Using Generative AI for Coding Tasks in Scientific Software

ANSHU DUBEY & AKASH DHRUV

Mathematics and Computer Science,
Argonne National Laboratory,
Lemont, IL

JULY 9, 2025

WEBINAR SERIES *BEST PRACTICES FOR HPC SOFTWARE DEVELOPERS*

Argonne NATIONAL LABORATORY

U.S. DEPARTMENT OF ENERGY | Office of Science

Acknowledgements

- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), under the SciDAC Projects RAPIDS, Neucol, and ENAF
- This work was done in the Mathematics and Computer Science Division at Argonne National Laboratory

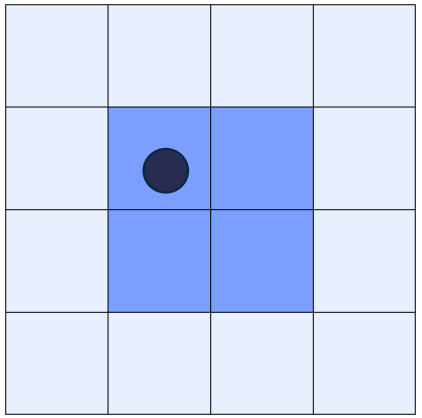
Motivation

Every code related task involves some tedium

- One would ideally like an assistant who handles all the tedious tasks
  - LLMs are supposedly such assistants
  - Many claims about increased productivity in software development
- We wanted to explore the use of LLMs in scientific code development in two scenarios
  - One is more for fun, and therefore optional
  - The other was nothing but tedium, so any assistance would help
- This presentation is about two use cases
  - New code for a new communication algorithm
  - Code translation of a legacy Fortran code to C++
- We share our experience and observations

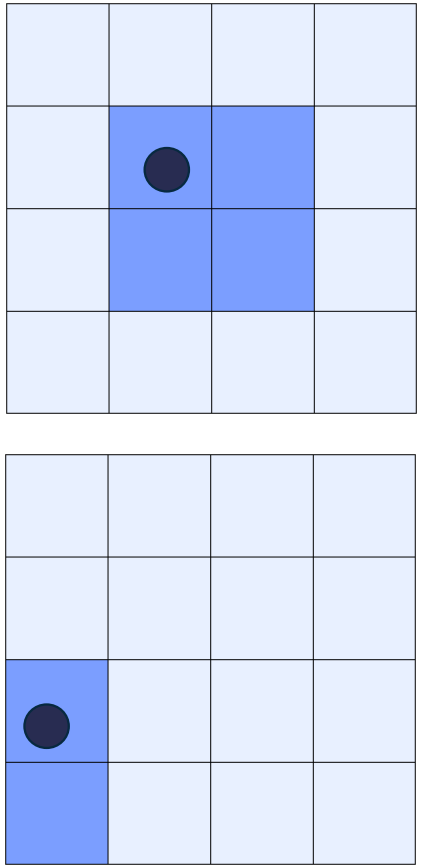
Use-case 1 – Developing New Code

- Massive particles for N-body methods used in astrophysics
  - The code has both mesh and particles
    - In parallel codes mesh is divided into blocks that are distributed among processors
  - Particles carry mass, mesh carries density
  - Particles deposit density onto the mesh
  - Mesh computes gravitational potential, forces and acceleration
  - Forces and acceleration are conveyed back to particles who move to a new position
  - Cycle repeats
- The deposition of mass from a particle as density into the mesh covers cells adjacent to the one where the particles is located



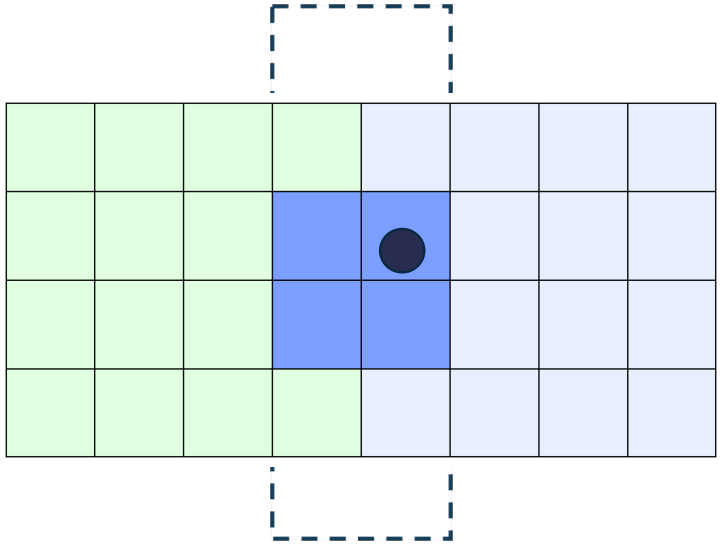
Use-case 1 – Developing New Code

- Massive particles for N-body methods used in astrophysics
  - The code has both mesh and particles
    - In parallel codes mesh is divided into blocks that are distributed among processors
  - Particles carry mass, mesh carries density
  - Particles deposit density onto the mesh
  - Mesh computes gravitational potential, forces and acceleration
  - Forces and acceleration are conveyed back to particles who move to a new position
  - Cycle repeats
- The deposition of mass from a particle as density into the mesh covers cells adjacent to the one where the particles is located
- Need for communication occurs when a particle is occupying a cell on the boundary of a block



