Distributed Partial Differential Equation Solving with Julia Fast Fourier Transform Library

Christopher E. Mottola
University of New Hampshire, Durham

Follow this and additional works at: https://scholars.unh.edu/honors

Part of the Numerical Analysis and Scientific Computing Commons

Recommended Citation
https://scholars.unh.edu/honors/694

This Senior Honors Thesis is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars’ Repository. It has been accepted for inclusion in Honors Theses and Capstones by an authorized administrator of University of New Hampshire Scholars’ Repository. For more information, please contact Scholarly.Communication@unh.edu.
Distributed Partial Differential Equation Solving with Julia Fast Fourier Transform Library

Christopher Mottola

December 15, 2022
Abstract

Scientific computing relies on advanced computational and mathematical techniques to solve complex problems in scientific domains. For the numerical rendering of spectral, nonlinear, and dynamic phenomena, there is a growing need for greater availability of a broad class of Fourier-based algorithms to perform large scale operations on multidimensional data in distributed and optimized ways. To this effect, the Julia programming language is new and has significant advantages compared to other common languages used in scientific computing. The research presented here formulates a basis for further development in high-performance scientific computing of periodic partial differential equations through the application of distributed Fast Fourier Transforms in Julia with the PencilFFTs.jl library.
# Table of contents

1 Background ................................................................. 3  
   1.1 Introduction ........................................................ 3  
   1.2 Motivations .......................................................... 3  
   1.3 Research Questions ............................................... 4  

2 Applied Mathematics Review ........................................ 5  
   2.1 Partial Differential Equations .................................. 5  
   2.2 Spectral Analysis ................................................... 6  
      2.2.1 Fourier Series ............................................. 6  
   2.3 Fourier Transforms ............................................... 8  

3 Development Methods .................................................. 13  
   3.1 The Julia Language ................................................ 13  
   3.2 Parallel and Distributed Programming ............................ 13  
      3.2.1 Message-Passing Interface ................................ 14  
   3.3 PencilFFTs.jl Package ........................................... 14  
      3.3.1 MPI Topology ............................................. 15  
      3.3.2 Configuring Pencil Array Data ............................ 16  
      3.3.3 Transforms, Plans, and Views ............................ 16  
      3.3.4 Partial Differentiation to Gradient Space .............. 16  

4 Results ................................................................. 17  
   4.1 Applications and Extensibility .................................. 18  
   4.2 Concluding remarks .............................................. 18  

References ............................................................... 19  

Appendices .............................................................. 20
Background

1.1 Introduction

Scientific computing allows us to experiment on natural systems in a scientifically robust way, such as to investigate the pattern formation and time-evolution of natural and measurable phenomena (heat, air, and fluid flow, geophysics, climate modeling, molecular chemistry, circuit components, population dynamics, economics, etc.). The goal of scientific computing is to resolve systems of equations and to simulate them over time, which is computationally intensive. Of particular interest in scientific computing research is addressing techniques for solving partial differential equations (PDEs), which are mathematical systems that enforce specific relationships between variables of a multivariable function. The focus of this research was to investigate the methodologies involved in solving periodic (wavelike) PDEs using the Fast Fourier Transform (FFT), and to develop usage code for an FFT library in the Julia programming language.

1.2 Motivations

Scientific computing frameworks typically require a high degree of computational efficiency and performance due to the intensive workloads of the large scale operations and data sets involved. There is a performance tradeoff inherent in most programming languages between the ease of use, flexibility, and runtime optimization of program execution. For example, C and Fortran, two legacy languages, are arguably the two most computationally efficient (in terms of memory allocation per workload), yet they are also two of the most difficult languages to program in, and require years to master. Domain experts thus tend to rely on slower dynamic languages for daily computational
work, such as Python, because of its simplicity, adaptability, and vast library support. However, Python is limited in its ability to scale to larger architectures and workloads. Modern language design and compiler techniques have made it possible to eliminate most of these performance trade-offs inherent in older programming languages, and the Julia language fulfills this role.

