SHARCNET General Interest SEMINAR

Solving the 2D diffusion PDE using MPI with Claude AI

Ge Baolai
bge@sharcnet.ca
Western University
SHARCNET | Compute Ontario
Digital Research Alliance of Canada

Purpose of the project

We would like

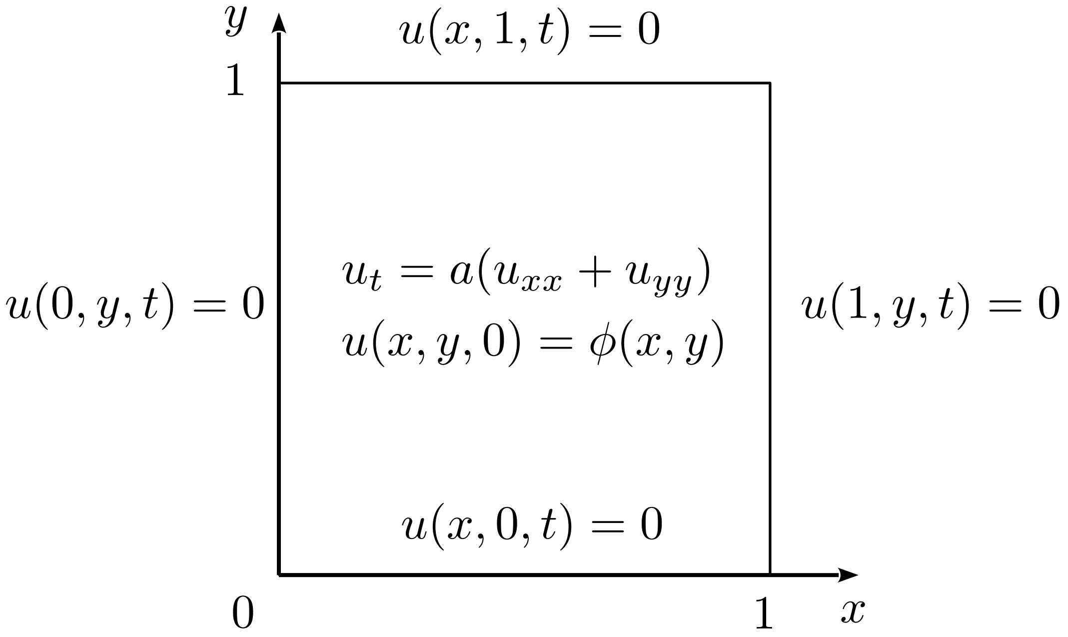
- To see how well Claude can “understand” math problems, in natural language.
- To see if Claude can translate a math problem into computer programs for solutions by itself.
- To see if Claude can draft an implementation plan for coding.
- To see how “good” the code looks like.
- To see if the code is bug free, can it “debug” the code?
- To see if the code generates correct result?
- Overall, to see if Claude can play a role of research assistant and at what level.

Scope and criteria

- We selected a classic PDE problem: 2D diffusion equation on a rectangular domain.
- To be solved numerically in parallel using MPI.
- We give a set tasks to Claude
    - Task 1: Literature search.
    - Task 2: Formulation.
    - Task 3: Implementation design.
    - Task 4: Code implementation.
    - Task 5: Code verification and visualization.
    - Task 5: Final report.
- We take a laissez-faire approach

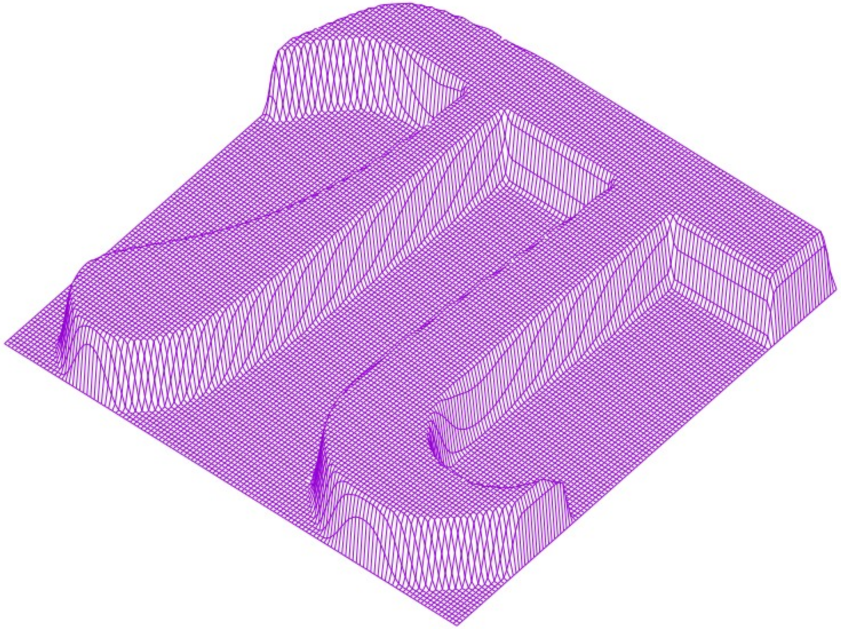
The statement of the mathematical problem

Consider a 2D parabolic equation – diffusion equation – on a unit square with Dirichlet boundary condition and initial condition at $t=0$.

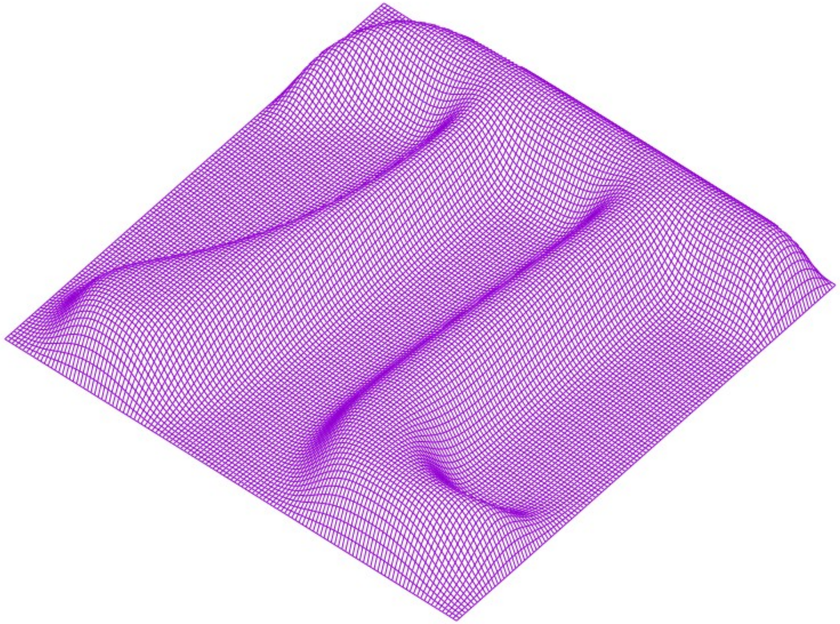


The statement of the mathematical problem

The solution looks like this over time



Initial state at $t = 0$



Some time later

Task 1: Literature search

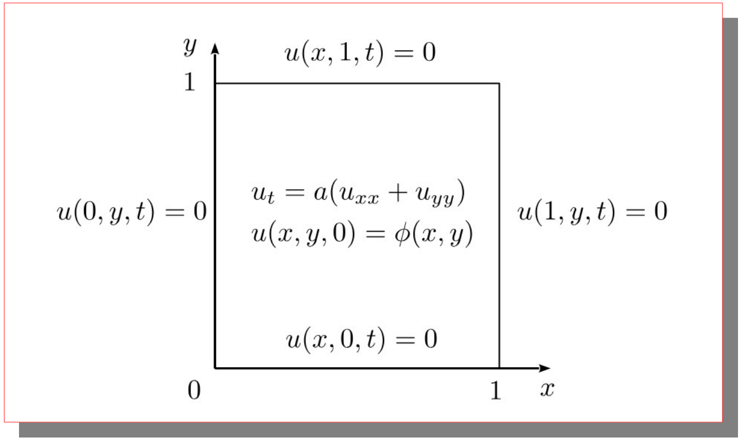
Literature review

Me: Let's start with Task 1. I'll show you the mathematical statement of the problem. Tell me are you able to see LaTeX expressions or you are able to see an screenshot of the mathematical expressions?

Claude: I can read and interpret raw LaTeX code directly (e.g., \frac{d^2u}{dx^2} = f(x)). I can also read mathematical expressions from an uploaded image or screenshot, as I can analyze images.

Literature review

Me: The mathematical statement of the problem is as follows (image uploaded):



Claude: I can see the problem clearly. Let me summarize what I understand:

The Problem

We are solving the 2D heat/diffusion equation:

$$u_t = a(u_{xx}+u_{yy})$$

on the unit square $(x,y)\in[0,1]\times[0,1]$ for $t>0$.

Boundary Conditions (Dirichlet, homogeneous):

- $u(0,y,t)=0$ (left)
- $u(1,y,t)=0$ (right)
- $u(x,0,t)=0$ (bottom)
- $u(x,1,t)=0$ (top)

Initial Condition:

- $u(x,y,0)=\phi(x,y)$ — a given function to be specified

Literature review

Me: Yes, please proceed.

Claude: Task 1: Literature Search

Step 1: Problem Classification

Let's start by properly classifying the problem, as this will guide the literature search.

The equation $u_t = a(u_{xx} + u_{yy})$ can be written as

$$u_t = a\nabla^2 u$$

where $\nabla^2 = \partial^2/\partial x^2 + \partial^2/\partial y^2$ is a Laplacian in 2D.

This is a:

- Linear PDE ☑
- Parabolic PDE ☑ (one time derivative, second-order in space)
- Initial Boundary Value Problem (IBVP) ☑

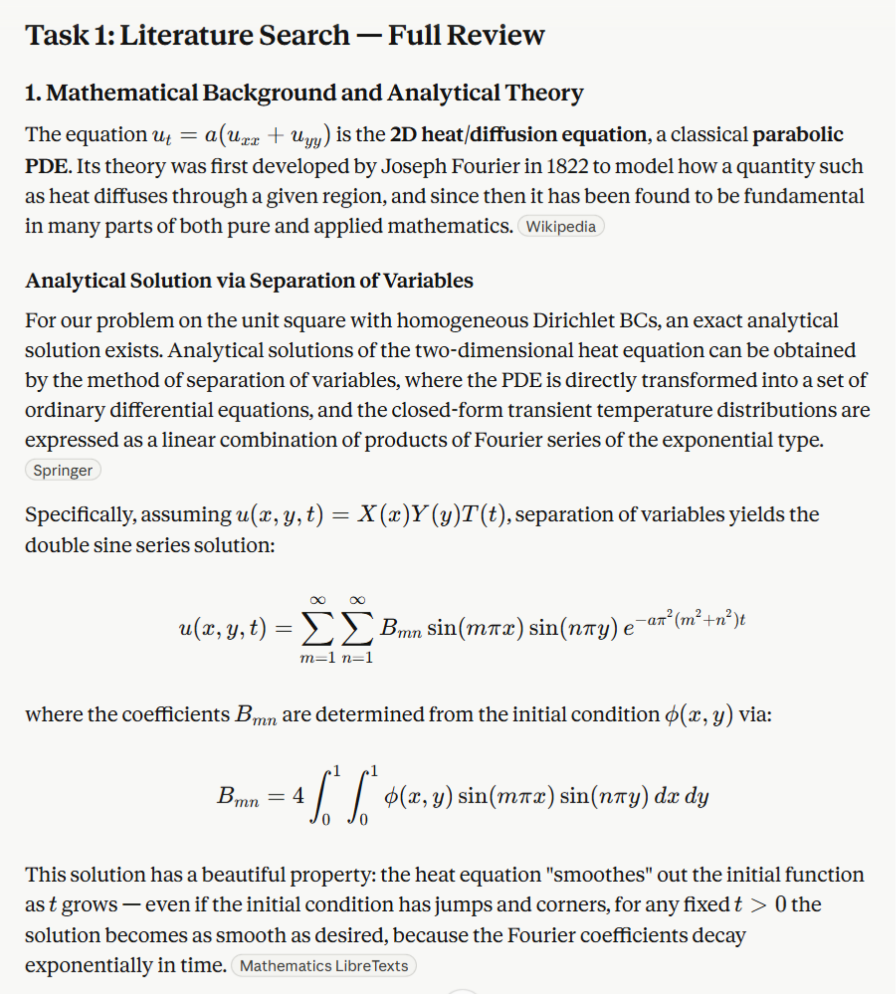
This places it squarely in the classical theory of parabolic PDEs, closely related to the heat equation.

It stopped right there.

Literature review

Me: Wait, you haven't done the literature search that you mentioned above. Let's do a thorough review first.

Claude: You're absolutely right — I jumped ahead without actually doing the literature search. My apologies for that! Let me now conduct a proper one.



Task 1: Literature Search — Full Review

1. Mathematical Background and Analytical Theory

The equation $u_t = a(u_{xx} + u_{yy})$ is the **2D heat/diffusion equation**, a classical **parabolic PDE.** Its theory was first developed by Joseph Fourier in 1822 to model how a quantity such as heat diffuses through a given region, and since then it has been found to be fundamental in many parts of both pure and applied mathematics. Wikipedia

Analytical Solution via Separation of Variables

For our problem on the unit square with homogeneous Dirichlet BCs, an exact analytical solution exists. Analytical solutions of the two-dimensional heat equation can be obtained by the method of separation of variables, where the PDE is directly transformed into a set of ordinary differential equations, and the closed-form transient temperature distributions are expressed as a linear combination of products of Fourier series of the exponential type. Springer

Specifically, assuming $u(x,y,t) = X(x)Y(y)T(t)$, separation of variables yields the double sine series solution:

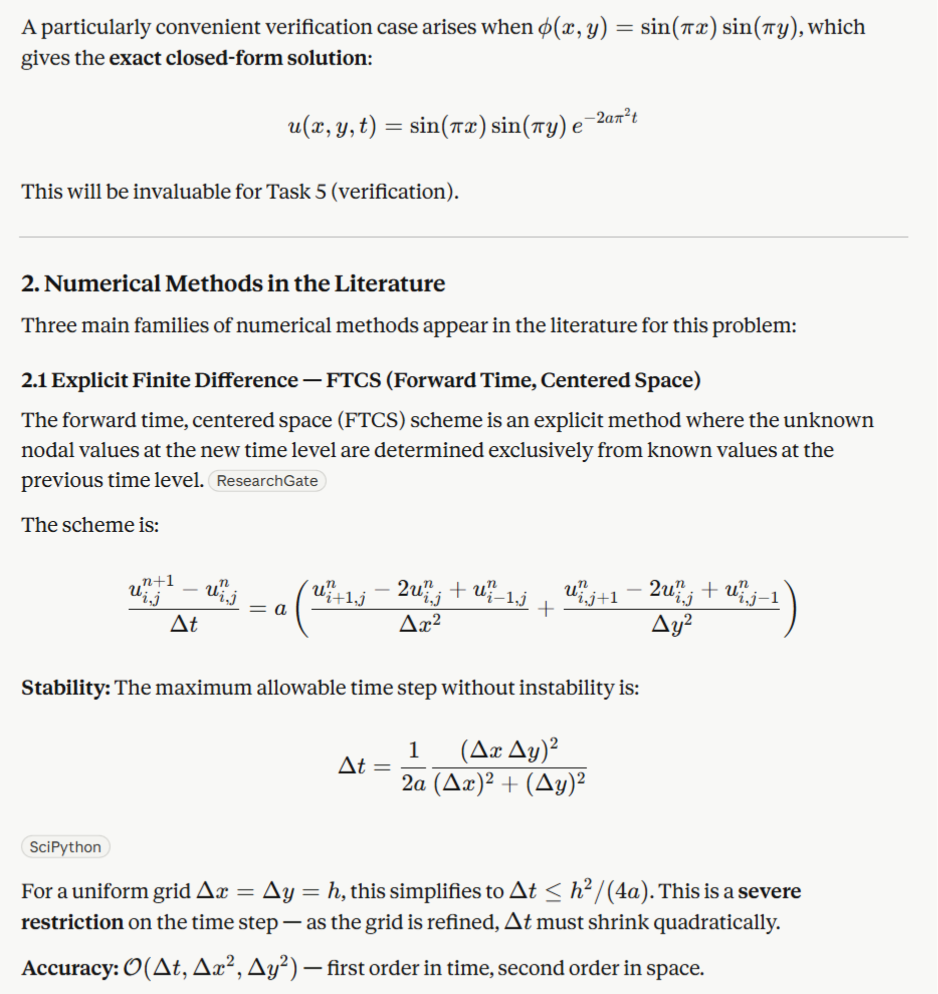
$$u(x,y,t) = \sum_{m=1}^{\infty}\sum_{n=1}^{\infty} B_{mn}\sin(m\pi x)\sin(n\pi y)\,e^{-a\pi^2(m^2+n^2)t}$$

where the coefficients B_{mn} are determined from the initial condition $\phi(x,y)$ via:

$$B_{mn} = 4\int_0^1\int_0^1 \phi(x,y)\sin(m\pi x)\sin(n\pi y)\,dx\,dy$$

This solution has a beautiful property: the heat equation "smoothes" out the initial function as t grows — even if the initial condition has jumps and corners, for any fixed $t > 0$ the solution becomes as smooth as desired, because the Fourier coefficients decay exponentially in time. Mathematics LibreTexts

Literature review



A particularly convenient verification case arises when $\phi(x,y)=\sin(\pi x)\sin(\pi y)$, which gives the **exact closed-form solution**:

$$u(x,y,t)=\sin(\pi x)\sin(\pi y)\,e^{-2a\pi^2 t}$$

This will be invaluable for Task 5 (verification).

