Solving the Convective-Diffusive Heat Equation with Numerical Methods

GPU Parallel Processing

Grado de Ingeniería Aeroespacial

Madrid, Septiembre de 2024

Índice

0. Introduction and Theoretical framework

The heat equation is a PDE that describes the heat distribution in a region over time. In its most general form, diffusion and convection effects appear, making it a complex problem to simulate numerically. This document aims to address the solution of the heat equation with convective and diffusive terms using different numerical techniques to determine which one achieves greater speed.

In particular, two approaches will be employed: the three-point finite difference method and global interpolation, the latter in two forms, using matrix-vector and matrix-matrix operations. These methods will be implemented on CPU and GPU, evaluating their computational performance in execution times and floating-point operations (flops). The comparative analysis will allow for an evaluation of the speed of each method.

The two-dimensional heat equation, with convection and diffusion terms, is expressed as:

$$\frac{\partial T}{\partial t} + u\frac{\partial T}{\partial x} = \alpha\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2}\right)$$

where $T(x, y, t)$ is the Temperature at point(x, y) at time t, u is the flow velocity in the x-direction, and α is the thermal diffusivity.

A hypothesis was thought about the expected results in performance between writing a code that performs matrix-matrix operations vs matrix-vector operations.

> At first glance, one might expect that matrix-vector multiplication would behave similarly to matrix-matrix multiplication in terms of performance trends, given that both involve the multiplication of matrix elements. However, the relationship between input data and the number of operations is more significant in the matrix-vector case. The ratio of memory accesses to computational operations is higher for matrix-vector multiplication compared to matrix-matrix multiplication. This means that more memory accesses are required for the same amount of CPU work, leading to longer computation times and a decrease in FLOPS (floating-point operations per second). This section delves into why this disparity occurs, explaining the memory bandwidth limitations and their impact on overall computational performance.

0.1. Numerical Methods used

0.1.1. Three-Point Finite Differences

The three-point finite differences are based on the discretization of the heat equation on a rectangular grid. The second-order approximation for the second derivative is expressed as follows:

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}$$

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}$$

This approach is simpler to implement. El código empleado para irá a continuación.

0.1.2. Global Interpolantion

Global interpolation using Lagrange polynomials is a method used to find a polynomial that passes through a set of points. Given $n+1$ distinct points $(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)$, where $y_i = f(x_i)$, the goal is to construct a polynomial $P(x)$ of degree n such that:

$$P(x_i) = y_i \quad \text{for} \quad i = 0, 1, \ldots, n.$$

Lagrange Interpolating Polynomial

The Lagrange interpolating polynomial $P(x)$ is given by:

$$P(x) = \sum_{i=0}^{n} y_i \ell_i(x),$$

where $\ell_i(x)$ are the Lagrange basis polynomials, defined as:

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}.$$

These basis polynomials satisfy two important properties:

- $\ell_i(x_j) = 0$ for $j \neq i$,
- $\ell_i(x_i) = 1$.

Thus, the interpolating polynomial $P(x)$ passes through all the points (x_i, y_i).

Error of Interpolation

The error of the Lagrange interpolating polynomial is given by:

$$f(x) - P(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i),$$

where $\xi \in [x_0, x_n]$ is some point in the interval. This shows that the interpolation's accuracy depends on the function's smoothness and the placement of the points x_i.

Example

For three points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, the Lagrange interpolating polynomial is:

$$P(x) = y_0 \frac{(x - x_1)(x - x_2)}{(x_0 - x_1)(x_0 - x_2)} + y_1 \frac{(x - x_0)(x - x_2)}{(x_1 - x_0)(x_1 - x_2)} + y_2 \frac{(x - x_0)(x - x_1)}{(x_2 - x_0)(x_2 - x_1)}.$$

This is the polynomial of second degree that passes through $(x_0, y_0), (x_1, y_1), (x_2, y_2)$.

0.2. Computational Implementation

All the code is written in Julia in the IDE Visual Studio Code. Julia is a language designed for numerical analysis and high-performance computing(HPC). Its ease of use, combined with its ability to execute code at speeds comparable to low-level languages like C or Fortran, makes it an interesting choice for scientific computing and large-scale simulations.