It is of a high degree of importance to continue finding new and improved ways of solving complex systems and simulating them over time for scientific investigation. By exploring and expanding on the development of Fourier-based algorithms into the growing Julia language, the research here distills the basic ingredients for interacting with multidimensional and distributed data through Fourier modes with the PencilFFTs.jl library. These steps form the basis for future applications of solving periodic partial differential equations in scientific computing.

1.3 Research Questions

I. Understanding pattern formation; investigating why shapes arise and what properties they have, resolving coherent structures in turbulence as dynamical solutions, equilibria to traveling waves, etc.

II. Learning about spectral analysis, Fourier Transform modes, and parallel computing over a Message-Passing Interface (MPI).

III. Exploring support for solving PDEs with FFTs in a Julia programming framework. Reconciling existing Julia libraries to provide a parallelization speedup for multi-dimensional scientific computing.

IV. Developing usage code for the PencilFFTs.jl library and extending the work toward possible parallel PDE-solving implementations.
2.1 Partial Differential Equations

A Partial Differential Equation (PDE) is a mathematical equation that describes how a system evolves according to the function’s partial derivatives. In other words, a PDE expresses the relationships between multiple variables with respect to changes in each other’s values over a defined domain. Different PDEs are typically solved with different techniques, owing to the vast range of possible behavior in nonlinear and multidimensional, differentiable systems. While beyond the scope of this paper, popular techniques often involve the use of separation of variables, change of variables, integral transforms, and finite element analysis.

For mathematical context, a PDE is said to be harmonic if all of its second-derivatives sum to zero. The example below represents the solution to a harmonic PDE in 3-D Cartesian coordinates.

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0
\]

PDEs are fundamental to many mathematically-oriented fields and form the basis for our scientific understanding of physical phenomena, including sound, heat, electromagnetism, thermodynamics, general relativity, and quantum mechanics. PDEs allow us to mathematically represent these systems, numerically simulate them over time with computer modeling, and experiment with their variables to investigate and find patterns in the ways they can evolve. PDEs are wide-ranging and qualitatively distinct in both their mathematical formulations as well the applied and analytical techniques that are involved in solving them. Much research in pure mathematics is dedicated
to defining when analytical solutions are well-posed and existing locally for a system on a domain. However, due to complex nonlinear relationships between variables, many PDEs are not explicitly solvable, i.e., they cannot worked out on paper. Therefore, a large amount of research in applied mathematics and sciences is committed to developing applied numerical techniques to numerically approximate PDE solutions.

2.2 Spectral Analysis

Spectral methods are a class of techniques for numerically solving PDEs by using the sum of a set of basis functions (e.g., Fourier series) to determine the spectral coefficients that best satisfy the rendering of the gradient of a differential equation. The gradient describes how the differentiation of a system is distributed across its variables, i.e., how the variables affect each other as the system evolves, and is key to techniques for numerically solving PDEs. Spectral analysis involves the calculation of periodic oscillations in sets of sequenced data. Periodic behavior means mathematically that certain numerical values are repeated over discrete periods (often measured in cycles of $2\pi$) which can be analyzed as frequency relationships. Datasets are sampled as a function of one or more independent variables (e.g.: $x, y, z, t$); for example, from the spatial and temporal components of traveling waves and surface oscillations. Data points are then discretized over constant intervals, and mathematical techniques are used to transform between time and frequency bases. Such transforms allow for simpler mathematical representations of the data spectra, even if the sampled measurements are not periodic. This leads to more computationally-efficient operations to be performed on them, thus saving memory resources and improving runtime performance for scientific computing.

2.2.1 Fourier Series

The Fourier theorem states that any periodic waveform can be decomposed into a sum of sinusoidal waves, each with different frequencies, amplitudes and phases. When summed back together into a superposition, these component waves reconstitute the original waveform. The accuracy of this reconstitution depends on the sampling rate, frequency intervals, data coherency, and the nature of the functions themselves.
There are numerous ways to represent a Fourier series. The exponential form is shown below in Equation 2a, where $N$ represents the number of elements being summed, and $P$ is the function’s period. Known function values or sampled data points can be entered for $x$. The $c_n$ variables are the spectral coefficients, which are derived from harmonic analysis. The integer index, $n$, stands for the number of cycles that the $n^{th}$ harmonic makes in the function’s period.

$$s_N(x) = \sum_{n=-N}^{N} c_n e^{\frac{2\pi i n x}{P}}$$

(2a)

The harmonics are the related sinusoids that are being summed in the series, each with a wavelength of $\frac{n}{P}$. The Fourier series $s_N$ is taken as a sum of complex waves through integral relationships with Euler’s formula (Fig. 2.1), which allows for symmetries in the collocation of transform coordinates.