2. Numerical Methods in the Literature

Three main families of numerical methods appear in the literature for this problem:

2.1 Explicit Finite Difference — FTCS (Forward Time, Centered Space)

The forward time, centered space (FTCS) scheme is an explicit method where the unknown nodal values at the new time level are determined exclusively from known values at the previous time level. ResearchGate

The scheme is:

$$\frac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t}=a\left(\frac{u_{i+1,j}^{n}-2u_{i,j}^{n}+u_{i-1,j}^{n}}{\Delta x^2}+\frac{u_{i,j+1}^{n}-2u_{i,j}^{n}+u_{i,j-1}^{n}}{\Delta y^2}\right)$$

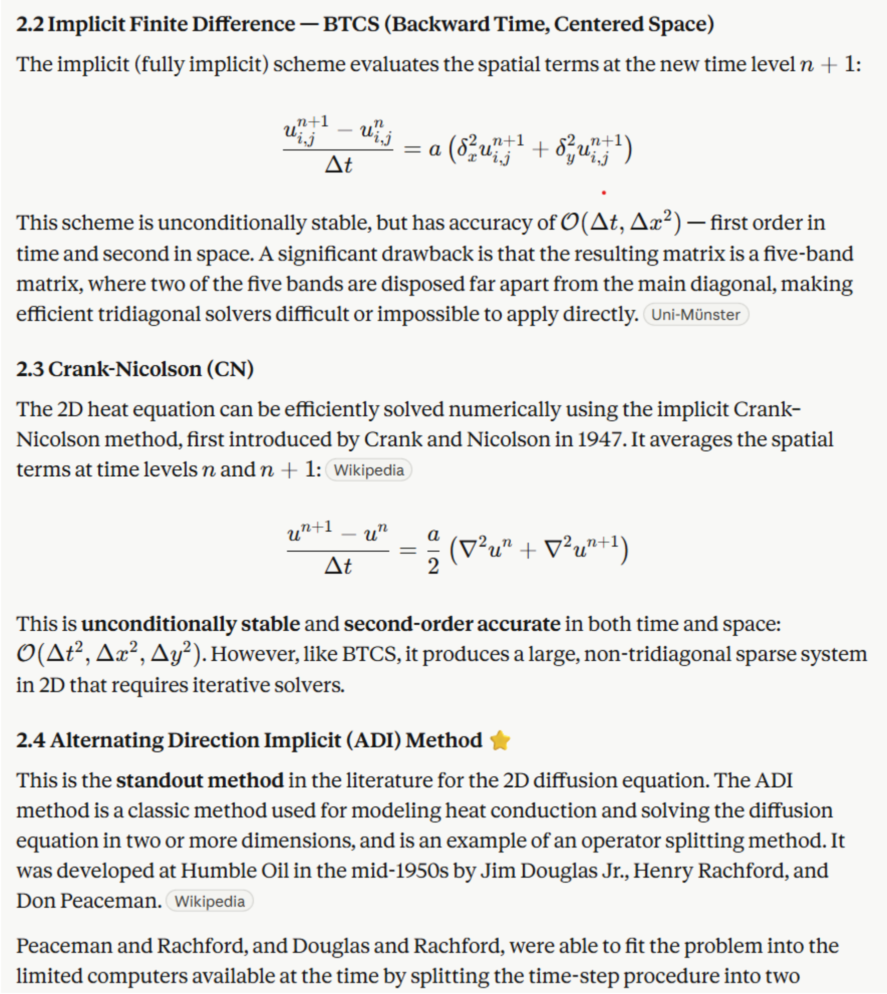
Stability: The maximum allowable time step without instability is:

$$\Delta t=\frac{1}{2a}\frac{(\Delta x\,\Delta y)^2}{(\Delta x)^2+(\Delta y)^2}$$

SciPython

For a uniform grid $\Delta x=\Delta y=h$, this simplifies to $\Delta t\le h^2/(4a)$. This is a **severe restriction** on the time step — as the grid is refined, Δt must shrink quadratically.

Accuracy: $\mathcal{O}(\Delta t,\Delta x^2,\Delta y^2)$ — first order in time, second order in space.



2.2 Implicit Finite Difference — BTCS (Backward Time, Centered Space)

The implicit (fully implicit) scheme evaluates the spatial terms at the new time level $n+1$:

$$\frac{u_{i,j}^{n+1}-u_{i,j}^{n}}{\Delta t}=a\left(\delta_x^2 u_{i,j}^{n+1}+\delta_y^2 u_{i,j}^{n+1}\right)$$

This scheme is unconditionally stable, but has accuracy of $\mathcal{O}(\Delta t,\Delta x^2)$ — first order in time and second in space. A significant drawback is that the resulting matrix is a five-band matrix, where two of the five bands are disposed far apart from the main diagonal, making efficient tridiagonal solvers difficult or impossible to apply directly. Uni-Münster

2.3 Crank-Nicolson (CN)

The 2D heat equation can be efficiently solved numerically using the implicit Crank-Nicolson method, first introduced by Crank and Nicolson in 1947. It averages the spatial terms at time levels n and $n+1$: Wikipedia

$$\frac{u^{n+1}-u^{n}}{\Delta t}=\frac{a}{2}\left(\nabla^2 u^{n}+\nabla^2 u^{n+1}\right)$$

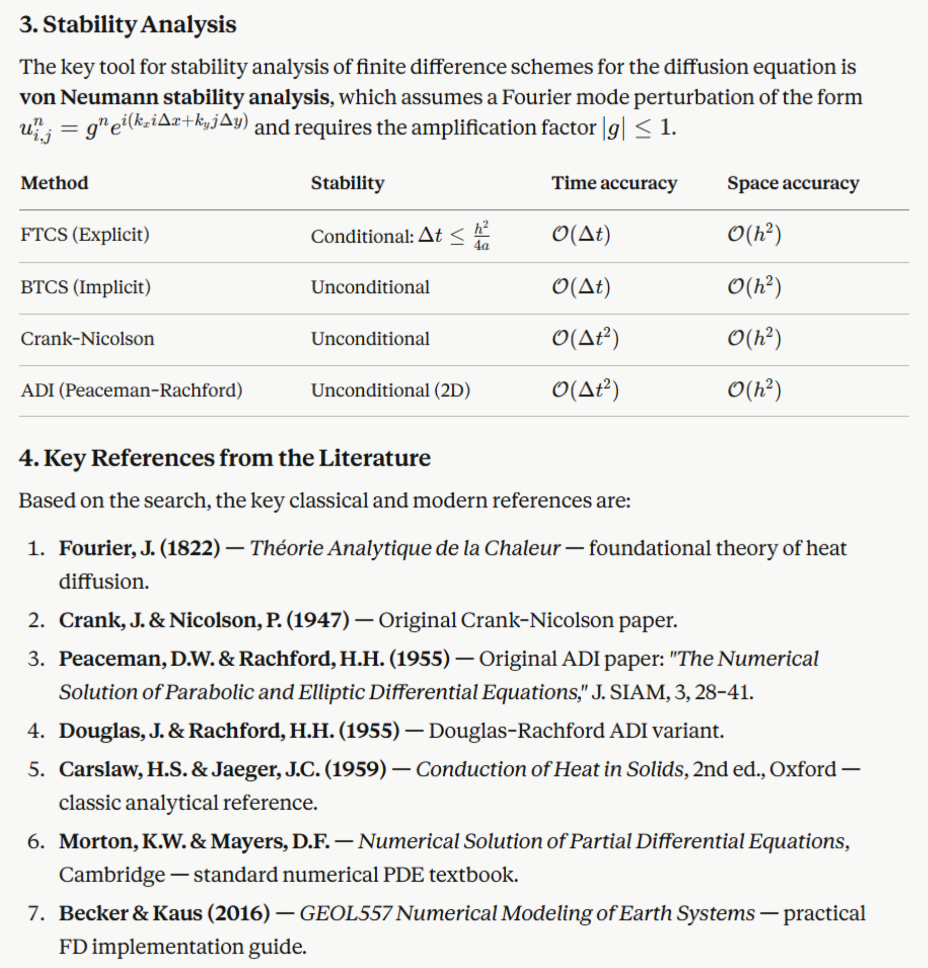
This is **unconditionally stable** and **second-order accurate** in both time and space: $\mathcal{O}(\Delta t^2,\Delta x^2,\Delta y^2)$. However, like BTCS, it produces a large, non-tridiagonal sparse system in 2D that requires iterative solvers.

2.4 Alternating Direction Implicit (ADI) Method ⭐

This is the **standout method** in the literature for the 2D diffusion equation. The ADI method is a classic method used for modeling heat conduction and solving the diffusion equation in two or more dimensions, and is an example of an operator splitting method. It was developed at Humble Oil in the mid-1950s by Jim Douglas Jr., Henry Rachford, and Don Peaceman. Wikipedia

Peaceman and Rachford, and Douglas and Rachford, were able to fit the problem into the limited computers available at the time by splitting the time-step procedure into two

Literature review



3. Stability Analysis

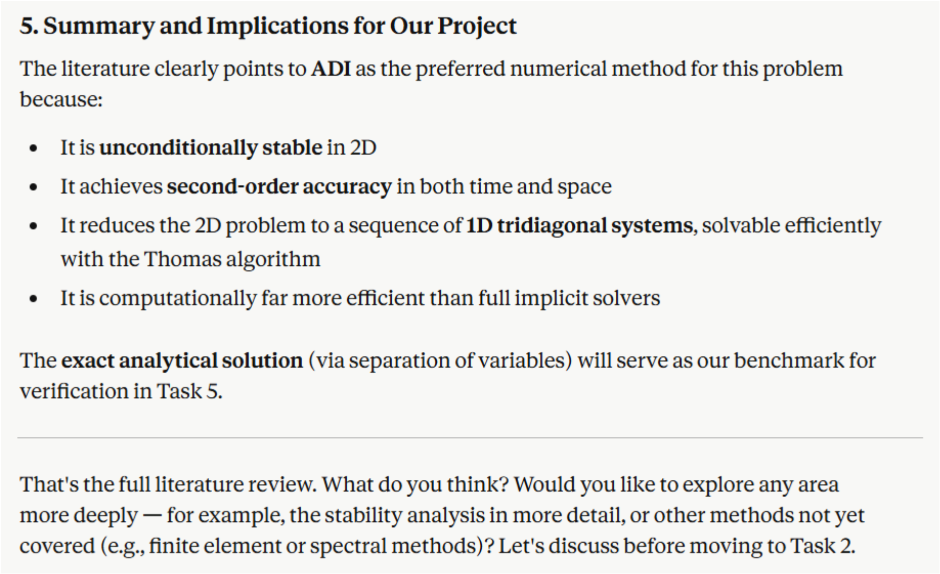
The key tool for stability analysis of finite difference schemes for the diffusion equation is **von Neumann stability analysis**, which assumes a Fourier mode perturbation of the form $u^n_{i,j} = g^n e^{i(k_x i\Delta x + k_y j\Delta y)}$ and requires the amplification factor $|g| \le 1$.

Method	Stability	Time accuracy	Space accuracy
FTCS (Explicit)	Conditional: $\Delta t \le \frac{h^2}{4a}$	$\mathcal{O}(\Delta t)$	$\mathcal{O}(h^2)$
BTCS (Implicit)	Unconditional	$\mathcal{O}(\Delta t)$	$\mathcal{O}(h^2)$
Crank-Nicolson	Unconditional	$\mathcal{O}(\Delta t^2)$	$\mathcal{O}(h^2)$
ADI (Peaceman-Rachford)	Unconditional (2D)	$\mathcal{O}(\Delta t^2)$	$\mathcal{O}(h^2)$

4. Key References from the Literature

Based on the search, the key classical and modern references are:

1. **Fourier, J. (1822)** — *Théorie Analytique de la Chaleur* — foundational theory of heat diffusion.
2. **Crank, J. & Nicolson, P. (1947)** — Original Crank-Nicolson paper.
3. **Peaceman, D.W. & Rachford, H.H. (1955)** — Original ADI paper: *"The Numerical Solution of Parabolic and Elliptic Differential Equations,"* J. SIAM, 3, 28–41.
4. **Douglas, J. & Rachford, H.H. (1955)** — Douglas-Rachford ADI variant.
5. **Carslaw, H.S. & Jaeger, J.C. (1959)** — *Conduction of Heat in Solids*, 2nd ed., Oxford — classic analytical reference.
6. **Morton, K.W. & Mayers, D.F.** — *Numerical Solution of Partial Differential Equations*, Cambridge — standard numerical PDE textbook.
7. **Becker & Kaus (2016)** — *GEOL557 Numerical Modeling of Earth Systems* — practical FD implementation guide.



5. Summary and Implications for Our Project

The literature clearly points to **ADI** as the preferred numerical method for this problem because:

- It is **unconditionally stable** in 2D
- It achieves **second-order accuracy** in both time and space
- It reduces the 2D problem to a sequence of **1D tridiagonal systems**, solvable efficiently with the Thomas algorithm
- It is computationally far more efficient than full implicit solvers

The **exact analytical solution** (via separation of variables) will serve as our benchmark for verification in Task 5.

That's the full literature review. What do you think? Would you like to explore any area more deeply — for example, the stability analysis in more detail, or other methods not yet covered (e.g., finite element or spectral methods)? Let's discuss before moving to Task 2.

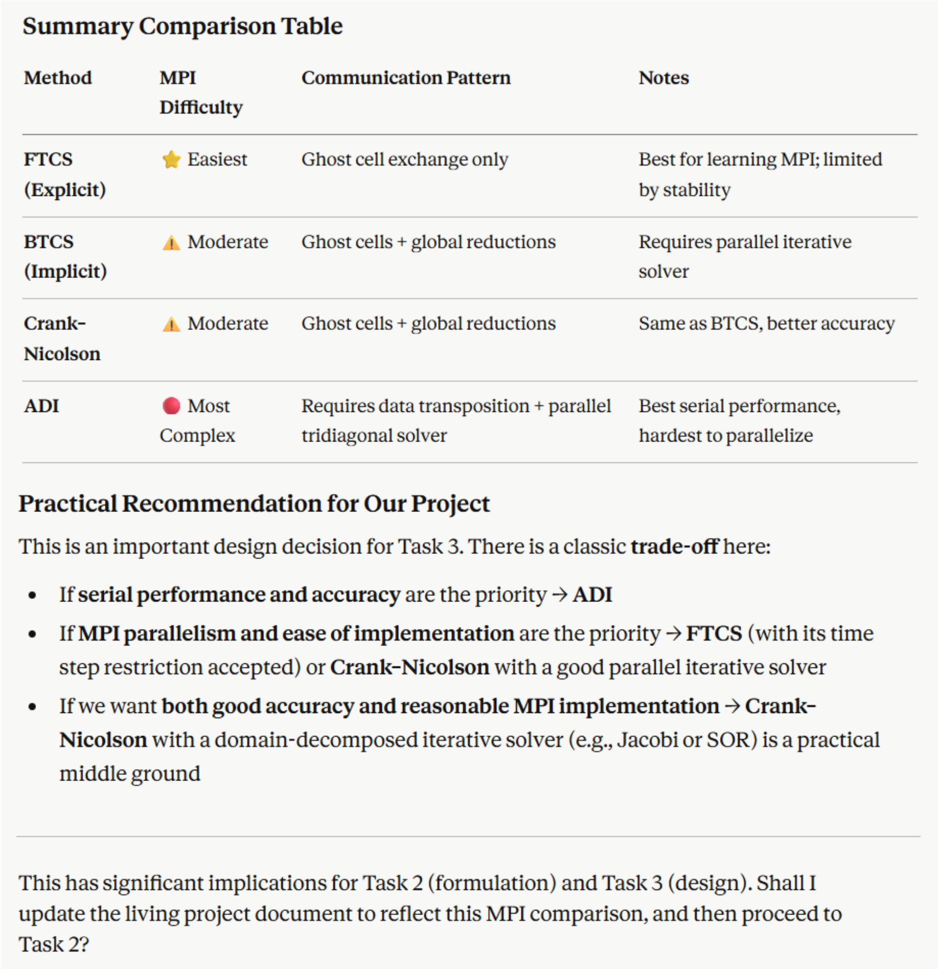
Literature review

Me: This great! Yes, we can move on to Task 2. But before we go, let's step back a bit, and take a look at the four numerical methods. I'd like to compare among them the ease of implementing a parallel solution using message passing interface (MPI). Can we do that first?