Figura 1: Julia and VScode logos

CPU Implementation: The CPU implementation was carried out using BLAS (Basic Linear Algebra Subprograms), which is a highly optimized library for performing common linear algebra operations such as matrix multiplication, vector addition, and matrix factorizations. BLAS is widely used in numerical computing for its efficiency and reliability, and its optimized routines are specifically tailored for performance on modern processors. By leveraging BLAS, the CPU implementation benefits from both multi-threading and memory optimizations, ensuring that the computations are performed as efficiently as possible.

GPU Implementation: For the GPU implementation, CUDA (Compute Unified Device Architecture) will be used, which allows for the parallel execution of tasks on NVIDIA GPUs. CUDA is well-suited for highly parallelizable workloads, such as matrix operations, where thousands of cores on the GPU can work simultaneously to accelerate computations. In the context of this project, CUDA might enable performance gains by offloading the computationally intensive parts of the code, such as matrix-matrix multiplications and vector operations, to the GPU. The use of CUDA could allow the simulation to run faster and handle larger datasets more efficiently compared to a CPU-only implementation.

By utilizing both BLAS on the CPU and CUDA on the GPU, we can compare the performance of each architecture and analyze the trade-offs in terms of speed, and computational efficiency. This approach demonstrates how Julia can seamlessly integrate with these powerful computing libraries to optimize numerical algorithms on both CPUs and GPUs.

1. Results and Analysis

The first test was carried out considering only the diffusive term, in a square plate at a certain temperature, where a peak with a higher temperature is introduced. In this case, it was tested by writing it in two ways: using matrix-vector operations (1) and matrix-matrix operations (2).

