, Ny+1))
Ayy = kron(sparse(I, Nx+1, Nx+1), A_1Dy)

A = Axx + Ayy # The full 2D Laplacian matrix

# Handle Dirichlet boundary conditions (v = 0 on the boundaries)
IG = ones(Int, Nx+1, Ny+1)
IG[2:end-1, 2:end-1] .= 0 # Interior points
IGvec = reshape(IG, (Nx+1)*(Ny+1)) # Vectorize

boundary_indices = findall(x -> x == 1, IGvec)
```

```
A[boundary_indices, :] .= 0.0
A[boundary_indices, boundary_indices] .= I(length(boundary_indices))

# MULTIPLYING EI(x,y) WITH A

# We will multiply each row of A by the corresponding EI value
EI_vector = reshape(EI_matrix, (Nx+1)*(Ny+1)) # Flatten the EI matrix

# Element-wise multiplication of each row of A by EI_vector
EI_A = sparse(Diagonal(EI_vector)) * A # This creates the product EI(x,y) * A

# EI_A is now the matrix representing the action of EI(x,y) on the Laplacian matrix


# SOLVING THE SYSTEM WITH EI(x,y) \nabla^2 u = v

# Assume v_reshaped is already computed from a previous step or given as input
m_reshaped = reshape(m_reshaped, (Nx+1)*(Ny+1))

# Use the updated EI_A matrix from the previous step
linear_problem2 = LinearProblem(EI_A, m_reshaped)

# Solve the linear system
solver_solution2 = solve(linear_problem2)

# Extract the solution
u = solver_solution2.u

# Reshape u back to 2D for visualization
u_reshaped = reshape(u, Nx+1, Ny+1)

# Visualization of the final displacement field u(x,y)
heatmap(x, y, u_reshaped', title="Solution of the stationnary biharmonic equation", xlabel="
    X-axis", ylabel="Y-axis", color=:plasma, colorbar_title="Displacement (m)")
```

Listing 21: Biharmonic 2D Stationary Method - Version 2

```
# FLEXURAL RIGIDITY EI(x,y)

# Frame and glass dimensions (example values)
b_frame = 0.1 # Width of the frame section (in meters)
h_frame = 0.2 # Height of the frame section (in meters)
b_glass = 0.01 # Width of the glass section (in meters)
h_glass = 0.01 # Height of the glass section (in meters)

# Material properties
E_frame = 2.0e11 # Young's modulus of the frame (Pa)
E_glass = 7.0e10 # Young's modulus of the glass (Pa)

# Calculate the moment of inertia for the frame and glass using the rectangular section
    formula
I_frame = (b_frame * h_frame^3) / 12 # Moment of inertia for the frame (m^4)
I_glass = (b_glass * h_glass^3) / 12 # Moment of inertia for the glass (m^4)

# Create the grid
x = range(0, stop=Lx, length=Nx+1)
y = range(0, stop=Ly, length=Ny+1)

# Initialize E(x,y) and I(x,y) matrices
E_matrix = fill(E_glass, Nx+1, Ny+1)
```

```
I_matrix = fill(I_glass, Nx+1, Ny+1)

# Define the frame region
t_c = 0.1 # Thickness of the frame (in meters)
for j in 1:Ny+1
    for i in 1:Nx+1
        if x[i] <= t_c || x[i] >= Lx - t_c || y[j] <= t_c || y[j] >= Ly - t_c
            E_matrix[i, j] = E_frame
            I_matrix[i, j] = I_frame
        end
    end
end

# Compute EI(x,y) matrix
EI_matrix = E_matrix .* I_matrix

# Flatten the EI matrix for multiplication with the Laplacian operator
EI_vector = reshape(EI_matrix, (Nx+1)*(Ny+1))

# MULTIPLYING EI(x,y) WITH A (Fourth-Order Operator)

# Element-wise multiplication of each row of A by EI_vector
EI_A = sparse(Diagonal(EI_vector)) * A # This creates the product EI(x,y) * A