Claude: Great idea — this is very relevant to Task 3 (implementation design) as well. Let me do a proper literature search on this before commenting.

Excellent — the literature is very clear on this. Here is the comparison:

Full text details omitted



Summary Comparison Table

Method	MPI Difficulty	Communication Pattern	Notes
FTCS (Explicit)	⭐ Easiest	Ghost cell exchange only	Best for learning MPI; limited by stability
BTCS (Implicit)	⚠️ Moderate	Ghost cells + global reductions	Requires parallel iterative solver
Crank-Nicolson	⚠️ Moderate	Ghost cells + global reductions	Same as BTCS, better accuracy
ADI	🔴 Most Complex	Requires data transposition + parallel tridiagonal solver	Best serial performance, hardest to parallelize

Practical Recommendation for Our Project

This is an important design decision for Task 3. There is a classic **trade-off** here:

- If **serial performance and accuracy** are the priority → **ADI**
- If **MPI parallelism and ease of implementation** are the priority → **FTCS** (with its time step restriction accepted) or **Crank-Nicolson** with a good parallel iterative solver
- If we want **both good accuracy and reasonable MPI implementation** → **Crank-Nicolson** with a domain-decomposed iterative solver (e.g., Jacobi or SOR) is a practical middle ground

---

This has significant implications for Task 2 (formulation) and Task 3 (design). Shall I update the living project document to reflect this MPI comparison, and then proceed to Task 2?

Literature review

Me: To achieve the ultimate goal of solving the mathematical problem, let's make a decision: Let's choose the explicit finite difference - FTCS (forward time, centred space) - method for the project, as it is easy to implement and hence give us first available solution quick. We shall fork the project with another method as a second project later.

Lte's move on the Task 2.

Claude: Great decision — and a very pragmatic engineering approach. Start with the simplest correct solution first, then improve.

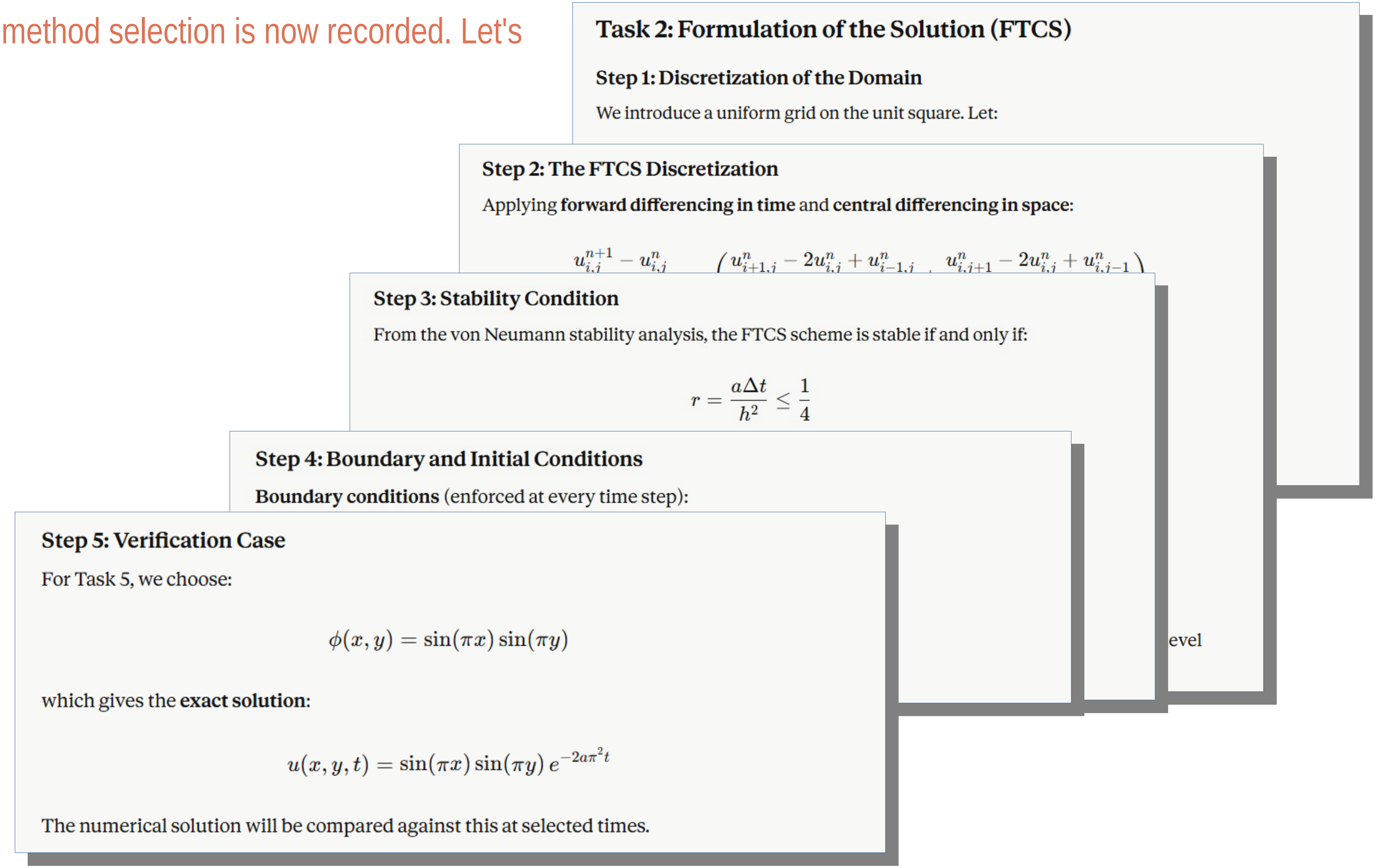
One small typo to flag: "Lte's" should be "Let's" in your last line. No issue, just keeping my promise to point these out! 😊

Let me update the project log to record this decision before we dive into Task 2.

Task 2: Numerical formulation

Numerical formulation

Claude: Good. The method selection is now recorded. Let's proceed.



Task 2: Formulation of the Solution (FTCS)

Step 1: Discretization of the Domain

We introduce a uniform grid on the unit square. Let:

Step 2: The FTCS Discretization

Applying **forward differencing in time** and **central differencing in space**:

$u_{i,j}^{n+1} - u_{i,j}^{n}$ $\Big(u_{i+1,j}^{n} - 2u_{i,j}^{n} + u_{i-1,j}^{n}$ $u_{i,j+1}^{n} - 2u_{i,j}^{n} + u_{i,j-1}^{n} \Big)$

Step 3: Stability Condition

From the von Neumann stability analysis, the FTCS scheme is stable if and only if:

$$r = \frac{a\Delta t}{h^2} \leq \frac{1}{4}$$

Step 4: Boundary and Initial Conditions

Boundary conditions (enforced at every time step):

evel

Step 5: Verification Case

For Task 5, we choose:

$$\phi(x,y) = \sin(\pi x)\sin(\pi y)$$

which gives the **exact solution**:

$$u(x,y,t) = \sin(\pi x)\sin(\pi y)\, e^{-2a\pi^2 t}$$

The numerical solution will be compared against this at selected times.

Numerical formulation

Claude: Open Questions for Discussion

Before we move to Task 3, I'd like your input on two things:

- Value of a : Do you have a preferred value, or shall we use $a=1$ for simplicity?
- Grid size N : Do you have a preference, or shall we start with something like $N=50$ and refine later?

What do you think of the formulation so far? Any changes or additions before we proceed?

Me: The formulation is great. It's clear and ready for implementation. Before we move to the next task, let me answer your questions:

1. The diffusion constant a is a predefined the constant. Without the loss of generality, we can set it to 1.

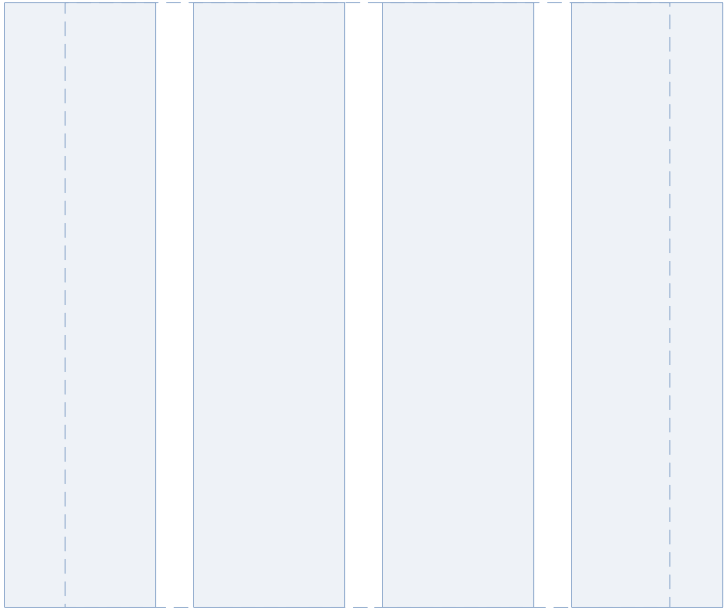
2. Grid size, determined by N. It can be set to any numerically meaningful value, subject to the stability condition, which is constrained by Δt.

3. Initial condition. It is a great idea to compare with the true solution for verification. So yes, we can use it. But I'd also like to use something more visually interesting later. We can defer that to Task 5.

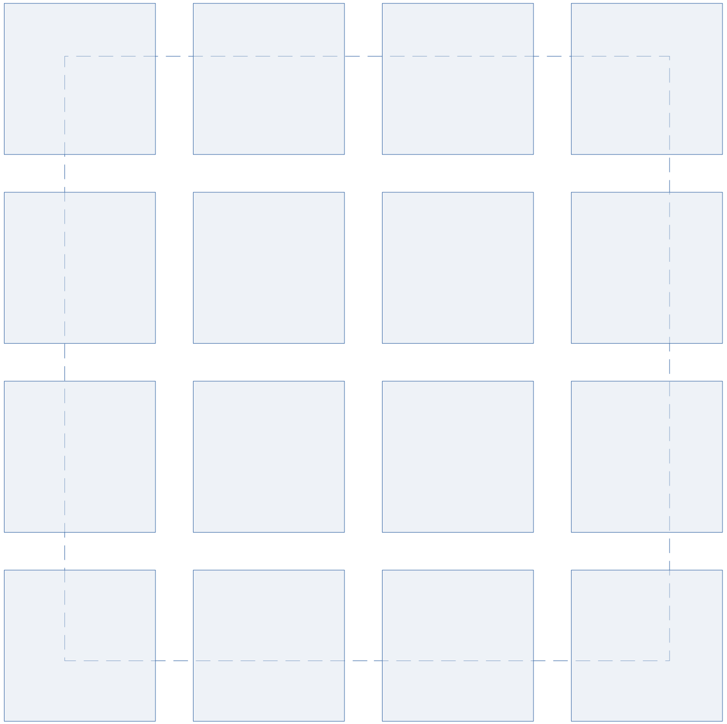
Task 3: Implementation design

Implementation design

Domain decomposition



1D partition



2D partition

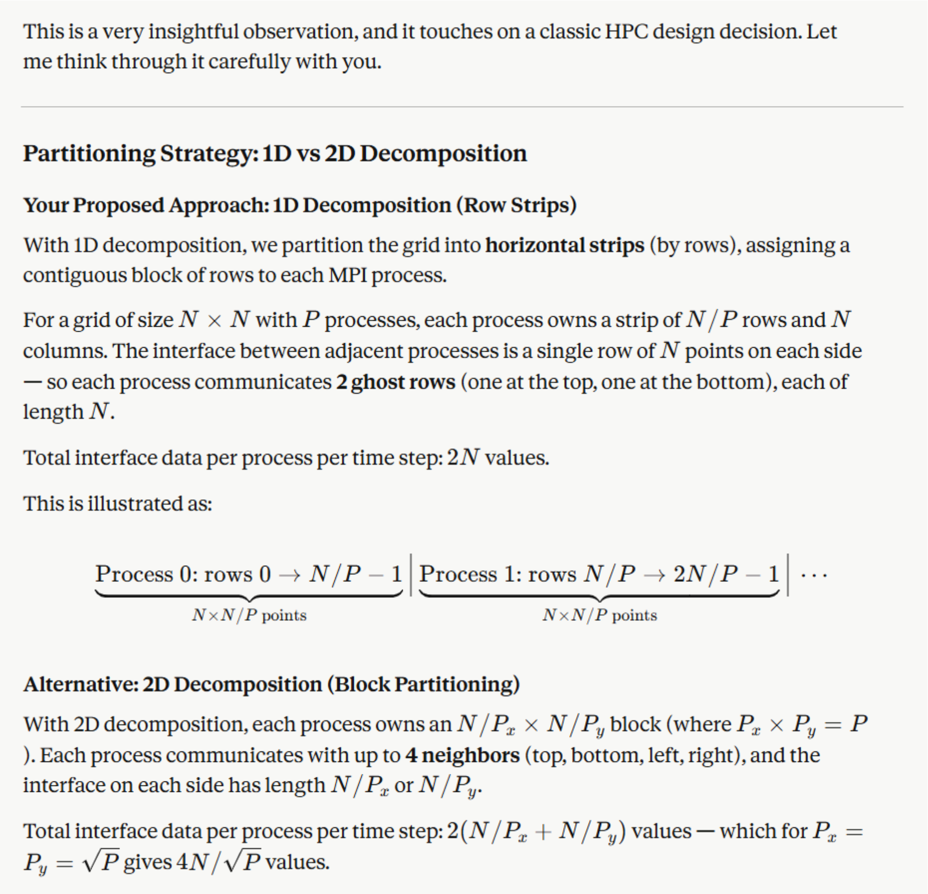
Implementation design

Me: Let's now work on the implementation design. The update formula $u_{i,j}^{n+1} = u_{i,j}^{n} + r(......)$ reveals that two arrays are to be used, one holds $u_{i,j}^{n+1}$ and one holds $u_{i,j}^{n}$.

For domain decomposition, would it make sense to partition in one direction instead of both, so the subdomains will have a total smaller area of interfaces to minimize the MPI message passing along the interface between adjacent subdomains?

Claude: This is a very insightful observation, and it touches on a classic HPC design decision. Let me think through it carefully with you.

Implementation design



This is a very insightful observation, and it touches on a classic HPC design decision. Let me think through it carefully with you.

Partitioning Strategy: 1D vs 2D Decomposition

Your Proposed Approach: 1D Decomposition (Row Strips)

With 1D decomposition, we partition the grid into **horizontal strips** (by rows), assigning a contiguous block of rows to each MPI process.

For a grid of size $N \times N$ with P processes, each process owns a strip of N/P rows and N columns. The interface between adjacent processes is a single row of N points on each side — so each process communicates **2 ghost rows** (one at the top, one at the bottom), each of length N.

Total interface data per process per time step: $2N$ values.