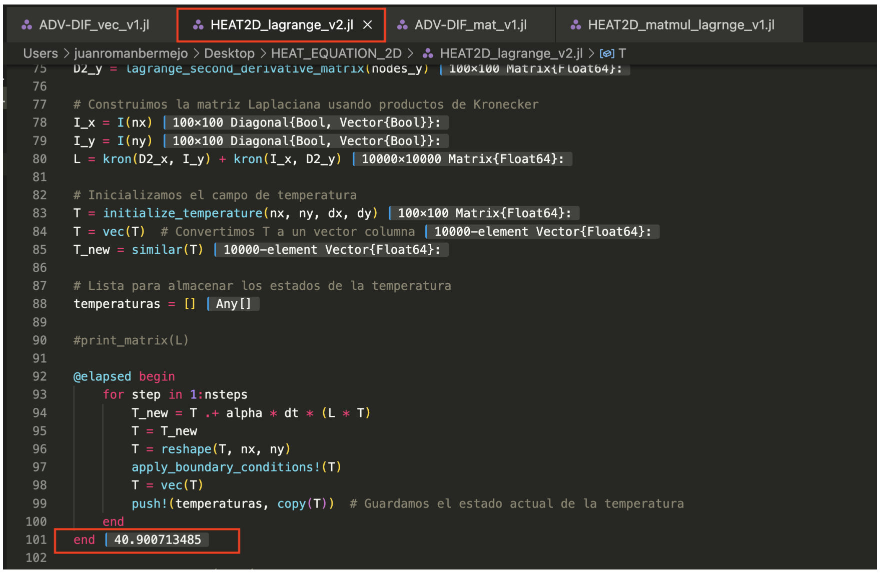

Figura 2: Matrix-vector

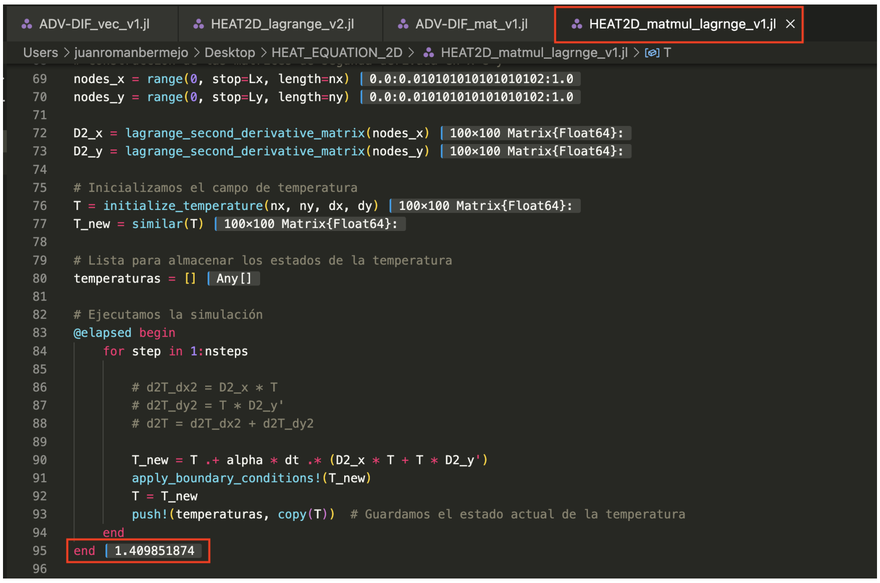

Figura 3: Matriz por Matriz

The results are the following:

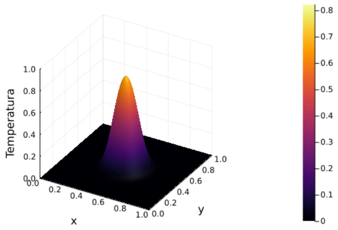

Figura 4: Image 1

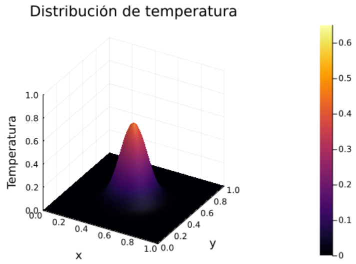

Figura 5: Image 2

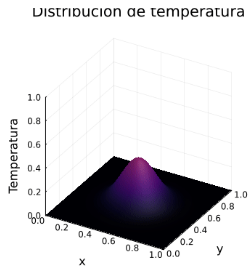

Figura 6: Image 3

After these results, we attempted a more complex simulation, where we observed the evolution of the temperature around a rectangle at a constant temperature T_{const}, immersed in a flow with a different ambient temperature T_∞ and a uniform velocity. The first test is done using finite differences, as shown in the figure below(t=0.416):

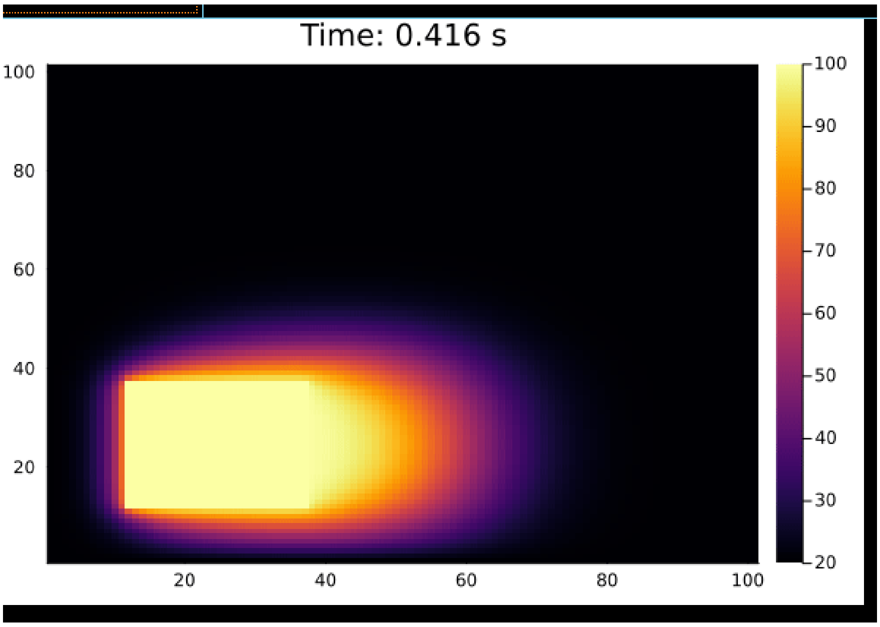

Figura 7: Heat distribution around rectangle (t=0.416)