![Euler’s Formula Visual Representation](image)

Figure 2.1: Euler’s Formula Visual Representation.
As each harmonic is added in the iteration of a Fourier series expansion, the resultant partial sum converges to the function being decomposed. Convergence of Fourier series means that as more harmonics (sinusoidal iterations) are summed, each successive partial Fourier series sum will better approximate the function, and with an infinite number of harmonics, will map the function exactly. In order for a periodic function to have a convergent Fourier series, it must meet what is known as Dirichlet’s conditions.

### 2.3 Fourier Transforms

The *Fourier Transform* transforms the basis of a function between time and frequency domains. What this means is that the equation can then be inspected and manipulated from a space of higher-order numerics, leading to
more efficient operations (even if the function is not periodic), as well as an enhanced view of the transverse nature of cyclical relationships in the function space. Each component sine wave receives its own frequency, amplitude, and phase parameters, as well as Fourier wave coefficients that distill the amplitude and phase relationship at each wave’s frequency. The Fourier transform is a linear operation, i.e., \( f(x+y) = f(x) + f(y) \) and \( f(cx) = cf(x) \). Typically, a function will be transformed to Fourier space, operated upon to model a change upon the system state, and then transformed back into a time-based representation.

![Figure 2.3: Fourier Time and Frequency Domains.](image)

The Fourier Transform is a powerful mathematical tool when used with spectral method algorithms for solving partial differential equations. An example operation that can performed on a periodic function signal in the frequency domain is to determine its power spectrum. This is accomplished through squaring and summing the Fourier coefficients at each component wave frequency, which describes the distribution of energy at each spectral density composing the waveform in the time segment analyzed.
The equations above outline the integrals used for transforming between bases, with $H$ representing the frequency harmonics, and $h$ the time values space. Equations (1) and (2) use frequency in the classical sense, as in the number of cycles (wave periods) that pass a fixed point along the path per a given time interval; e.g., 1 Hz means 1 cycle per second. Equations (3) and (4) instead use the angular frequency representation, replacing $f$ with an $\omega$, measured in radians per second. The Fourier Transform is also used in quantum physics to transform between position and momentum spaces.

**Discrete Fourier Transform**

The Discrete Fourier Transform (DFT) is used for Fourier analysis in many practical applications, by transforming discrete sampled values of a quantity or signal varying over time, and mapping them into a periodic frequency space over evenly spaced grid points. The DFT replaces the integral of the continuous transform with a summation of evenly spaced samples over a finite time interval (window function). The DFT is thus said to be a frequency domain representation of the original input sequence. The DFT is used to efficiently solve partial differential equations with real-world data, and also to perform other operations, such as function convolutions or large integer multiplications.
Inverse Discrete Fourier Transform

An Inverse Discrete Fourier Transform (IDFT) transforms a discrete function signal from the frequency domain back into the time domain, using the spectral coefficients as collocation points for the discretization of the continuous function being translated. The immediate result should have the same sample-values as the original input sequence, contingent upon the degree of numerical precision being used.

Fast Fourier Transform

For scientific computing problems, the Fast Fourier Transform (FFT) forms the crux for many of the world’s most important mathematical algorithms. The FFT is the primary tool we have for numerically solving partially differentiable equations, particularly for systems that are periodic. FFTs are typically used to compute characteristics of or operations on a scalar or vector field along three spatial dimensions. The FFT is considered one of the most important numerical algorithms we have available.

Figure 2.5: The data decomposition and redistribution steps using symmetries of the Fast Fourier Transform algorithm.