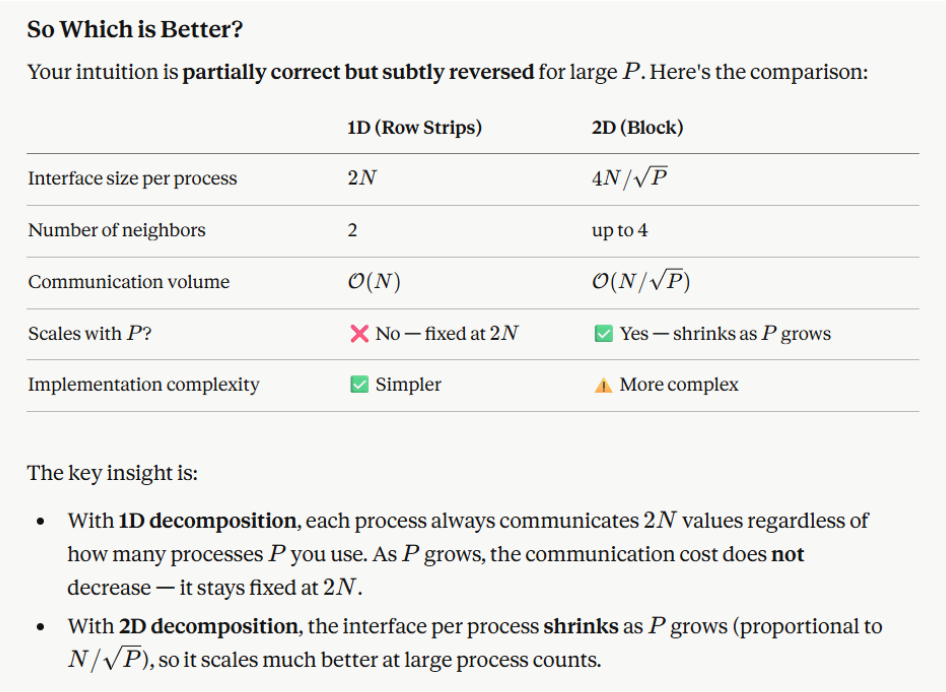
This is illustrated as:

$$\underbrace{\text{Process 0: rows } 0 \to N/P-1}_{N\times N/P\text{ points}} \Big| \underbrace{\text{Process 1: rows } N/P \to 2N/P-1}_{N\times N/P\text{ points}} \Big| \cdots$$

Alternative: 2D Decomposition (Block Partitioning)

With 2D decomposition, each process owns an $N/P_x \times N/P_y$ block (where $P_x \times P_y = P$). Each process communicates with up to **4 neighbors** (top, bottom, left, right), and the interface on each side has length N/P_x or N/P_y.

Total interface data per process per time step: $2(N/P_x + N/P_y)$ values — which for $P_x = P_y = \sqrt{P}$ gives $4N/\sqrt{P}$ values.



So Which is Better?

Your intuition is **partially correct but subtly reversed** for large P. Here's the comparison:

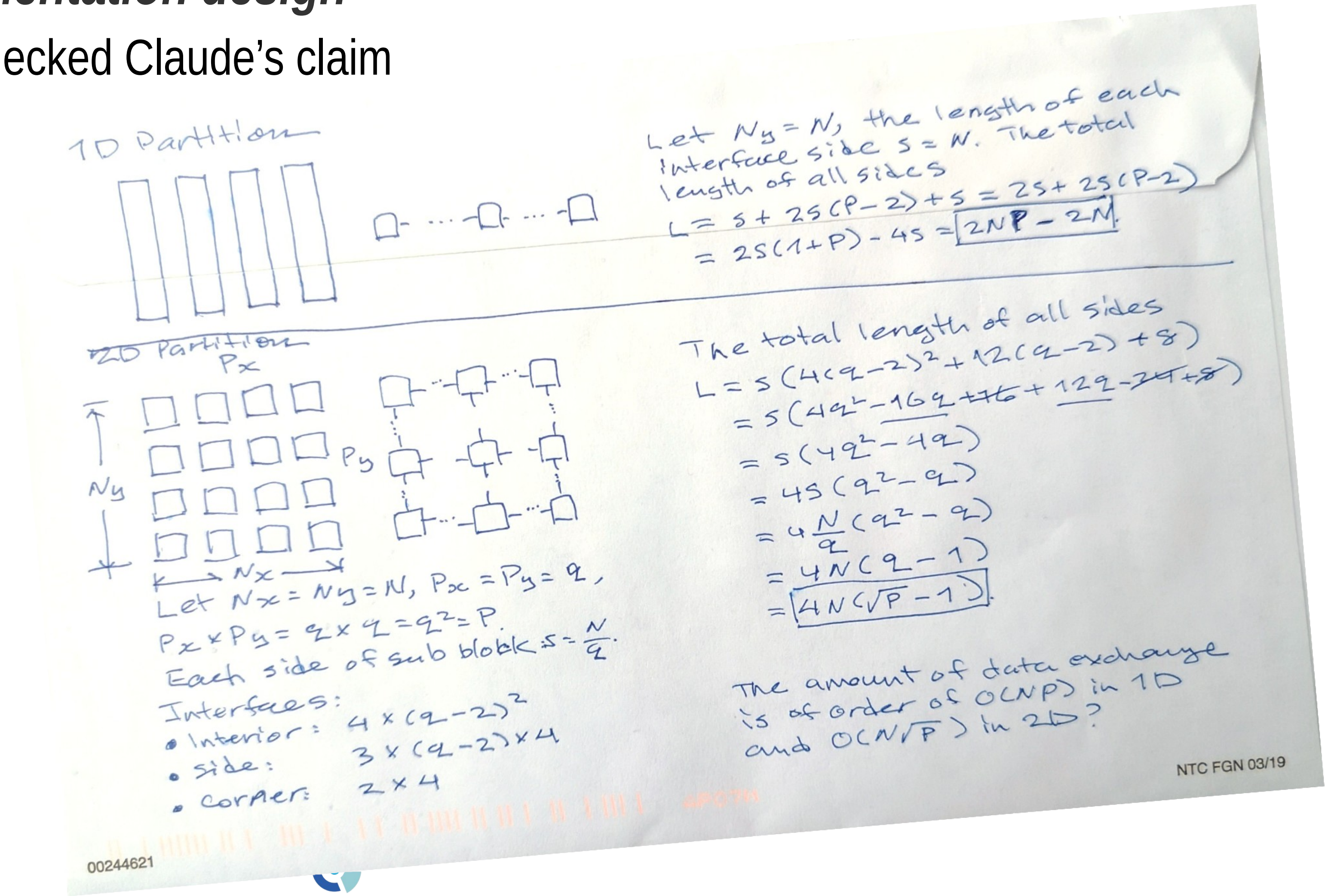
	1D (Row Strips)	**2D (Block)**
Interface size per process	$2N$	$4N/\sqrt{P}$
Number of neighbors	2	up to 4
Communication volume	$\mathcal{O}(N)$	$\mathcal{O}(N/\sqrt{P})$
Scales with P?	❌ No — fixed at $2N$	✅ Yes — shrinks as P grows
Implementation complexity	✅ Simpler	⚠ More complex

The key insight is:

- With **1D decomposition**, each process always communicates $2N$ values regardless of how many processes P you use. As P grows, the communication cost does **not** decrease — it stays fixed at $2N$.
- With **2D decomposition**, the interface per process **shrinks** as P grows (proportional to $N/\sqrt{P}$), so it scales much better at large process counts.

Implementation design

I hand checked Claude's claim



Let $N_y = N$, the length of each interface side $s = N$. The total length of all sides

$L = s + 2s(P-2) + s = 2s + 2s(P-2)$

$= 2s(1+P) - 4s = \boxed{2NP - 2N}.$

Let $N_x = N_y = N$, $P_x = P_y = q$,

$P_x \times P_y = q \times q = q^2 = P$.

Each side of sub block $s = \frac{N}{q}$.

Interfaces:

- Interior: $4 \times (q-2)^2$
- side: $3 \times (q-2) \times 4$
- corner: 2×4

The total length of all sides

$L = s(4(q-2)^2 + 12(q-2) + 8)$

$= s(4q^2 - \underline{16q} \cancel{+16} + \underline{12q} \cancel{-24} \cancel{+8})$

$= s(4q^2 - 4q)$

$= 4s(q^2 - q)$

$= 4\frac{N}{q}(q^2 - q)$

$= 4N(q-1)$

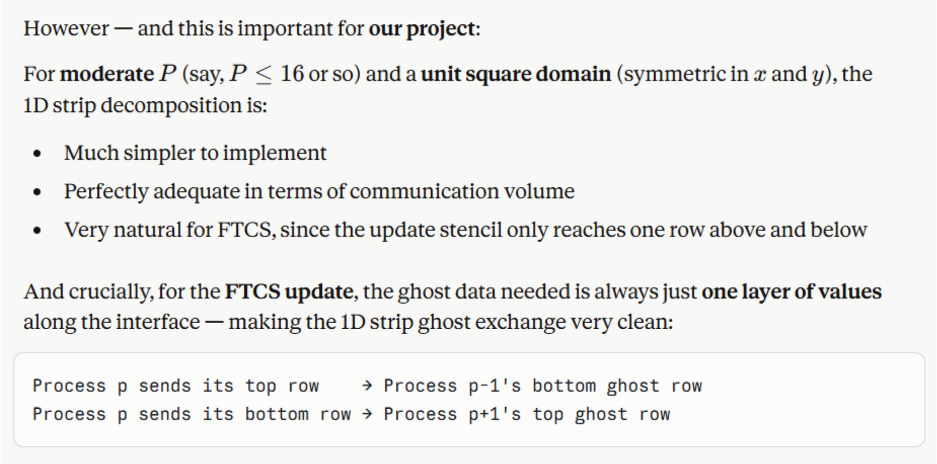
$= \boxed{4N(\sqrt{P} - 1)}.$

The amount of data exchange is of order of $O(NP)$ in 1D and $O(N\sqrt{P})$ in 2D?

NTC FGN 03/19

00244621

Implementation design



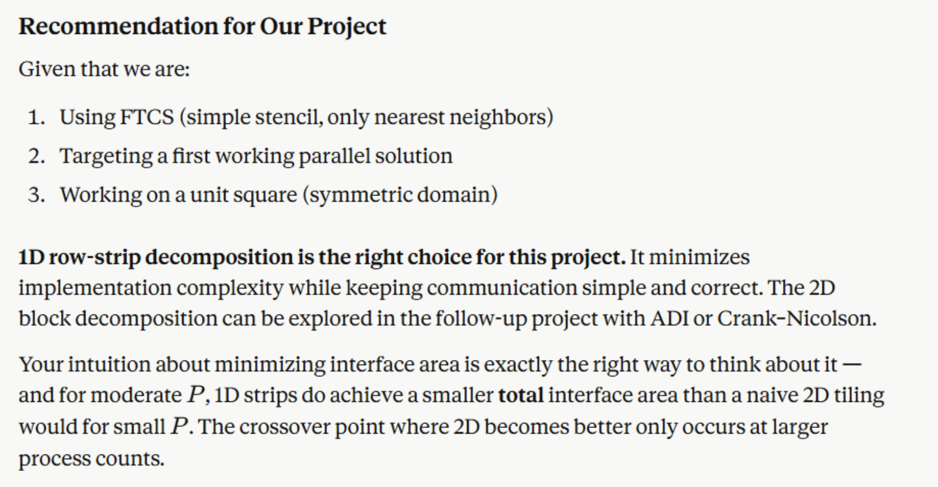
However — and this is important for **our project**:

For **moderate** P (say, $P \leq 16$ or so) and a **unit square domain** (symmetric in x and y), the 1D strip decomposition is:

- Much simpler to implement
- Perfectly adequate in terms of communication volume
- Very natural for FTCS, since the update stencil only reaches one row above and below

And crucially, for the **FTCS update**, the ghost data needed is always just **one layer of values** along the interface — making the 1D strip ghost exchange very clean:

```
Process p sends its top row    → Process p-1's bottom ghost row
Process p sends its bottom row → Process p+1's top ghost row
```



Recommendation for Our Project

Given that we are:

1. Using FTCS (simple stencil, only nearest neighbors)
2. Targeting a first working parallel solution
3. Working on a unit square (symmetric domain)

1D row-strip decomposition is the right choice for this project. It minimizes implementation complexity while keeping communication simple and correct. The 2D block decomposition can be explored in the follow-up project with ADI or Crank-Nicolson.

Your intuition about minimizing interface area is exactly the right way to think about it — and for moderate P, 1D strips do achieve a smaller **total** interface area than a naive 2D tiling would for small P. The crossover point where 2D becomes better only occurs at larger process counts.

Note: Claude is making recommendations here:

- 1D decomposition is much simpler to implement.
- Adequate in terms of communication volume.

Its arguments

- Targeting at a first working parallel solution.
- Using FTCS, the ghost data (MPI term for exchanging data owned by the neighboring rank) needed is always just one layer of values along the interface (not convincing)

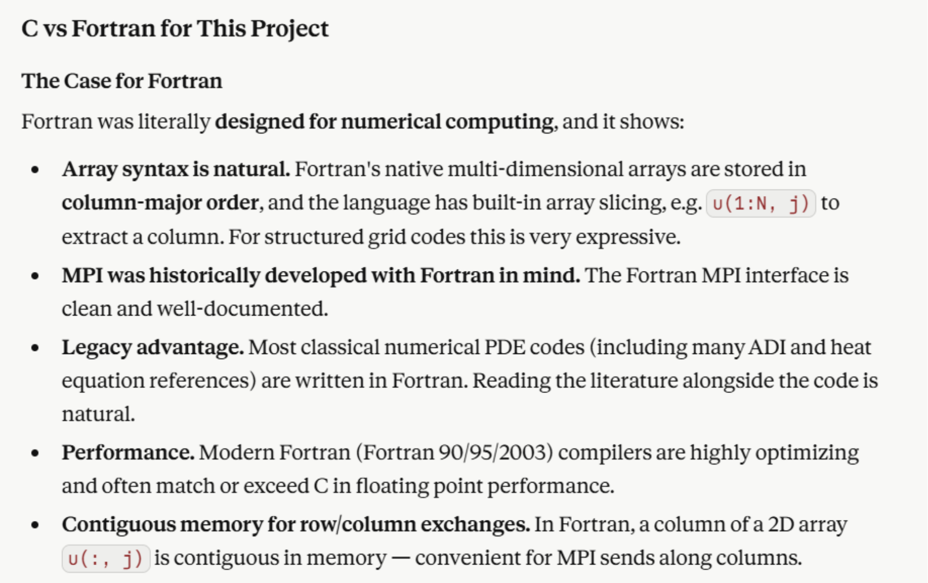
But sensible recommendations.

Implementation design

Claude: Shall I now move to the next design decision — the choice of programming language? Or would you like to discuss the decomposition further first?

Me: Excellent recommendation! Let's go with the 1D row-strip decomposition approach. Let's discuss the selection of programming language: C or Fortran, as I have a working knowledge of these two with MPI. What would you like to recommend and why?

Implementation design

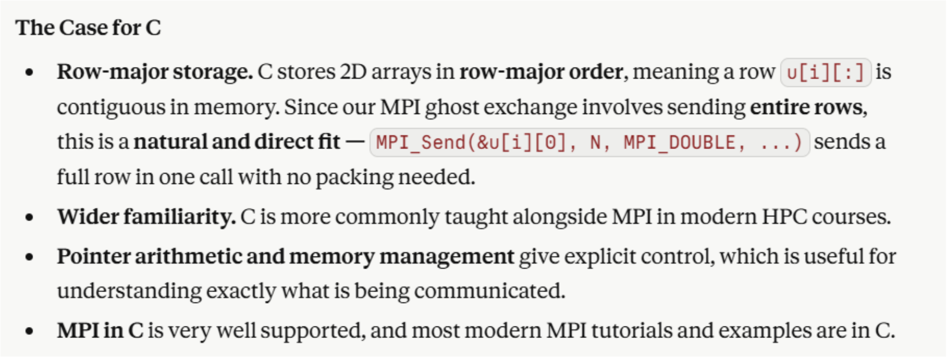


C vs Fortran for This Project

The Case for Fortran

Fortran was literally **designed for numerical computing**, and it shows:

- **Array syntax is natural.** Fortran's native multi-dimensional arrays are stored in **column-major order**, and the language has built-in array slicing, e.g. `u(1:N, j)` to extract a column. For structured grid codes this is very expressive.
- **MPI was historically developed with Fortran in mind.** The Fortran MPI interface is clean and well-documented.
- **Legacy advantage.** Most classical numerical PDE codes (including many ADI and heat equation references) are written in Fortran. Reading the literature alongside the code is natural.
- **Performance.** Modern Fortran (Fortran 90/95/2003) compilers are highly optimizing and often match or exceed C in floating point performance.
- **Contiguous memory for row/column exchanges.** In Fortran, a column of a 2D array `u(:, j)` is contiguous in memory — convenient for MPI sends along columns.