The code used for this is the following:

```
using Plots
using Printf

# Parameters
nx, ny = 100, 100        # n mero de puntos en x e y
Lx, Ly = 1.0, 1.0        # dimensiones del dominio
dx, dy = Lx/nx, Ly/ny    # espaciado de la malla
alpha = 0.01         # difusividad t rmica
T_inf = 20.0             # temperatura del aire entrante
T_rect = 100.0           # temperatura del rect ngulo
u = 0.1              # velocidad del flujo de aire (izquierda a derecha)

# Esabilidad usando CFL
dt_diff = (dx^2) / (2 * alpha)
dt_conv = dx / u
dt_max = min(dt_diff, dt_conv)
dt = min(dt_max, 0.0001)  # elegir dt estable

println("Paso temporal: $dt")
println("Maximo paso estable: $dt_max")

# N mero de pasos
t_end = 1.0              # tiempo final
nsteps = Int(t_end / dt)

# Inicializaci n del campo de temperatura
T = fill(T_inf, nx+1, ny+1)

#  ndices  del rect ngulo
rect_x = Int(floor(nx/8)):Int(floor(3*nx/8))
rect_y = Int(floor(ny/8)):Int(floor(3*ny/8))
T[rect_x, rect_y] .= T_rect

# Condiciones de contorno
function apply_boundary_conditions!(T, T_inf, u, dt, dx)
    nx, ny = size(T)

    # Inflow (left side)
    T[1, :] .= T_inf

    # Outflow (right side) - COND CONT Convectiva outflow
   # for j in 1:ny
   #     T[end, j] = T[end, j] - u * dt / dx * (T[end, j] - T[end-1, j])
    #end

```

```
    # Arriba
    T[:, end] .= T_inf

    # Abajo
    T[:, 1] .= T_inf
end

# Actualizar el campo de temperatura
function update_temperature!(T, alpha, u, dx, dy, dt)
    nx, ny = size(T)
    T_new = copy(T)
    for i in 2:nx-1
        for j in 2:ny-1
            # Convecci n
            convection = -u * (T[i,j] - T[i-1,j]) / dx
 #upwinding

 # Tx = Dx T  Incluye todas las derivadas (cualquier orden) ver Matvect/ ...
    matmat// para ver comparaci n(tiempo y resultados) u*gradT
            # Difusi n

            # Txx = Dxx T igual evaluar que implementaci n va m s r pido ...
                Dxx la calculas una vez

            diffusion = alpha * (
                (T[i+1,j] - 2*T[i,j] + T[i-1,j]) / dx^2 +
                (T[i,j+1] - 2*T[i,j] + T[i,j-1]) / dy^2 )

            T_new[i,j] = T[i,j] + dt * (convection + diffusion)
        end
    end
    return T_new
end

# Temperatura del rect ngulo
function set_rectangle_temperature!(T, rect_x, rect_y, T_rect)
    T[rect_x, rect_y] .= T_rect
end

# Configuraci n de la visualizaci n
anim = Animation()

# Bucle de integraci n en el tiempo
for step in 1:nsteps
```

```
    # Aplicar condiciones de contorno
    apply_boundary_conditions!(T, T_inf, u, dt, dx)

    # Actualizar el campo de temperatura
    global T = update_temperature!(T, alpha, u, dx, dy, dt)

    # Fijar la temperatura del rect ngulo
    set_rectangle_temperature!(T, rect_x, rect_y, T_rect)

    # Visualizaci n
    if step % 10 == 0
        heatmap(T', c=:inferno, clim=(T_inf, T_rect), title=@sprintf("Time: ...
           %.3f s", step*dt))
        frame(anim)
    end
end

# Guardar la animaci n como GIF
gif(anim, "heat_transfer_simulation1.gif", fps=10)
```