On each layer of the Fast Fourier Transform, the algorithm implementation uses complex bit symmetries to compute one cycle (iteration) of the
Discrete Fourier Transform, and its inverse (backwards, or reverse, transform) produces one cycle of the Inverse Discrete Fourier Transform. Because of the nature of the algorithm, a recursive optimization is reached through complex wave collocation and binary characteristics that is computationally convenient and allows the algorithm to scale indefinitely. This factorization reduces the computational complexity of the discrete Fourier transform from $O(N^2)$ to $O(N\log N)$, where $N$ is the data size. In high-performance computing environments, the massive data transpositions at the end of a distributed FFT algorithm tend to be the most expensive part of the transform, due to the large computational cost of communications between processor nodes. Fortunately, there are tweaks that can be made to the formula for certain spectral domain symmetries and logical representations to further optimize the computation for a given system architecture and operational workload.

**Fastest Fourier Transform of the West**

FFTW is a popular subroutine library written in C for computing the Fourier transform in one or more dimensions, of arbitrary size and of both real and complex data. FFTW was originally developed in 1997 by Matteo Frigo and Steven G. Johnson at the Massachusetts Institute of Technology (MIT) Laboratory for Computer Science. FFTW transforms are included in the PencilFFTs.jl library, and a potential speedup over FFTW methods that PencilFFTs.jl addresses is the need for parallelization over non-serially decomposed dimensions of the distributed transforms.
Development Methods

3.1 The Julia Language

The Julia programming language was designed from the ground up (2009-2012) to resolve issues in performance-tradeoffs from slower dynamic languages for scientific computing. It is high performance and multi-paradigm, and combines many features of other programming languages. Julia has many rich and immersive environments for data visualization as well as for numerous domain-specific ecosystems, thus PDE-solving algorithms can be well-integrated across the sciences.

As a dynamically-typed language, it is built for high-performance scientific computing and is well-equipped to solve a myriad of complex and multi-dimensional systems, with performance close to statically-compiled legacy languages like C and Fortran. The flexible parametric type system and use of multiple dispatch give Julia its ability to efficiently express abstract high-level algorithms decoupled from implementation details, and to then generate specialized code to infer and apply each case at run time. This allows for PDE equations to be solved efficiently on a variety of different architectures. Julia also has an integrated parallel computing interface, which provides built-in primitives for instruction level parallelism, multi-threading, GPU computing, and distributed computing, which further bolsters project scalability.

3.2 Parallel and Distributed Programming

A parallelized program is a program that has been split up into smaller interlocking threads and processes, which are executed simultaneously over independent memory spaces. In parallel computing, parts of a program pro-
cess are spread across different processor cores of the central processing unit (CPU). With distributed computing, these processes can spread across many CPUs, and even extend to multiple computer networks.

### 3.2.1 Message-Passing Interface

To perform parallelized computations in this research, the PencilFFTs.jl package incorporates array decomposition through the message passing interface (MPI), which is a standardized means of exchanging messages between processes of a parallelized program across separate and shared memory spaces. PencilFFTs is well adapted for scalability of up to more than tens of thousands of MPI processes.

The virtual topology of an MPI interface facilitates the synchronization of parallel processes via point-to-point and collective communicator message-passing between nodes and networks. Additional library functionalities and interoperable optimizations can be installed to handle more intricate topologies and distributed operations.

The figure below demonstrates the output of a simple ”Hello world” Julia program being run across four MPI processes, with each parallel process finishing execution at different times.

![Figure 3.1: MPI ”Hello world” example.](image)

### 3.3 PencilFFTs.jl Package

PencilFFTs.jl is a library package for the Julia language that provides multidimensional Fourier Transforms over MPI-distributed arrays. The package gets its name from what is known as a pencil decomposition, a 2D decomposition of a 3D array for performing parallelized FFTs. Using PencilFFTs.jl,
FFTs are performed one dimension at a time along the non-decomposed di-
rection, and transforms are interleaved with global data transpositions to
coordinate between distributed array configurations. The implementation
of PencilFFTs can work efficiently with geometries of arbitrary dimension,
while achieving performance very close to the C FFTW library and Fortran
P3DFFT. [2]

![Figure 3.2: Pencil Decomposition – each uniquely colored block is managed
by a different MPI process.](image)

**3.3.1 MPI Topology**

Using MPI.Init() initializes the Message-Passing Interface (MPI) to synchro-
nize the distribution of a local process’s resources with any other processes
that are being run in parallel as part of the global program.