The Case for C

- **Row-major storage.** C stores 2D arrays in **row-major order**, meaning a row `u[i][:]` is contiguous in memory. Since our MPI ghost exchange involves sending **entire rows**, this is a **natural and direct fit** — `MPI_Send(&u[i][0], N, MPI_DOUBLE, ...)` sends a full row in one call with no packing needed.
- **Wider familiarity.** C is more commonly taught alongside MPI in modern HPC courses.
- **Pointer arithmetic and memory management** give explicit control, which is useful for understanding exactly what is being communicated.
- **MPI in C** is very well supported, and most modern MPI tutorials and examples are in C.

Implementation design

The Deciding Factor for Our Project

The key architectural decision we just made — **1D row-strip decomposition with ghost row exchange** — actually **favours C** for a subtle but important reason:

In C (row-major), each row of the 2D array `u[i][j]` is **contiguous in memory**. Sending a ghost row to a neighbour is therefore a single, efficient `MPI_Send` of N contiguous doubles:

c

```c
MPI_Send(&u[local_rows][0], N, MPI_DOUBLE, rank+1, ...); // send bottom ghost row
```

In Fortran (column-major), a **row** of a 2D array `u(i, j)` is **not contiguous** in memory — columns are contiguous. Sending a row would require either a manual loop to pack values into a buffer first, or defining an MPI derived datatype — adding complexity.

So:

Criterion	C	Fortran
Array storage order	Row-major ✅ natural for row exchange	Column-major ⚠ rows not contiguous
Ghost row MPI send	Direct, no packing needed ✅	Requires buffer or derived type ⚠
Numerical expressiveness	Good	Excellent
MPI support	Excellent	Excellent
Modern tooling & examples	Abundant	Good
Match with our 1D strip design	✅ Strong	⚠ Minor friction

Remarks:

The language selection based on memory access pattern is invalid.

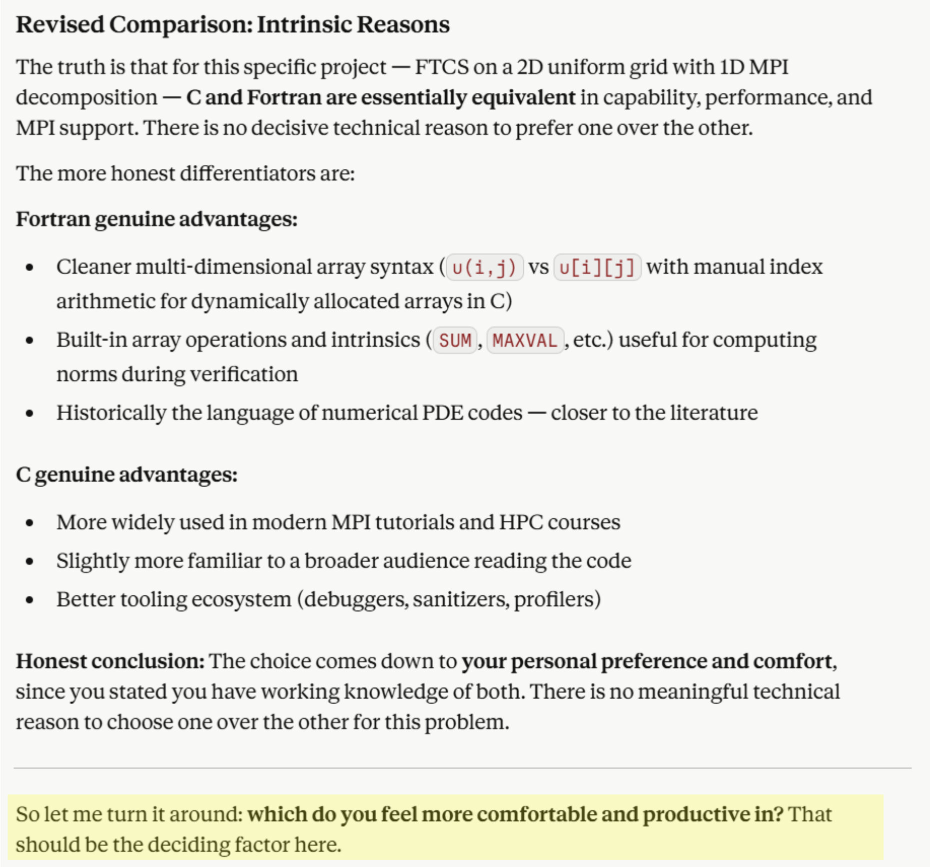
Implementation design

Me: Umm, the memory access pattern reason is not convincing, as you said, if we switch to column-strip, then the case flips, Fortran wins.

Claude: You are absolutely right, and I retract that argument. It was circular reasoning — I justified C based on a row-strip decomposition that we chose partly based on convenience, and the whole thing is self-referential. If we had chosen column-strip decomposition, Fortran would win by the same logic. The memory layout argument is not an intrinsic reason to prefer one language over the other.

Let me give you a more honest assessment.

Implementation design



Revised Comparison: Intrinsic Reasons

The truth is that for this specific project — FTCS on a 2D uniform grid with 1D MPI decomposition — **C and Fortran are essentially equivalent** in capability, performance, and MPI support. There is no decisive technical reason to prefer one over the other.

The more honest differentiators are:

Fortran genuine advantages:

- Cleaner multi-dimensional array syntax (`u(i,j)` vs `u[i][j]` with manual index arithmetic for dynamically allocated arrays in C)
- Built-in array operations and intrinsics (`SUM`, `MAXVAL`, etc.) useful for computing norms during verification
- Historically the language of numerical PDE codes — closer to the literature

C genuine advantages:

- More widely used in modern MPI tutorials and HPC courses
- Slightly more familiar to a broader audience reading the code
- Better tooling ecosystem (debuggers, sanitizers, profilers)

Honest conclusion: The choice comes down to **your personal preference and comfort**, since you stated you have working knowledge of both. There is no meaningful technical reason to choose one over the other for this problem.

So let me turn it around: **which do you feel more comfortable and productive in?** That should be the deciding factor here.

Me: Given that there is no meaningful technical reason to pick one over another and Fortran's simpler notation (e.g. u(i,j) vs u[i][j]), let's choose Fortran.

Claude: Perfect — and that is a completely sound reason. Notation clarity and developer comfort are legitimate and important practical considerations in any real project. Fortran it is!

Implementation design

Claude started proactively working on the procedure

Step 1 – Ghost row exchange (MPI).

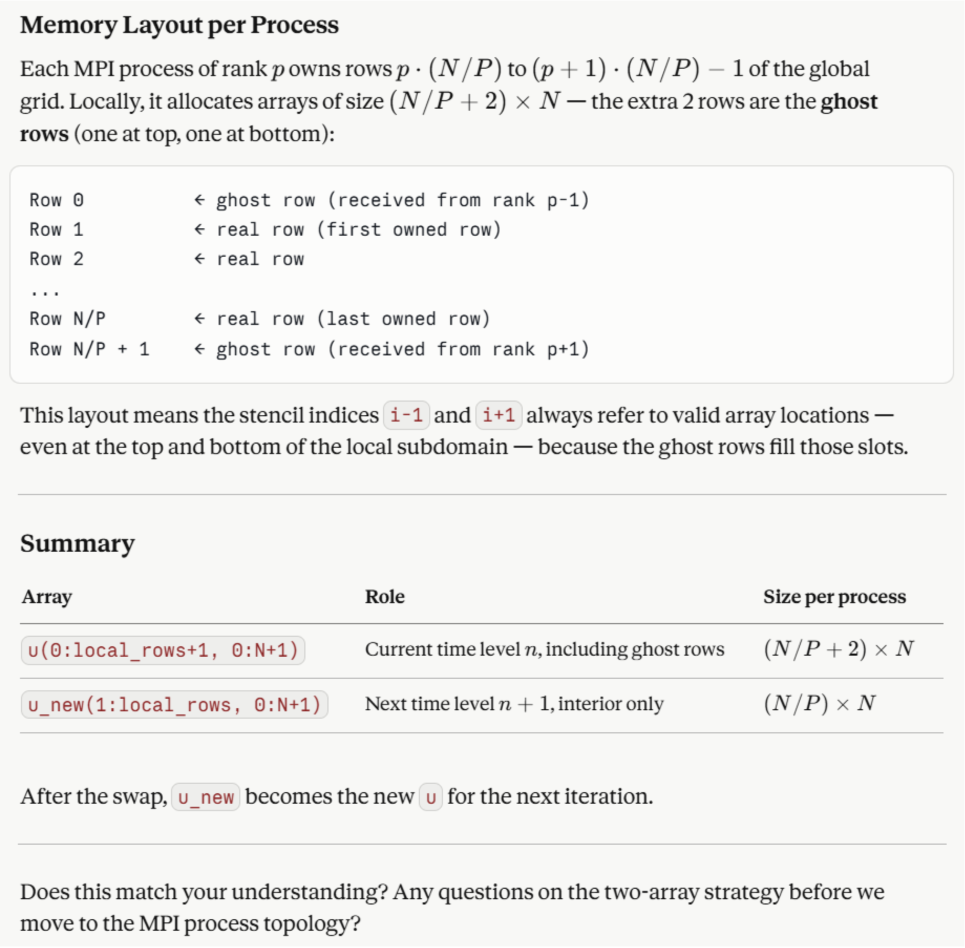
Step 2 – Computing interior update.

Step 3 – Apply boundary conditions.

Step 4 – Swap arrays holding u^{n+1} and u^n .

Step 5 – Advance time.

And it goes on with a wrong assumption – row strip decomposition of the domain.



Memory Layout per Process

Each MPI process of rank p owns rows $p \cdot (N/P)$ to $(p+1) \cdot (N/P) - 1$ of the global grid. Locally, it allocates arrays of size $(N/P+2) \times N$ — the extra 2 rows are the **ghost rows** (one at top, one at bottom):

```
Row 0           ← ghost row (received from rank p-1)
Row 1           ← real row (first owned row)
Row 2           ← real row
...
Row N/P         ← real row (last owned row)
Row N/P + 1     ← ghost row (received from rank p+1)
```

This layout means the stencil indices `i-1` and `i+1` always refer to valid array locations — even at the top and bottom of the local subdomain — because the ghost rows fill those slots.

Summary

Array	**Role**	**Size per process**
`u(0:local_rows+1, 0:N+1)`	Current time level n, including ghost rows	$(N/P+2) \times N$
`u_new(1:local_rows, 0:N+1)`	Next time level $n+1$, interior only	$(N/P) \times N$

After the swap, `u_new` becomes the new `u` for the next iteration.

Does this match your understanding? Any questions on the two-array strategy before we move to the MPI process topology?

Implementation design

Me: Wait, I think there is a flaw: we've decided to go with column-strips so the choice will be in favour of Fortran we've decided to use?

Claude: I need to gently push back here — with respect, I think there may be a mix-up. We actually decided on row-strip decomposition, not column-strip. Let me recap the exact decisions we made:

1. Row-strip decomposition was chosen in the domain decomposition discussion — each process owns a contiguous block of rows.
2. Fortran was chosen shortly after, but the reason given was notation clarity (u(i,j) vs u[i][j]), explicitly not memory layout. We both agreed that the memory layout argument was circular and was retracted.

So there is no inconsistency in our decisions. The choice of Fortran was made independently of the decomposition direction, on its own merits.

Remarks:

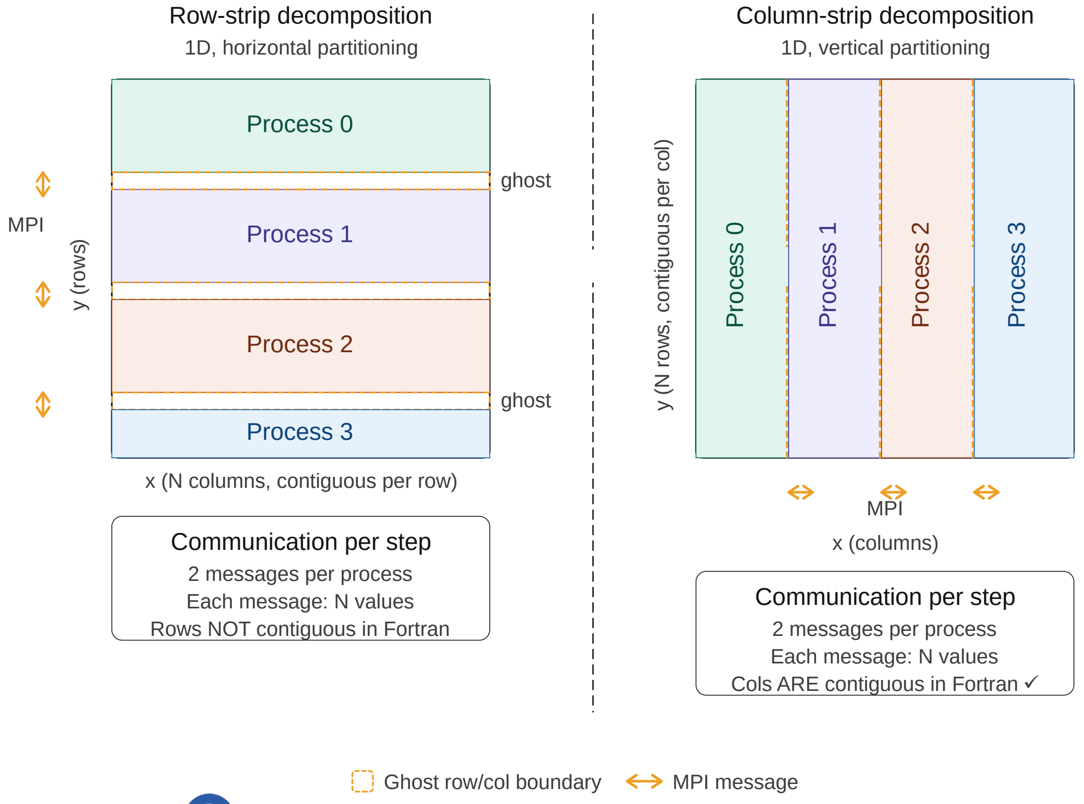
I made a mistake, I forgot I did say earlier “Let's go with the 1D row-strip decomposition approach.” Claude is correct indeed!

I decided not to retract what I said, but take another approach.

Implementation design

Me: Just to confirm so that we are both on the same page, would you be able to show a diagram for each?

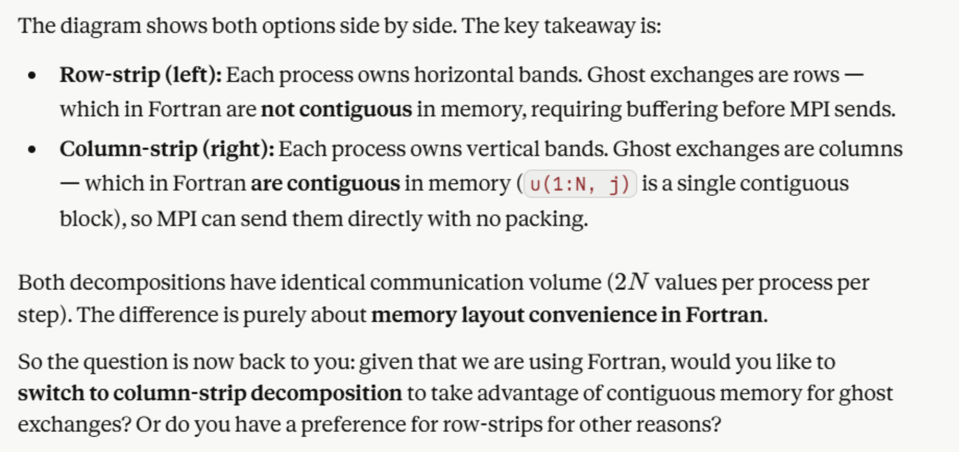
Claude: Absolutely — let me draw both clearly.



Communication per step
2 messages per process
Each message: N values
Rows NOT contiguous in Fortran

Communication per step
2 messages per process
Each message: N values
Cols ARE contiguous in Fortran ✓

Implementation design



The diagram shows both options side by side. The key takeaway is:

- **Row-strip (left):** Each process owns horizontal bands. Ghost exchanges are rows — which in Fortran are **not contiguous** in memory, requiring buffering before MPI sends.
- **Column-strip (right):** Each process owns vertical bands. Ghost exchanges are columns — which in Fortran **are contiguous** in memory (`u(1:N, j)` is a single contiguous block), so MPI can send them directly with no packing.

Both decompositions have identical communication volume ($2N$ values per process per step). The difference is purely about **memory layout convenience in Fortran**.