2. Global Interpolation solving

The next step is to solve the simulation using other methods that will allow us to compare implementations that perform matrix-vector operations to matrix-matrix operations. The choice is to solve the equation using a Global Interpolation based on Lagrange polynomials, The Code used is listed below and performs matrix-matrix operations to solve the problem:

```
#using Pkg
#Pkg.add("Plots")
#Pkg.add("Kronecker")
using LinearAlgebra, SparseArrays, Kronecker, BenchmarkTools
using Plots

function print_matrix(m)
    for row in 1:size(m, 1)
        println(join(m[row, :], " "))
    end
end

# Par metros del problema
nx = 25  # N mero de puntos en la direcci n x
ny = 25   # N mero de puntos en la direcci n y
Lx = 1.0  # Longitud del dominio en la direcci n x
Ly = 1.0  # Longitud del dominio en la direcci n y
alpha = 0.001  # Difusividad t rmica
v = [0, 0.01]  # Vector de velocidad (vx, vy)
dt = 0.01  # Paso de tiempo
tfinal = 15  # Tiempo final de la simulaci n
nsteps = Int(tfinal / dt)  # N mero de pasos de tiempo

# Definimos el tama o de la malla
dx = Lx / (nx - 1)
dy = Ly / (ny - 1)

# Inicializaci n continua de temperaturas
function initialize_temperature(nx, ny, dx, dy)
    T = zeros(nx, ny)
    for i in 1:nx
        for j in 1:ny
            x = (i-1) * dx
            y = (j-1) * dy
            T[i,j] = exp(-(25*(x-0.5)^2 + 25*(y-0.5)^2))
            if T[i,j] < 0
                T[i,j] = 0
            end
        end
```

```
    end
    return T
end



# Funci n para aplicar las condiciones de frontera y mantener la fuente ...
    t rmica fija
function apply_boundary_conditions!(T, nx, ny, dx, dy)
    # Mantener la temperatura fija en los bordes
    T[1, :] .= exp(-(25*(-0.5)^2))
    T[end, :] .= exp(-(25*(0.5)^2))
    T[:, 1] .= exp(-(25*(-0.5)^2))
    T[:, end] .= exp(-(25*(0.5)^2))

    # Mantener la fuente t rmica fija en el centro con la condici n inicial
    # for i in 1:nx
    #     for j in 1:ny
    #         x = (i-1) * dx
    #         y = (j-1) * dy
    #         distance = sqrt((x - 0.5)^2 + (y - 0.5)^2)
    #         if distance ≤ 0.05
    #             #T[i,j] = exp(-(25*(x-0.5)^2 + 25*(y-0.5)^2))
    #             T[i,j] = 1
    #         end
    #     end
    # end
end



# Definimos los nodos en x e y
nodes_x = range(0, stop=Lx, length=nx)
nodes_y = range(0, stop=Ly, length=ny)



# ...
    ===========================================================================

# C lculo de los polinomios de Lagrange para los x
function lagrange_basis(x_nodes, i, x)
    l_i = 1.0
    for j in 1:length(x_nodes)
        if j != i
            l_i *= (x - x_nodes[j]) / (x_nodes[i] - x_nodes[j])
        end
    end
```

```
    return l_i
end

# Funci n para calcular la derivada del polinomio de Lagrange
function lagrange_derivative(x_nodes, i, x)
    dl_i = 0.0
    for m in 1:length(x_nodes)
        if m != i
            term = 1.0 / (x_nodes[i] - x_nodes[m])
            for j in 1:length(x_nodes)
                if j != i && j != m
                    term *= (x - x_nodes[j]) / (x_nodes[i] - x_nodes[j])
                end
            end
            dl_i += term
        end
    end
    return dl_i
end

# Funci n para calcular la matriz de derivadas usando polinomios de Lagrange
function lagrange_derivative_matrix(x_nodes)
    n = length(x_nodes)
    D = zeros(n, n)
    for i in 1:n
        for j in 1:n
            D[i, j] = lagrange_derivative(x_nodes, j, x_nodes[i])
        end
    end
    return D
end


# Definimos los nodos en x e y
nodes_x = range(0, stop=Lx, length=nx)
nodes_y = range(0, stop=Ly, length=ny)

# Construcci n de las matrices de derivadas con POLINOMIOS DE LAGRANGE
D_x = lagrange_derivative_matrix(nodes_x)
D_y = lagrange_derivative_matrix(nodes_y)

# Las matrices de segunda derivada son simplemente el producto de la matriz ...
    de primera derivada con ella misma
D2_x = D_x * D_x
D2_y = D_y * D_y

# ...
    ===========================================================================
```

```
# Inicializamos el campo de temperatura
T = initialize_temperature(nx, ny, dx, dy)  # Convertimos T a un vector columna
T_new = similar(T)

# Lista para almacenar los estados de la temperatura
temperaturas = []


# Medir el tiempo del bucle
@elapsed begin
    for step in 1:nsteps

        # difusion = alpha * (Laplacian * T)
        # adveccion = -(Gradient * T)

        T_new = T .+ dt .* (alpha * (D2_x * T + T * D2_y') - (v[1] * (D_x * ...
            T) + v[2] * (T * D_y')))

        global T = T_new
        apply_boundary_conditions!(T, nx, ny, dx, dy)
        push!(temperaturas, copy(T))  # Guardamos el estado actual de la ...
            temperatura
    end
end


## === REPRESENTACI N GR FICA === ##

# Convertimos el resultado a una matriz para mostrarlo
T = reshape(T, nx, ny)

# Determinamos los l mites de los ejes
x = range(0, stop=Lx, length=nx)
y = range(0, stop=Ly, length=ny)
z_min, z_max = 0, 1.0  # L mites del eje z (temperatura)

# # Creamos la animaci n 3D
# intervalo_animacion = 30
# animation = @animate for i in 1:intervalo_animacion:length(temperaturas)
#     t = temperaturas[i]
#     surface(x, y, reshape(t, nx, ny), title="Distribuci n de temperatura ...
    [M][M]", xlabel="x", ylabel="y", zlabel="Temperatura", c=:inferno, ...
    xlims=(0, Lx), ylims=(0, Ly), zlims=(z_min, z_max))
# end
```

```

# # Guardamos la animaci n como un gif
# gif(animation, "adveccion_difusion_diferencias_finitas_matxmat_3d.gif", ...
    fps=30)


# Creamos la animaci n 2D con l neas de contorno sin letras
intervalo_animacion = 30
animation = @animate for i in 1:intervalo_animacion:length(temperaturas)
    t = temperaturas[i]
    contourf(x, y, reshape(t, nx, ny), c=:inferno, xlims=(0, Lx), ylims=(0, ...
        Ly), zlims=(z_min, z_max), xlabel="", ylabel="", title="", ...
        clabels=false)
end

# Guardamos la animaci n como un gif
gif(animation, "adveccion_difusion_contornos2d.gif", fps=10)
```