- Multidimensional data arrays are decomposed along one or more dimen-
sions, so that each MPI process handles one of the resulting portions
of the array.

- Given a decomposition, each process node can perform serial FFTs over
the non-decomposed dimensions.

- To transform along the remaining dimensions, the data then has to be
redistributed (or transposed) across MPI processes.

- The parallelized processes and pencil arrays finally coalesce in the com-
bined transform calculation.
3.3.2 Configuring Pencil Array Data

The PencilArrays package is built into PencilFFTs, and it provides an efficient framework for working with multidimensional Julia arrays over distributed MPI processes. Pencil arrays are created using the MPI communicator object and the geometry of the system you are trying to solve. They can be constructed using either in-place or out-of-place transforms, with the operational syntax varying slightly between them. Memory is allocated according to the transform plan; data is then assigned to the allocated space. To do so, a function can be defined independently over the domain of the MPI grid, and then broadcast to the initial condition of the function space matrix.

3.3.3 Transforms, Plans, and Views

All transforms supported by FFTW.jl are supported in the PencilFFT library, including complex-to-complex, real-to-complex and real-to-real transforms. FFTs and related transforms may be arbitrarily combined along different dimensions. To create a plan for the FFT algorithm implementation, the pencil array and transform settings are passed as parameters to the PencilFFTPlan() function. Other parameters can be passed as well, such as processor dimensions and permutation dimensions. For indexing the data arrays, and performing the transforms, different grid objects and views can be used. There are four main methods for this described in the PencilFFTs.jl package documentation, including global views, explicit global indexing, using local indices, and using broadcasting. Each option provides different alternatives to indexing MPI-distributed data that may be subject to dimension permutations within the transform.

3.3.4 Partial Differentiation to Gradient Space

By multiplying a function in the frequency domain by the imaginary number, $i$, along with an array of its spectral coefficients, one can perform differentiation in Fourier space. In other words, the multiplication operator is transformed into a differentiation operator through Fourier expansion, and by utilizing the subsequent inverse transform after applying spectral coefficients, the gradient of the original function can be obtained. An example of this process can be found in the appendix.
Results

This research project included identifying the primary steps for scientific computing of distributed and multi-dimensional Fourier transforms. It also serves in outlining and distilling the usage of the PencilFFTs.jl Julia package for solving periodic partial-differential equations through Fourier analysis and spectral collocation of the gradient space.

*Fourier Expansion over Distributed Memory:* Fourier expansion is an essential mathematical tool to scientific computing and has extensible significance for many related domains. Utilizing distributed memory to harness Fourier analysis and spectral methods for a promising new language is an important step for scientific computing. Parallel and distributed processing allow for implementations to scale with increasingly larger and more complex computational architectures and networks.

*Differentiation of Functions in Fourier Space:* Differentiation is a fundamental operation in calculus that allows for multivariable functions to be made sense of in terms of explicit and implicit relationships between variables. Performing differentiation through Fourier expansion and applying spectral coefficients to obtain a gradient is of fundamental importance to resolving complex systems in scientific computing, as it allows for many PDE-solving algorithms to be implemented.

*Using PencilFFTs to Solve Partial Differential Equations:* Solving partial differential equations is significant for scientific computing as it enables us to model and experiment with complex mathematical systems, and to make new discoveries for research and development in nearly all empirical domains. Through the distributed decomposition, transformation, and redistribution of multidimensional data, high-performance time-evolution of
periodic partial differential equations is enabled for the broader scientific computing community.

4.1 Applications and Extensibility

This research is extensible for countless dynamical systems and complex solvers, such as resolving the Navier-Stokes equations. Future developments can include encoding popular algorithms for numerical simulations and dynamical systems, signal processing and spectral models, as well as for more domain-specific systems of equations. Particular examples of interest include the calculation of vibrating strings in a musical instrument, simulating electronic circuits, rendering boundary layer flow and frequency emission spectra, resolving autoresonances in nonlinear phase-space, and determining planetary orbits in a solar system.