So the question is now back to you: given that we are using Fortran, would you like to **switch to column-strip decomposition** to take advantage of contiguous memory for ghost exchanges? Or do you have a preference for row-strips for other reasons?

Me: Perfect! Lets go with 2: Switch to column-strip + Fortran

Implementation design

Claude: The next design decisions remaining in Task 3 are:

1. Code architecture — module and subroutine structure of the Fortran program
2. MPI process topology — how ranks are laid out and how ghost column exchange is coded
3. I/O strategy — how we write output for verification and visualization

Shall we proceed with the code architecture?

Me: I propose to have the following components at a high level

1. input
2. set boundary condition
3. set initial condition
4. solver
5. output for verification/visualization

The input and output can go into the same module io, if that makes sense.

Implementation design

```
program diffusion_2d
  use mpi
  use io_module
  use grid_module
  use solver_module

  call mpi_setup()         ! initialize MPI, determine rank and domain partition
  call read_input()        ! read parameters: N, a, alpha, t_end
  call allocate_arrays()   ! allocate u and u_new with ghost columns
  call set_ic()            ! u = phi(x,y) at t=0
  call set_bc(u, t=0.0)    ! u = 0 on boundaries

  do while (t < t_end)
    call ghost_exchange(u) ! MPI: swap ghost columns with neighbours
    call ftcs_update()     ! u_new = FTCS(u)
    call set_bc(u_new, t)  ! enforce BCs on updated solution
    call swap(u, u_new)    ! u <- u_new
    t = t + dt
    if (output_due) call write_output(u, t)
  end do

  call write_final(u, t)   ! final output for verification
  call mpi_finalize()
end program
```

Claude comes back with this.

This is the main program. It shows a clear architecure that has been discussed.

The time integration part is explicit in the main program, although it could be in a function too.

Two arrays u and u_new are used. They are swapped after each time step.

The details of each component yet need to be worked out individually.

Implementation design

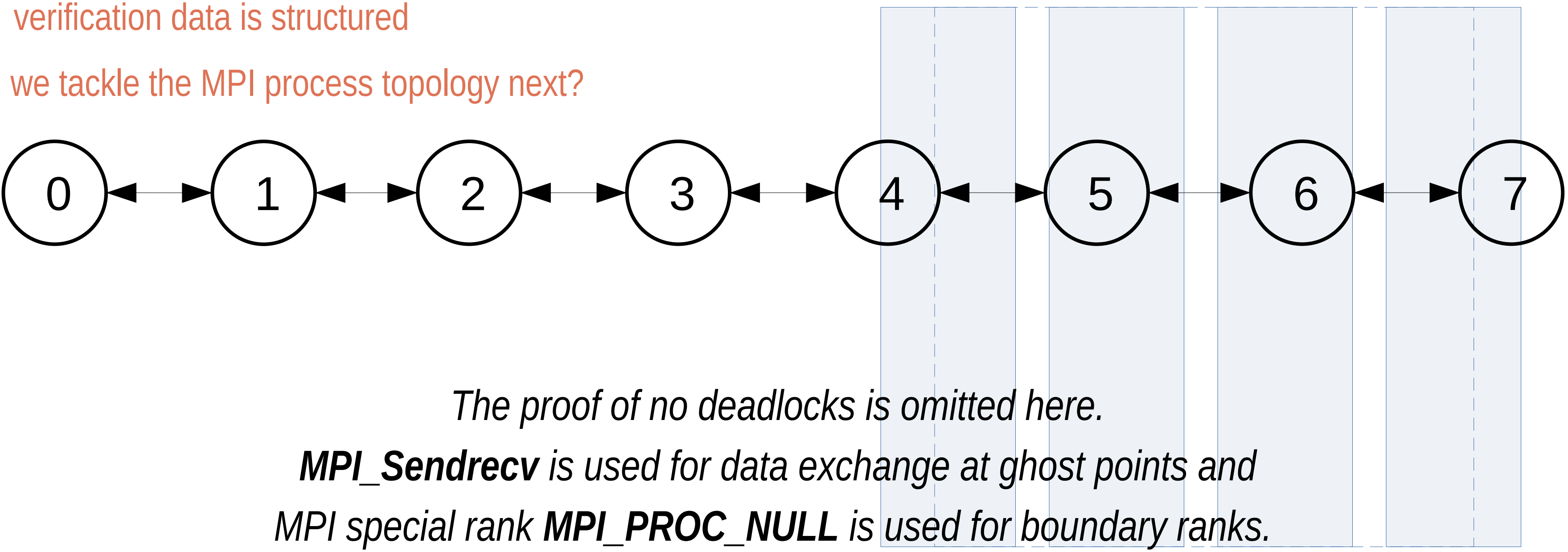
Claude: We have two remaining design items in Task 3:

1. MPI process topology — how ranks are numbered, how each rank determines its column range, and how ghost column exchanges are coded
2. I/O strategy — file format, what gets written, and how verification data is structured

Shall we tackle the MPI process topology next?

Me: Yes. If making sense, I'd number the MPI ranks in the column-strips from left to right with 0 to P-1.

If I understand correctly, we have a chain module with bidirectional communications. Can it be proven that with MPI_Send/Recv calls, there will be no deadlocks?



The proof of no deadlocks is omitted here.

***MPI_Sendrecv** is used for data exchange at ghost points and*

*MPI special rank **MPI_PROC_NULL** is used for boundary ranks.*

Implementation design

Input parameters

- a – diffusion constant.
- N_x – number of grid points in x.
- N_y – number of grid points in y.
- Δt – time step (-1 flag to computer from mesh Fourier numbers).
- r_x – mesh Fourier numner $a\Delta t/\Delta x^2$.
- r_y – mesh Fourier numner $a\Delta t/\Delta y^2$.
- t_{end} – final simulation time.
- n_{out} – output frequency (every how many steps).

Me: Shall we set to number of time steps or the final simulation time? Could you make a suggestion?

Claude: My recommendation is final simulation time t_{end} , for the following reasons:

- Physical meaning: t_{end} has a direct physical interpretation — it tells you how far the diffusion has evolved. n_{steps} is a numerical artifact that depends on Δt.
- Consistency with your input logic: Since Δt may be either given or computed, specifying t_{end} keeps the input parameters independent of each other.
- Verification friendliness: For Task 5, comparing against the exact solution $u(x,y,t)=\sin(\pi x)\sin(\pi y)\exp(-2a\pi^2 t)$ at a specific physical time t is natural and meaningful.

Task 4: Code implementation

Code implementation

Claude quickly completed the source code

- main.f90
- io_module.f90
  - read_input
  - write_output
  - write_final
- grid_module.f90
  - allocate_arrays
  - set_ic
  - set_bc
- solver_module.f90
  - ghost_exchange
  - ftcs_update
  - swap

Summary

- 800+ lines of code.
- Four source files.
- A Make file.
- An input file.

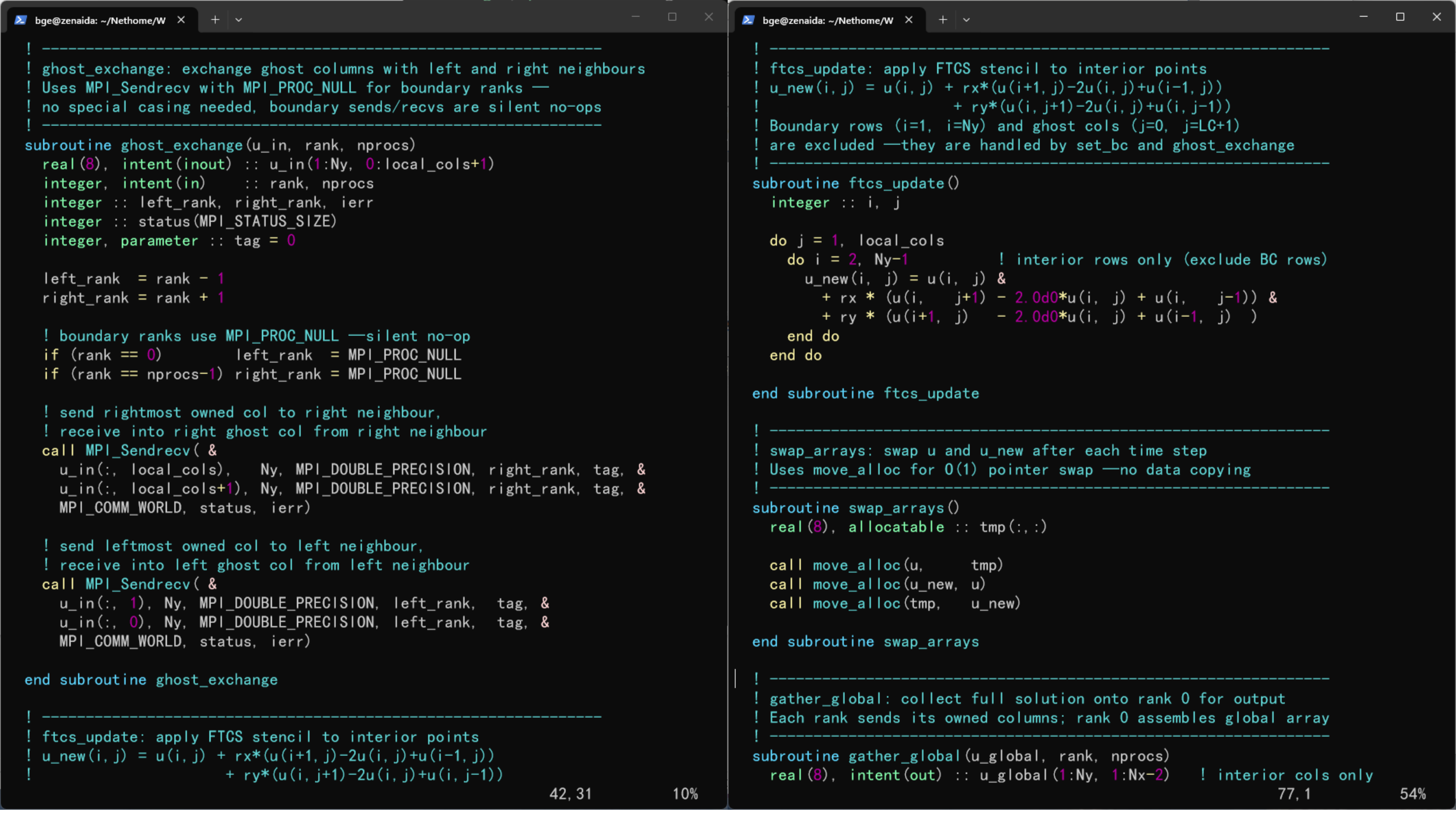
Code implementation

Claude also recommended a Fortran built-in key=value pair format for input, a hidden gem I didn’t know

```
&params
  a      = 1.0          ! diffusion constant
  Nx     = 141          ! grid points in x (includes boundaries)
  Ny     = 141          ! grid points in y (includes boundaries)
  dt     = -1.0         ! <= 0: compute dt from rx, ry below
  rx     = 0.20         ! mesh Fourier number in x: a*dt/dx^2
  ry     = 0.20         ! mesh Fourier number in y: a*dt/dy^2
  t_end  = 0.05         ! final simulation time
  n_out  = 10           ! output frequency (steps)
/
```

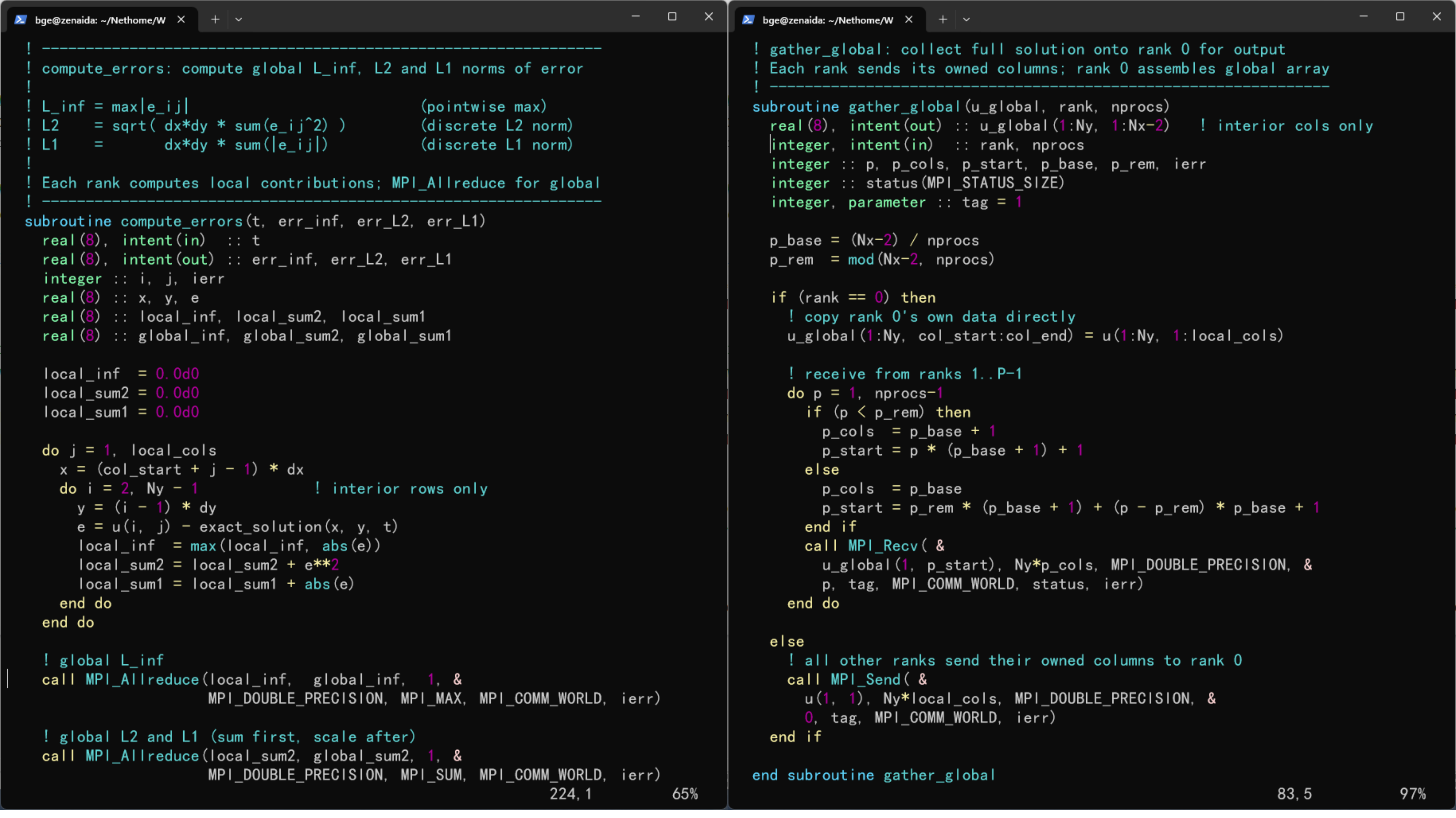
Use of namelist in Fortran

```
real(8) :: a          ! diffusion constant
integer :: Nx, Ny     ! grid points in x and y
real(8) :: dt         ! time step
real(8) :: rx, ry     ! mesh Fourier numbers: a*dt/dx^2, a*dt/dy^2
real(8) :: dx, dy     ! grid spacings
real(8) :: t_end      ! final simulation time
integer :: n_out      ! output frequency (every n_out steps)
namelist /params/ a, Nx, Ny, dt, rx, ry, t_end, n_out
```



```
! ----------------------------------------------------------------
! ghost_exchange: exchange ghost columns with left and right neighbours
! Uses MPI_Sendrecv with MPI_PROC_NULL for boundary ranks —
! no special casing needed, boundary sends/recvs are silent no-ops
! ----------------------------------------------------------------
subroutine ghost_exchange(u_in, rank, nprocs)
  real(8), intent(inout) :: u_in(1:Ny, 0:local_cols+1)
  integer, intent(in)    :: rank, nprocs
  integer :: left_rank, right_rank, ierr
  integer :: status(MPI_STATUS_SIZE)
  integer, parameter :: tag = 0

  left_rank  = rank - 1
  right_rank = rank + 1

  ! boundary ranks use MPI_PROC_NULL —silent no-op
  if (rank == 0)        left_rank  = MPI_PROC_NULL
  if (rank == nprocs-1) right_rank = MPI_PROC_NULL

  ! send rightmost owned col to right neighbour,
  ! receive into right ghost col from right neighbour
  call MPI_Sendrecv( &
    u_in(:, local_cols),   Ny, MPI_DOUBLE_PRECISION, right_rank, tag, &
    u_in(:, local_cols+1), Ny, MPI_DOUBLE_PRECISION, right_rank, tag, &
    MPI_COMM_WORLD, status, ierr)

  ! send leftmost owned col to left neighbour,
  ! receive into left ghost col from left neighbour
  call MPI_Sendrecv( &
    u_in(:, 1), Ny, MPI_DOUBLE_PRECISION, left_rank,  tag, &
    u_in(:, 0), Ny, MPI_DOUBLE_PRECISION, left_rank,  tag, &
    MPI_COMM_WORLD, status, ierr)

end subroutine ghost_exchange

! ----------------------------------------------------------------
! ftcs_update: apply FTCS stencil to interior points
! u_new(i,j) = u(i,j) + rx*(u(i+1,j)-2u(i,j)+u(i-1,j))
!                      + ry*(u(i,j+1)-2u(i,j)+u(i,j-1))
                                                      42,31         10%
```

```
! ----------------------------------------------------------------
! ftcs_update: apply FTCS stencil to interior points
! u_new(i,j) = u(i,j) + rx*(u(i+1,j)-2u(i,j)+u(i-1,j))
!                      + ry*(u(i,j+1)-2u(i,j)+u(i,j-1))
! Boundary rows (i=1, i=Ny) and ghost cols (j=0, j=LC+1)
! are excluded —they are handled by set_bc and ghost_exchange
! ----------------------------------------------------------------
subroutine ftcs_update()
  integer :: i, j

  do j = 1, local_cols
    do i = 2, Ny-1           ! interior rows only (exclude BC rows)
      u_new(i, j) = u(i, j) &
        + rx * (u(i,   j+1) - 2.0d0*u(i, j) + u(i,   j-1)) &
        + ry * (u(i+1, j)   - 2.0d0*u(i, j) + u(i-1, j)  )
    end do
  end do

end subroutine ftcs_update

! ----------------------------------------------------------------
! swap_arrays: swap u and u_new after each time step
! Uses move_alloc for O(1) pointer swap —no data copying
! ----------------------------------------------------------------
subroutine swap_arrays()
  real(8), allocatable :: tmp(:,:)

  call move_alloc(u,     tmp)
  call move_alloc(u_new, u)
  call move_alloc(tmp,   u_new)

end subroutine swap_arrays

! ----------------------------------------------------------------
! gather_global: collect full solution onto rank 0 for output
! Each rank sends its owned columns; rank 0 assembles global array
! ----------------------------------------------------------------
subroutine gather_global(u_global, rank, nprocs)
  real(8), intent(out) :: u_global(1:Ny, 1:Nx-2)   ! interior cols only
                                                      77,1          54%
```



```fortran
! ------------------------------------------------------------------
! compute_errors: compute global L_inf, L2 and L1 norms of error
!
! L_inf = max|e_ij|                          (pointwise max)
! L2    = sqrt( dx*dy * sum(e_ij^2) )        (discrete L2 norm)
! L1    =       dx*dy * sum(|e_ij|)          (discrete L1 norm)
!
! Each rank computes local contributions; MPI_Allreduce for global
! ------------------------------------------------------------------
subroutine compute_errors(t, err_inf, err_L2, err_L1)
  real(8), intent(in)  :: t
  real(8), intent(out) :: err_inf, err_L2, err_L1
  integer :: i, j, ierr
  real(8) :: x, y, e
  real(8) :: local_inf, local_sum2, local_sum1
  real(8) :: global_inf, global_sum2, global_sum1

  local_inf  = 0.0d0
  local_sum2 = 0.0d0
  local_sum1 = 0.0d0

  do j = 1, local_cols
    x = (col_start + j - 1) * dx
    do i = 2, Ny - 1             ! interior rows only
      y = (i - 1) * dy
      e = u(i, j) - exact_solution(x, y, t)
      local_inf  = max(local_inf, abs(e))
      local_sum2 = local_sum2 + e**2
      local_sum1 = local_sum1 + abs(e)
    end do
  end do

  ! global L_inf
  call MPI_Allreduce(local_inf,  global_inf,  1, &
                     MPI_DOUBLE_PRECISION, MPI_MAX, MPI_COMM_WORLD, ierr)

  ! global L2 and L1 (sum first, scale after)
  call MPI_Allreduce(local_sum2, global_sum2, 1, &
                     MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, ierr)
                                                            224,1         65%
```