1. Heat Diffusion

The heat diffusion equation describes how heat spreads through the medium over time due to thermal diffusivity. Mathematically, it is expressed as:

$$\frac{\partial T}{\partial t} = \alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)$$

where:

- $T(x, y, t)$ is the temperature field,
- α is the thermal diffusivity, which determines the rate at which heat diffuses through the material,
- $\frac{\partial^2 T}{\partial x^2}$ and $\frac{\partial^2 T}{\partial y^2}$ are second-order spatial derivatives representing the diffusion of heat in the x and y directions.

In this simulation, the diffusion term is discretized using **Lagrange polynomial interpolation** to approximate the second-order derivatives in both directions.

2. Advection

Advection describes the transport of heat due to fluid motion, which can be represented as:

$$-\mathbf{v} \cdot \nabla T = -v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y}$$

where:

- $\mathbf{v} = (v_x, v_y)$ is the velocity field, representing the movement of the heat due to fluid flow,
- ∇T is the gradient of the temperature, representing the directional change in temperature.

In this simulation, the velocity vector $\mathbf{v} = [0, 0.01]$ causes heat to be advected primarily in the y-direction (upward).

3. Initial and Boundary Conditions

- **Initial Condition:** The temperature field is initialized with a Gaussian-like heat distribution centered at the middle of the domain. This represents a concentrated heat source that spreads out over time.
- **Boundary Conditions:** The domain has fixed temperature values at its boundaries (Dirichlet conditions). These remain constant throughout the simulation, enforcing a specific temperature on the edges of the domain.

4. Numerical Method

The simulation uses **Lagrange polynomials** to discretize the spatial derivatives in both the x- and y-directions. This allows us to approximate the spatial derivatives necessary for solving the heat equation.

At each time step, the temperature field is updated using the following equation:

$$T_{\text{new}} = T + \Delta t \left(\alpha \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) - v_x \frac{\partial T}{\partial x} - v_y \frac{\partial T}{\partial y} \right)$$

where Δt is the time step size, α controls the rate of heat diffusion, and v_x and v_y represent the velocities in the x- and y-directions (advection).

5. Visualization

The simulation results are visualized as contour plots, which show the evolving temperature field over time. The animation demonstrates how the heat diffuses across the domain while being transported upward by the advection process.

Key Physical Concepts

- **Heat Diffusion:** The process by which heat spreads from areas of high temperature to low temperature.
- **Advection:** The transport of heat due to the fluid motion, represented by the velocity vector $\mathbf{v}$.
- **Lagrange Polynomial Interpolation:** Used to discretize the spatial derivatives of the temperature field.
- **Boundary Conditions:** Fixed temperature values are enforced on the domain boundaries.