# SOURCE TERM

# Create the 2D grid of points using comprehensions to simulate meshgrid
x = range(0, stop=Nx, length=Nx+1)
y = range(0, stop=Ny, length=Ny+1)

# Define the Gaussian pulse function in 2D
function gaussian_pulse(x, y, p)
    Amp, x0, y0, sigma = p
    return Amp * exp(-((x - x0)^2 + (y - y0)^2) / (2 * sigma^2))
end

# Parameters for the Gaussian pulse
Amp = 100
x0, y0 = Nx / 2, Ny / 2 # Center the pulse in the middle of the grid
sigma = Nx / 10 # Adjust sigma to scale with grid size

p = [Amp, x0, y0, sigma]

# Generate the Gaussian pulse matrix in 2D
S = [gaussian_pulse(xi, yi, p) for yi in y, xi in x]

# Reshape S into a vector
S_reshaped = reshape(S, (Nx+1)*(Ny+1))

# SOLVING THE SYSTEM A(EI A)u = S + du/dt

# Use the updated EI_A matrix for the biharmonic equation
A_biharmonic = A * EI_A # This is A(EI A)

# Solve the linear system A_biharmonic * u = S_reshaped
u = A_biharmonic \ S_reshaped

# Reshape u back to 2D for visualization
u_reshaped = reshape(u, Nx+1, Ny+1)
```

```
# Visualization of the final displacement field u(x,y)
heatmap(x, y, u_reshaped', title="Solution of the Stationary Biharmonic Equation", xlabel="X
    -axis", ylabel="Y-axis", color=:plasma, colorbar_title="Displacement (m)")
```

C.4.2 Time-Dependent 2D Biharmonic Equation

Listing 22: Code for Time-Dependent 2D Biharmonic Equation Solution

```
# SOLVING THE SYSTEM WITH SECOND-ORDER TIME DERIVATIVES

# Define the function for the system of first-order ODEs
function F!(du, u, p, t)
    # Use @view to access slices without copying the data
    displacement = @view u[1:(Nx+1)*(Ny+1)] # View for displacement
    velocity = @view u[(Nx+1)*(Ny+1)+1:end] # View for velocity

    # First equation: u/t = v
    du_displacement = @view du[1:(Nx+1)*(Ny+1)]
    du_displacement .= velocity

    # Second equation: v/t = (EI  u) - S
    du_velocity = @view du[(Nx+1)*(Ny+1)+1:end]
    du_velocity .= A * (EI_A * displacement) - S_reshaped
end

# Initial conditions: zero displacement and velocity
u0 = zeros(2 * (Nx+1) * (Ny+1)) # [displacement; velocity]

# Define the time span and time-stepping parameters
delta_t = 0.00001 # Time step size
num_steps = 100 # Number of time steps

tspan = (0.0, delta_t * num_steps)

# Setup the ODE problem with second-order time derivatives
prob = ODEProblem(F!, u0, tspan)

# Solve the system using implicit Euler method
sol = solve(prob, ImplicitEuler(), dt=delta_t)

# VISUALIZATION OF THE RESULTS

# Extract displacement from the solution and convert to 2D format
sol_reshaped = [reshape(u[1:(Nx+1)*(Ny+1)] * (-100), Nx+1, Ny+1) for u in sol.u] # Extract
    displacement

# Calculate the color scale limits over the entire solution
min_dep = minimum([minimum(u) for u in sol_reshaped])
max_dep = maximum([maximum(u) for u in sol_reshaped])

# Animation for the time evolution of the displacement field
anim = @animate for i in 1:length(sol_reshaped)
    current_time = delta_t * i # Calculate time in seconds for each frame
    heatmap(x, y, sol_reshaped[i]',
        title="Time = $num_steps",
        xlabel="X-axis",
```

```
        ylabel="Y-axis",
        color=:plasma,
        clims=(min_dep, max_dep),
        colorbar_title="Displacement␣(m)" # Add vertical color bar label
    )
end

# Display the animation
display(anim)

# Optional: Save the animation as a GIF
gif(anim, "biharmonic_time_dependent.gif", fps=10)
```