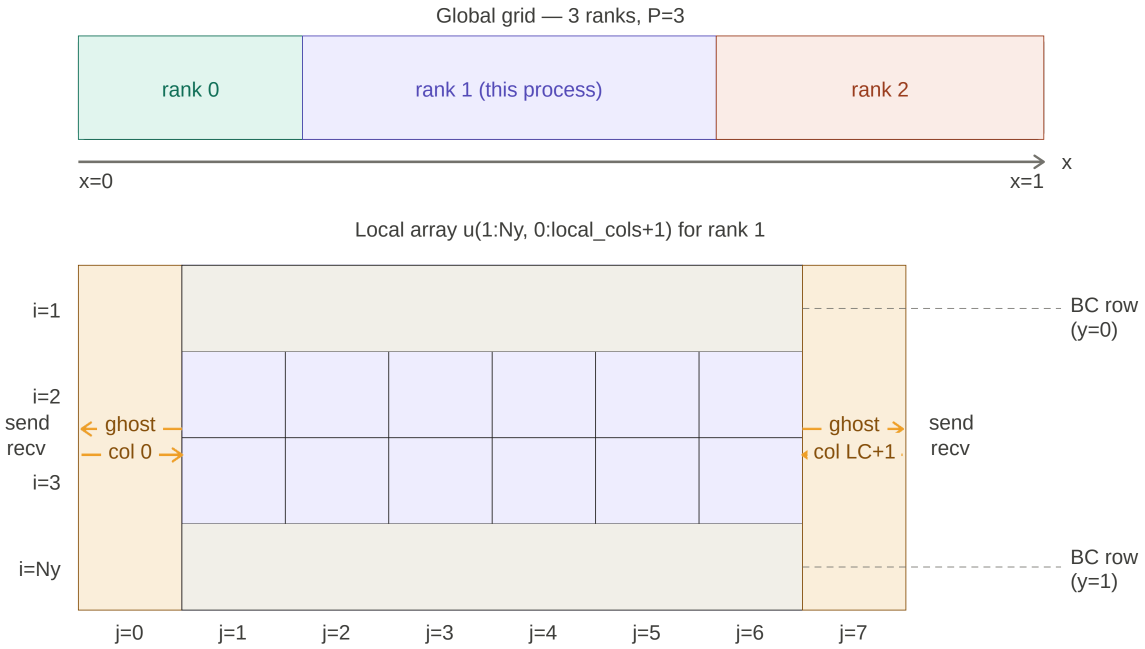
```fortran
! gather_global: collect full solution onto rank 0 for output
! Each rank sends its owned columns; rank 0 assembles global array
! ------------------------------------------------------------------
subroutine gather_global(u_global, rank, nprocs)
  real(8), intent(out) :: u_global(1:Ny, 1:Nx-2)   ! interior cols only
  integer, intent(in)  :: rank, nprocs
  integer :: p, p_cols, p_start, p_base, p_rem, ierr
  integer :: status(MPI_STATUS_SIZE)
  integer, parameter :: tag = 1

  p_base = (Nx-2) / nprocs
  p_rem  = mod(Nx-2, nprocs)

  if (rank == 0) then
    ! copy rank 0's own data directly
    u_global(1:Ny, col_start:col_end) = u(1:Ny, 1:local_cols)

    ! receive from ranks 1..P-1
    do p = 1, nprocs-1
      if (p < p_rem) then
        p_cols  = p_base + 1
        p_start = p * (p_base + 1) + 1
      else
        p_cols  = p_base
        p_start = p_rem * (p_base + 1) + (p - p_rem) * p_base + 1
      end if
      call MPI_Recv( &
        u_global(1, p_start), Ny*p_cols, MPI_DOUBLE_PRECISION, &
        p, tag, MPI_COMM_WORLD, status, ierr)
    end do

  else
    ! all other ranks send their owned columns to rank 0
    call MPI_Send( &
      u(1, 1), Ny*local_cols, MPI_DOUBLE_PRECISION, &
      0, tag, MPI_COMM_WORLD, ierr)
  end if

end subroutine gather_global
                                                            83,5          97%
```

Task 5: Code verification and visualization

Code verification and visualization

Lots of of details. So I asked Claude to draw a diagram to better understand the code

Column-strip subdomain: local array layout (rank p)



- 🟪 owned interior columns u(1:Ny, 1:local_cols)
- 🟧 ghost columns u(1:Ny, 0) and u(1:Ny, local_cols+1)
- ⬜ boundary rows u(1, :) and u(Ny, :) — set by set_bc
- → MPI_Sendrecv — ghost column exchange each time step

Code verification and visualization

By checking the key concepts inplemented, I discovered a flaw at the line

local_cols = n_interior_cols / nprocs

This assumes the number of integrior grid points can be evenly divided, which is generally untrue.

Claude fixed it with a balancing algorithm.

Code verification and visualization

First time to compile and run the code. Is it bug free? No!

```
  step=    1150  t= 9.200E-02  err_inf= 1.057E-02  err_rms= 7.153E-03
  step=    1200  t= 9.600E-02  err_inf= 9.937E-03  err_rms= 6.821E-03
At line 88 of file main.f90
Fortran runtime error: Allocatable actual argument 'u_global' is not allocated
Error termination. Backtrace:
At line 88 of file main.f90
Fortran runtime error: Allocatable actual argument 'u_global' is not allocated
Error termination. Backtrace:
At line 88 of file main.f90
Fortran runtime error: Allocatable actual argument 'u_global' is not allocated
Error termination. Backtrace:
  step=    1250  t= 1.000E-01  err_inf= 9.375E-03  err_rms= 6.498E-03
```

Memory not allocated on non zero ranks.

Claude: This occurs at line 88 of main.f90 — the call to gather_global(u_global, ...). The problem is clear: only rank 0 allocates u_global, but all ranks call gather_global — and Fortran's runtime checks (-fcheck=all) correctly flag that non-zero ranks pass an unallocated array.

The fix is straightforward — allocate a zero-size dummy array on non-zero ranks so the argument is always allocated:

Details omitted

Code verification and visualization

Tried the fixed code, different error!

```
  step=    1200  t= 9.600E-02  err_inf= 9.937E-03  err_rms= 6.821E-03
  step=    1250  t= 1.000E-01  err_inf= 9.375E-03  err_rms= 6.498E-03
At line 152 of file io_module.f90
Fortran runtime error: Index '51' of dimension 1 of array 'u' above upper bound of
50
```

Out-of-bounds writes

Claude: The error is now Index '51' of dimension 1 — so it's the row index i going out of bounds, not the column. The loop i = 1, Ny tries to access row Ny=51 but the array is declared u(1:Ny, ...) which should allow i=51.

Wait — looking more carefully at write_final, the dummy argument is declared as u(0:, 0:) which means dimension 1 starts at 0 and has size equal to whatever is passed. When we pass u_global(1:Ny, 1:Nx-2), the lower bound of dimension 1 inside write_final is remapped to 0, so the upper bound becomes Ny-1 = 50. Hence i=51 (from i=1,Ny) goes out of bounds.

The fix is to correct the dummy argument declaration in write_final to match exactly how u_global is shaped:

Details omitted

The bug is fixed and code runs successfully.

Code verification and visualization

The next is to verify the correctness of the code. Claude suggested:

- Pointwise comparison at every grid point, define $e_{i,j}^{n} = u_{i,j}^{n} - u_{true}(x_i, y_j, t^n)$
  - Using max (infinity) norm: $\|e\|_\infty = \max_{i,j} |e_{i,j}^{n}|$
  - Using RMS error (normalized): $\|e\|_2 = \sqrt{\frac{1}{N_x N_y} \sum_{i,j} (e_{i,j}^{n})^2}$
- Comparison at selected sampling points.
- Convergence study (grid refinement).
- Since global errors are calculated, MPI_Allreduce is used to collect the partial sums with MPI_MAX amnd MPI_SUM on the root rank.

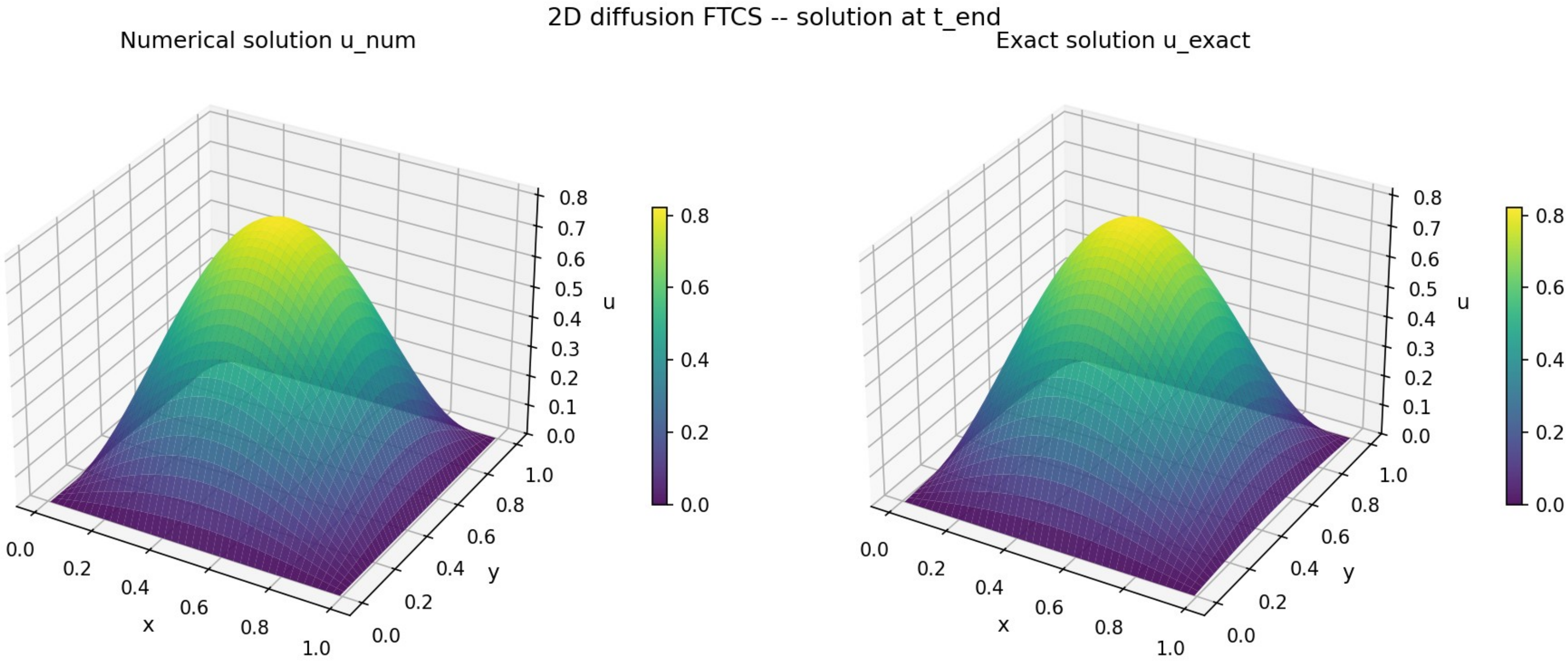
Code verification and visualization

I pointed out we should use (discrete approximation to) continuous norms instead of vector norms

- Define $e_{i,j} = u_{i,j} - u_{true}(x_i, y_j, t^n)$
  - Using max (infinity) norm: $\|e\|_\infty = \max_{i,j} |e_{i,j}|$
  - Using discrete L^2 norm: $\|e\|_{L^2} = \sqrt{\int_\Omega^2 e_{i,j}^2 dxdy} = \sqrt{\Delta x \Delta y \sum_{i,j} e_{i,j}^2}$

Code verification and visualization

Test 1: Initial condition 1 - $\sin(\pi x)\sin(\pi y)\exp(-2a\pi^2 t)$



Solution plots generated by a standalone Python script written by Claude.

Code verification and visualization

Test 2: Initial condition 2 - volcano-crater

Animation of numerical solution generated by a Python script written by Claude.