This combination of heat diffusion and advection captures the behavior of heat flow in systems with fluid motion and thermal effects. The numerical method used here allows for the accurate simulation of temperature evolution in the domain.

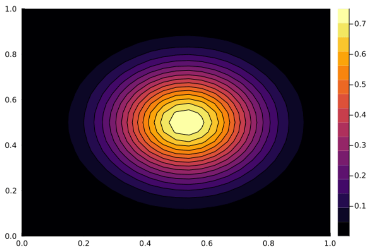

Figura 8: $t = 0\,\mathrm{s}$

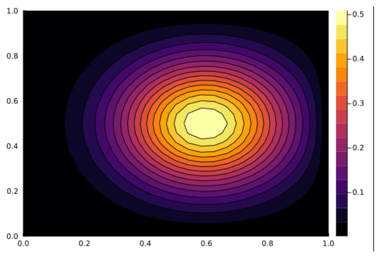

Figura 9: $t = 7.5\,\mathrm{s}$

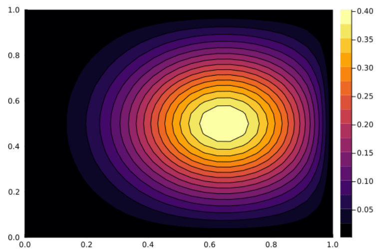

Figura 10: $t = 15\,\mathrm{s}$

2.0.1. Extra comparison

As an extra step, this code written by Javier solves the equation in 1D, 2D, or 3D , we will compare it to the other results

```
using LinearAlgebra, GLMakie


function heatpropagate(u::AbstractArray{T,N}, timesteps::Integer, ...
         tx      ::Real,
    uprev::AbstractArray{T,N}) where {T<:Number,N}

         tx            0.5 / N || throw(ArgumentError("      tx       = ...
        $      t    x   violates Fourier stability condition"))

    unitvecs = ntuple(i -> CartesianIndex(ntuple(==(i), Val(N))), Val(N))

    I = CartesianIndices(u)
    Ifirst, Ilast = first(I), last(I)
    I1 = oneunit(Ifirst)

    for t = 1:timesteps

        @inbounds @simd for i in Ifirst+I1:Ilast-I1

              u    = -2N * u[i]
            for uvec in unitvecs
                  u    += u[i+uvec] + u[i-uvec]
            end

            u[i] = uprev[i] +       tx        *    u

        end

        # Here Boundary conditions would be imposed (except if they are ...
            Dirichlet at boundary)

        uprev = u
    end

    return u, uprev
end

begin
    x = range(-1.0, 1.0, length=100)
    u = @. exp(-(x / 0.2)^2)
    u[1] = u[end] = 0.0
```

```

    set_publication_theme!()

    lines(x, u)

       = 1.0
     x   = x[2] - x[1]
    Fo = 1 / 2
     t   = Fo *   x ^2 /

    uprev = copy(u)
end

# @time heatpropagate(u, 20, Fo, uprev);

for i = 1:50
    u, uprev = heatpropagate(u, 20, Fo, uprev)
    lines!(x, u)
end
```

2.1. Graphs and tables

Graphs and tables summarizing the results obtained, allowing a comparison between the different methods and/or hardware platforms.

Method	**Simulation Time (seconds)**
Global Interpolant (matrix-matrix)	0.127
Global Interpolant (matrix-vector)	2.789
Finite Differences	0.646
Javier‘s Code	??

Tabla 1: Comparison of methods and expected simulation times

3. Conclusions

This study researches different numerical methods to solve the heat equation with convective and diffusive terms. The aim is to compare the different approaches and verify that the theoretical hypothesis are approximately proven.

4. Bibliografía

- Juan A. Hernandez Ramos ´ Mario A. Zamecnik Barros. *Interpolación Polinómica de Alto orden, Métodos Espectrales*
- Smith, G. D. (1985). *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press.
- LeVeque, R. J. (2007). *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics.
- Kirk, D. B., & Hwu, W.-m. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.
- Román Bermejo, J. (2024). *CPU Performance: Benchmark Analysis and Theoretical Limits.*
- Boyd2001 J. P. Boyd. (2001)*Chebyshev and Fourier Spectral Methods*, 2nd edition, Dover Publications.