Another valuable extension of this research would be more comprehensive benchmarking of performance comparisons for given workloads – measuring the timing of parallel transforms and the granularity of the transforms on different systems architectures.

4.2 Concluding remarks

While my stretch goal for this project to develop libraries and implement algorithms for solving many popular partial differential equations was not reached, my primary goals to better understand natural pattern formation, to explore distributed transform algorithms, and to engage with available Julia libraries and Fourier modes was satisfied. Working on this research project has enhanced and enriched my scientific computing education as well as my natural worldview, and I aspire to continue these and other relevant lines of inquiry into future endeavors.
Bibliography


Appendices
A PencilFFTs Example Code - Obtaining the Gradient

This example uses an in-place algorithm to resolve the gradient of a 1-D periodic system. The reader is encouraged to try running this code segment and compare the values output at the end.

```plaintext
using PencilFFTs
using FFTW
using MPI

MPI.Init()
comm = MPI.COMM_WORLD
Nproc = MPI.Comm_size(comm)
rank = MPI.Comm_rank(comm)

proc_dims = (Nproc, ) # performing a 1D decomposition
dims = (42,1,1) # data dimensions (Nx x Ny x Nz)
N = dims[1]; L = 1; # variables for filling in data arrays
xj = (0:N-1)*L/N; # using evenly spaced x coordinates
yj = (0:N-1)*L/N; # using evenly spaced y coordinates

# function to be transformed for spectral differentiation
f = 3*cos.(2π/L*xj) + 2*sin.(2π/L*yj);
# function’s known derivative to check differentiation
df = -6π*sin.(2π/L*xj) + 4π*cos.(2π/L*yj);

transform = Transforms.FFT!()  
plan = PencilFFTPlan(dims, transform, proc_dims, comm)  
println(plan, "\n")  
A = allocate_input(plan)

u_in = first(A) # input data view
u_out = last(A) # output data view
u_in[:,] .= f # apply function to data

plan * A # perform in-place forward transform
k = fftfreq(N)*N; # get spectral coefficients
```
The dependencies used in developing this research project can be substituted with other resources. However, the author considered it valuable to include what was used here as a starting point for beginners who may not know what steps to take in getting setup for development.

*Windows Subsystem for Linux (WSL)* can be installed on a Windows machine by opening Command Prompt or PowerShell in Administrator mode, entering: `wsl --install` and then rebooting the machine after the installation completes.

*Visual Studio Code (VSCode)* can be downloaded for Windows, Linux, and macOS. Once downloaded, open VSCode and install the *Remote Development* extension pack for WSL. Within the “WSL: Ubuntu” remote environment, install the Julia extension for language support and the Jupyter extension for notebook support.

*Julia 1.7.2* can be installed through the WSL terminal with:

```
wget https://julialang-s3.julialang.org/bin/linux/x64/1.7/julia-1.7.2-linux-x86_64.tar.gz
```

Extract (unzip) the Julia installation archive files:

```
tar xzvf julia-1.7.2-linux-x86_64.tar.gz
```

Copy the directory and create a symbolic link to `/bin`:

```
sudo cp -r julia-1.7.2 /opt/
sudo ln -s /opt/julia-1.7.2/bin/julia /usr/local/bin/Julia
```

It is also important to update in the VSCode settings the Executable Path to the Julia executable, so that Julia code can be run within the VSCode environment.

*OpenMPI 4.0.3* From within WSL command line:
sudo apt update
sudo apt install openmpi-bin libopenmpi-dev
sudo apt-get update openmpi
Update the system MPI binary within the working directory and build the Julia MPI package:

\texttt{julia --project -e \'ENV["JULIA_MPI_BINARY"]="system"; using Pkg; Pkg.build("MPI"; verbose=true)\'}

Export the OpenMPI library to the Linux user's start-up file:

\texttt{echo "export JULIA_MPI_PATH=/usr/lib/x86_64-linux-gnu/openmpi" >> ~/.profile}

C Environment


\textit{Jupyter Notebook (.ipynb)}: A notebook interface is a virtual programming
environment that allows for running code more interactively. Executable code segments and formattable markdown text can be embedded into separate sections (cells) of the document, and may be run in any order selected.