Code verification and visualization

Test 3: Image derived initial condition

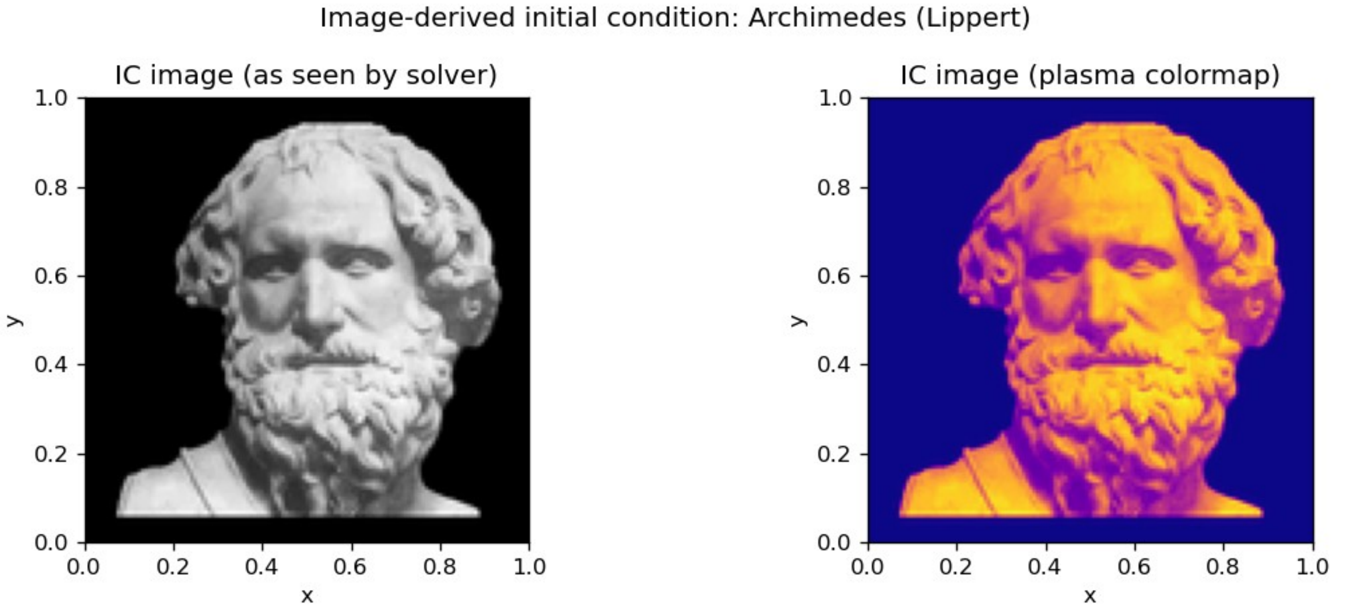
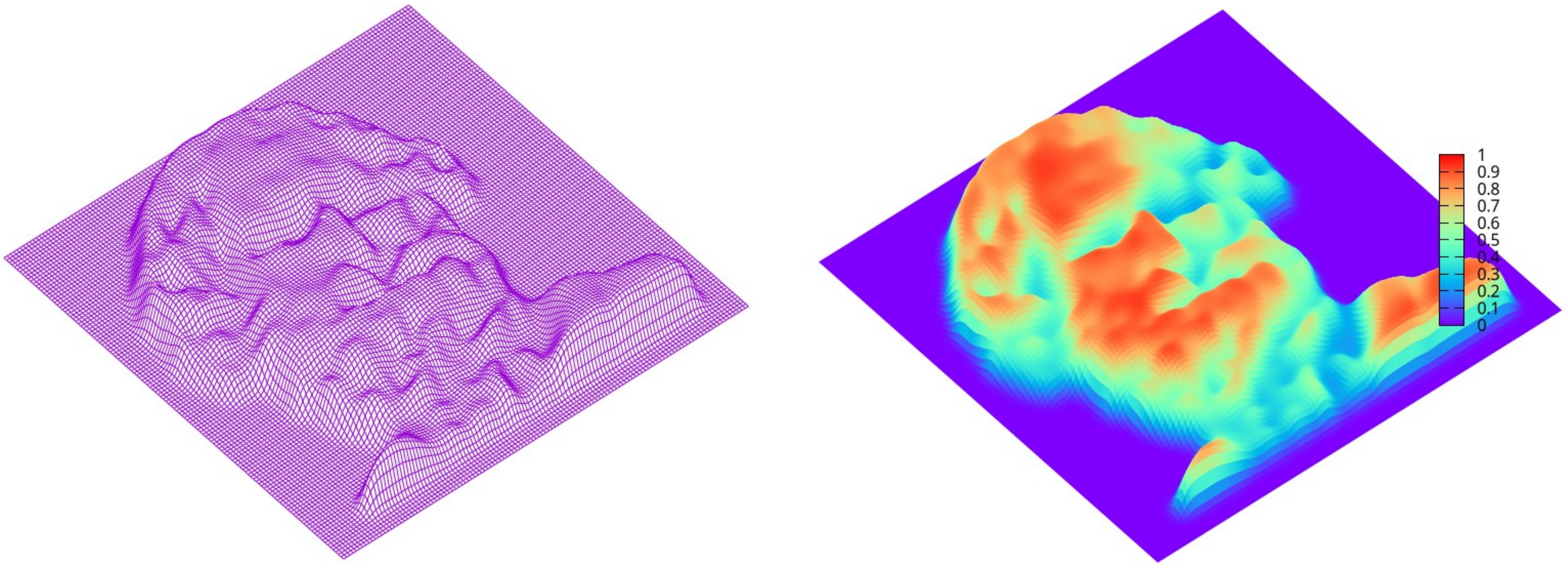


Image derived IC implemented by Claude with a Python script.

Code verification and visualization

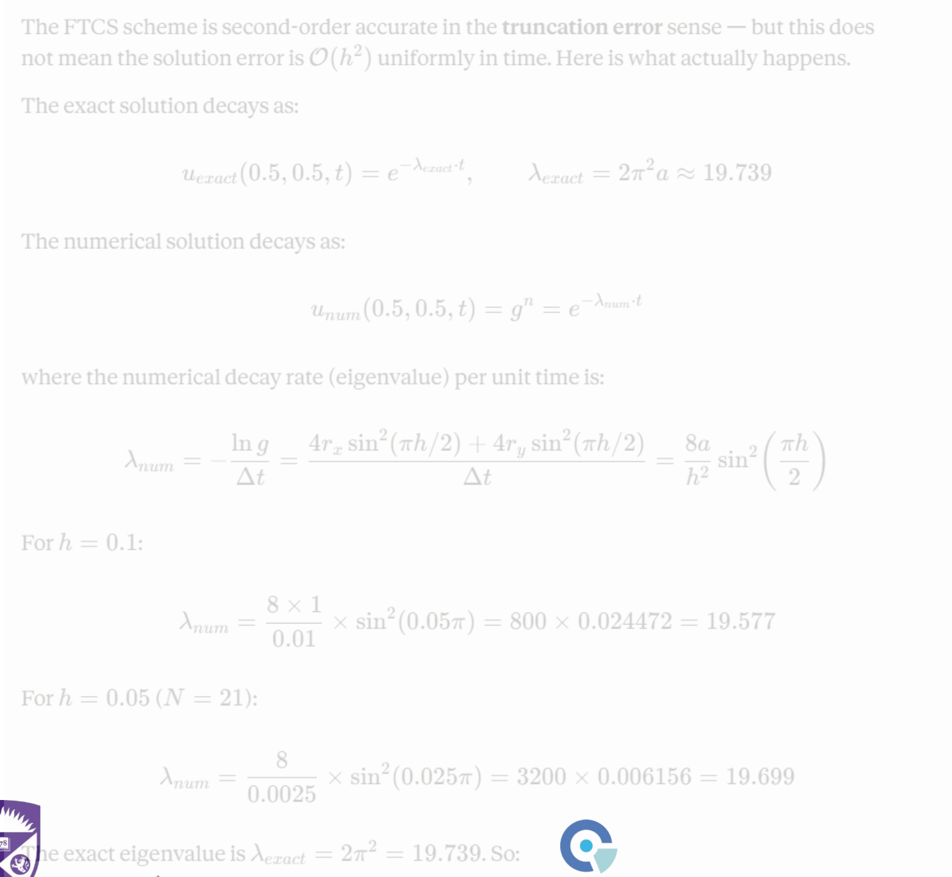
Test 3: Image derived initial condition (con’td)



Animation of the solution generated by Gnuplot by the author.

Code verification and visualization

Besides the numerical solution and visualization achieved, Claude did a whole whack of convergence analysis. The content is beyond the scope of this talk and omitted.



The FTCS scheme is second-order accurate in the **truncation error** sense — but this does not mean the solution error is $\mathcal{O}(h^2)$ uniformly in time. Here is what actually happens.

The exact solution decays as:

$$u_{exact}(0.5, 0.5, t) = e^{-\lambda_{exact} \cdot t}, \qquad \lambda_{exact} = 2\pi^2 a \approx 19.739$$

The numerical solution decays as:

$$u_{num}(0.5, 0.5, t) = g^n = e^{-\lambda_{num} \cdot t}$$

where the numerical decay rate (eigenvalue) per unit time is:

$$\lambda_{num} = -\frac{\ln g}{\Delta t} = \frac{4r_x \sin^2(\pi h/2) + 4r_y \sin^2(\pi h/2)}{\Delta t} = \frac{8a}{h^2} \sin^2\left(\frac{\pi h}{2}\right)$$

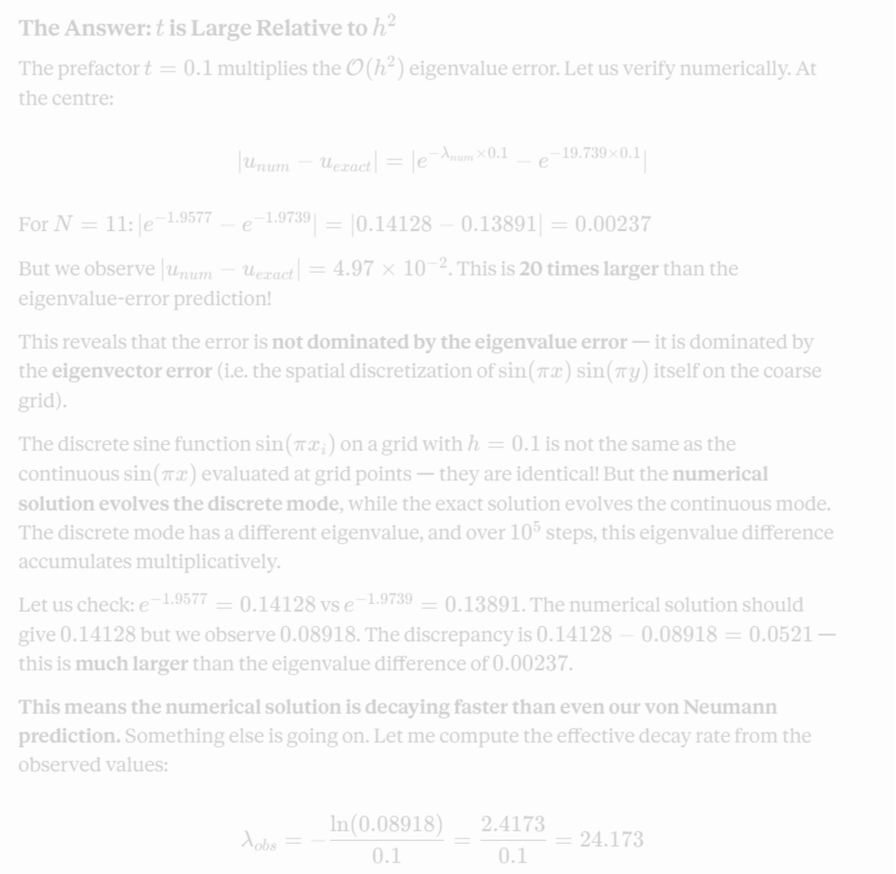
For $h = 0.1$:

$$\lambda_{num} = \frac{8 \times 1}{0.01} \times \sin^2(0.05\pi) = 800 \times 0.024472 = 19.577$$

For $h = 0.05$ ($N = 21$):

$$\lambda_{num} = \frac{8}{0.0025} \times \sin^2(0.025\pi) = 3200 \times 0.006156 = 19.699$$

The exact eigenvalue is $\lambda_{exact} = 2\pi^2 = 19.739$. So:



The Answer: t is Large Relative to h^2

The prefactor $t = 0.1$ multiplies the $\mathcal{O}(h^2)$ eigenvalue error. Let us verify numerically. At the centre:

$$|u_{num} - u_{exact}| = |e^{-\lambda_{num} \times 0.1} - e^{-19.739 \times 0.1}|$$

For $N = 11$: $|e^{-1.9577} - e^{-1.9739}| = |0.14128 - 0.13891| = 0.00237$

But we observe $|u_{num} - u_{exact}| = 4.97 \times 10^{-2}$. This is **20 times larger** than the eigenvalue-error prediction!

This reveals that the error is **not dominated by the eigenvalue error** — it is dominated by the **eigenvector error** (i.e. the spatial discretization of $\sin(\pi x)\sin(\pi y)$ itself on the coarse grid).

The discrete sine function $\sin(\pi x_i)$ on a grid with $h = 0.1$ is not the same as the continuous $\sin(\pi x)$ evaluated at grid points — they are identical! But the **numerical solution evolves the discrete mode**, while the exact solution evolves the continuous mode. The discrete mode has a different eigenvalue, and over 10^5 steps, this eigenvalue difference accumulates multiplicatively.

Let us check: $e^{-1.9577} = 0.14128$ vs $e^{-1.9739} = 0.13891$. The numerical solution should give 0.14128 but we observe 0.08918. The discrepancy is $0.14128 - 0.08918 = 0.0521$ — this is **much larger** than the eigenvalue difference of 0.00237.

This means the numerical solution is decaying faster than even our von Neumann prediction. Something else is going on. Let me compute the effective decay rate from the observed values:

$$\lambda_{obs} = -\frac{\ln(0.08918)}{0.1} = \frac{2.4173}{0.1} = 24.173$$

Concluding remarks

- Claude is good at information search (not surprising).
- It can make recommendations.
- It can derive some mathematical formula (comparable to computer algebra systems).
- It can do software architecture design.
- It can translate the mathematical formulation into computer language and generate complicated parallel code, e.g. using MPI functions in a sophisticated way.
- The generated code is not bug free. It makes the same, common mistakes in array indexing that human makes (why?).
- It can fix bugs (little surprising).
- It comes up with additional thoughts that the human supervisor did not think about and makes sensible suggestions (little inspiring).
- Overall Claude acts like a senior research assistant and a productive programmer.

Observations and thoughts

- The problem solved has been studied extensively. Can Claude and other AI tools tackle less well known, or unknown problems well?
- Claude shows it has certain level of “intelligence”, one needs to prompt and harness it properly in order to achieve the goals.
- Using AI tools like Claude can greatly increase one’s productivity.
- But one (still) faces challenges of reading references, checking formula, reasoning, etc.
- With AI tools – yet another set of skills to learn – researchers now can focus more on their actual scientific domains.
- Standing on the shoulder of giants, human research assistants are also getting smarter and more productive than ever before.

Reference

1. K. W. Morton, D. F. Mayers, ***Numerical Solution of Partial Differential Equations***, 2nd Ed., Cambridge University Press, 2005.
2. J. C. Strikwerda, ***Finite Difference Schemes and Partial Differential Equations***, Chapman and Hall, 1990.
3. J. Crank, P. Nicolson, “A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type”, *Proc. Camb. Philos. Soc.*, Vol. 43, Iss. 1, 1947.
4. D. W. Peaceman, H.H. Rachford, “The Numerical solution of parabolic and elliptic differential equations”, *J. Soc. Indust. Appl. Math.*, Vol. 3, No. 1, 1955.
5. J. Fourier, ***Théorie analytique de la chaleur****, Chapitre IX: De la diffusion de la chaleur*, 1822, Cambridge Univeristy Press, 2010.
6. W. Gropp *et al*, ***Using MPI***, 3rd Ed., The MIT Press, 2014.
7. SHARCNET, ***Introduction to MPI for distributed computing:*** *Lecture 4: Case study, 2D diffusion equation*, 2025-2026 Introduction to ARC Courses, https://training.sharcnet.ca/.
8. C++ Reference, https://www.cppreference.com/
9. M. Metcalf, J. Reid, M. Cohen, ***Modern Fortran Explained***, Oxford University Press, 2014.
10. Claude by Anthropic, https://claude.ai/.