D PDE Examples - Heat and Wave Equations

**Heat equation (2D - Cartesian Geometry)**

\[
\frac{\partial u}{\partial t} = \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) \tag{a}
\]

\[u = u(x, y, t) \tag{b}\]

where \(x, y \in \mathbb{R}, \quad t > 0, \quad \text{and } \alpha \text{ is a fixed coefficient.}\)

In this example, the superposition of the system space, \(u\), is a function of variables \(x\) and \(y\), and time \(t\), such that the gradient, \(\frac{du}{dt}\), describes how the system as a whole can evolve over time according to its (partial) derivatives. There are two second-order partial derivatives in this example: \(\frac{\partial^2 u}{\partial x^2}\) and \(\frac{\partial^2 u}{\partial y^2}\), which uphold the relationship between the system state and how it can change along the \(x\) and \(y\) axes (in a Cartesian plane) over time.

An **Initial Boundary Value Problem (IBVP):** is a differential equation that is subject to certain boundary conditions, which constrain the system. The system begins in an initial boundary-value state, and unique solutions can be derived through partial differentiation and spectral analysis.

**Wave equation (3D - Spherical Geometry - IBVP)**

\[
\frac{\partial^2 u}{\partial t^2} = c^2 \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \tag{c}
\]

\[u(t, x, y, z) = \frac{1}{4\pi c} \int \int \int \psi(\epsilon, \eta, \zeta) \frac{\delta(r - ct)}{r} \, d\epsilon d\eta d\zeta \tag{d}\]

\[u(0, x, y, z) = 0, \quad u_t(0, x, y, z) = \psi(x, y, z) \tag{e}\]

\[r^2 = (x - \epsilon)^2 + (y - \eta)^2 + (z - \zeta)^2 \tag{f}\]

where \(x, y, z \in \mathbb{R}, \quad t > 0, \quad \text{and } c \text{ is a fixed coefficient.}\)
The wave equation is constituted by second order differentials in both sides of the equation, where:

- $u$ is the superposition of a family of spherical waves that have a center point at $\epsilon, \eta, \zeta$;
- $r$ represents the radial distance of the wave from that center point;
- $\psi$ is an arbitrary spectral weighting function; and
- $\delta$ is the delta function for the waveform.

For the initial-value problem to become an initial-boundary value problem, further equations are introduced to resolve the system so that its normal modes are harmonic with the defined boundary conditions. A boundary $B$ is defined on a domain $D$ by a non-negative function $\mu$, where $n$ is the unit outward normal to the boundary. The solution $u$ will satisfy:

$$\frac{\partial u}{\partial n} + \mu u = 0 \quad (g)$$

Eigenvalues are represented by $\lambda$, and each eigenfunction $v$ satisfies:

$$\nabla \cdot \nabla v + \lambda v = 0, \quad \text{in } D, \quad \text{and}$$

$$\frac{\partial v}{\partial n} + \mu v = 0, \quad \text{on } B. \quad (i)$$

For spherical boundary conditions in three spatial dimensions, the angular components of the eigenfunctions are a set of periodic orthogonal functions defined on the surface of a sphere. The radial components of the eigenfunctions are Bessel functions of half-integer order. Bessel functions are solutions $y(x)$ of Bessel’s differential equation (see below), that can be integrated over a time or frequency domain with respect to its partial derivatives, current state, and boundary conditions.

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + (x^2 - \alpha^2)y = 0 \quad (j)$$
List of Illustrations

Figures

2.1 Euler’s Formula Visual Representation ......................... 7
2.2 Fourier Series Iteration of Partial Sums for a Square Wave..... 8
2.3 Fourier Time and Frequency Domains ......................... 9
2.4 Fourier Expansion Modes ...................................... 10
2.5 The data decomposition and redistribution steps using symmetries of the Fast Fourier Transform algorithm. ............... 11
3.1 MPI “Hello world” example ..................................... 14
3.2 Pencil Decomposition – each uniquely colored block is managed by a different MPI process. .......................... 15