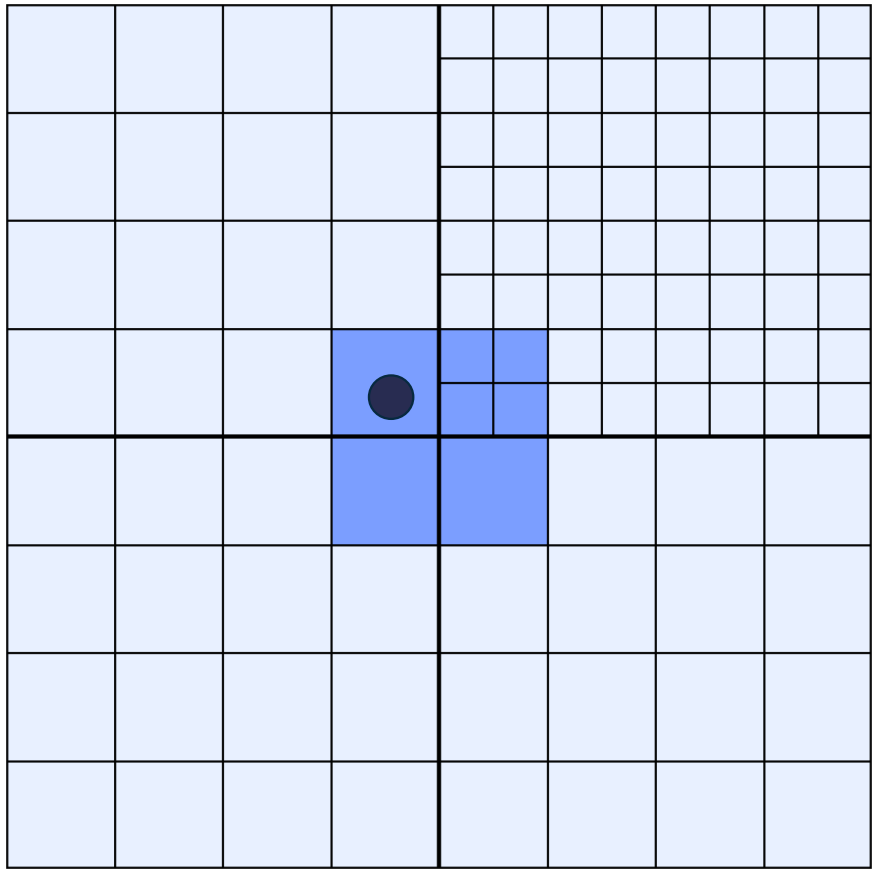
Communication Algorithm

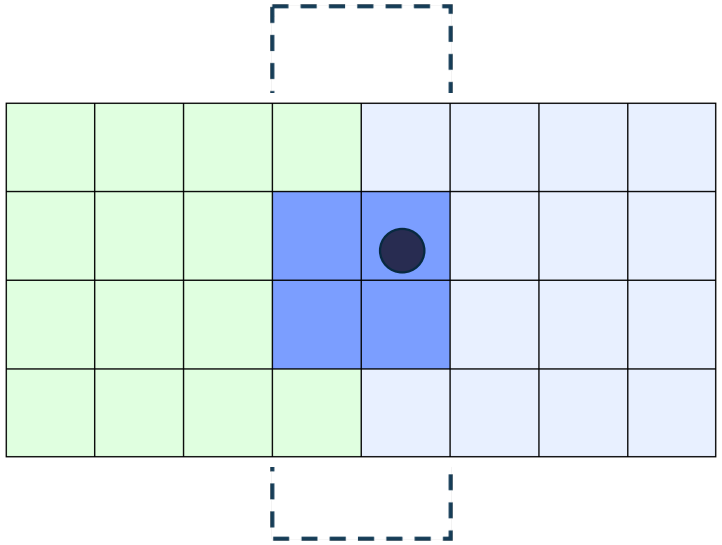
- Commonly used method is to use halo cells during deposition and then do a reverse halo filling
- Works fine with Uniformly discretized mesh



Communication Algorithm

- Commonly used method is to use halo cells during deposition and then do a reverse halo filling
- Works fine with Uniformly discretized mesh

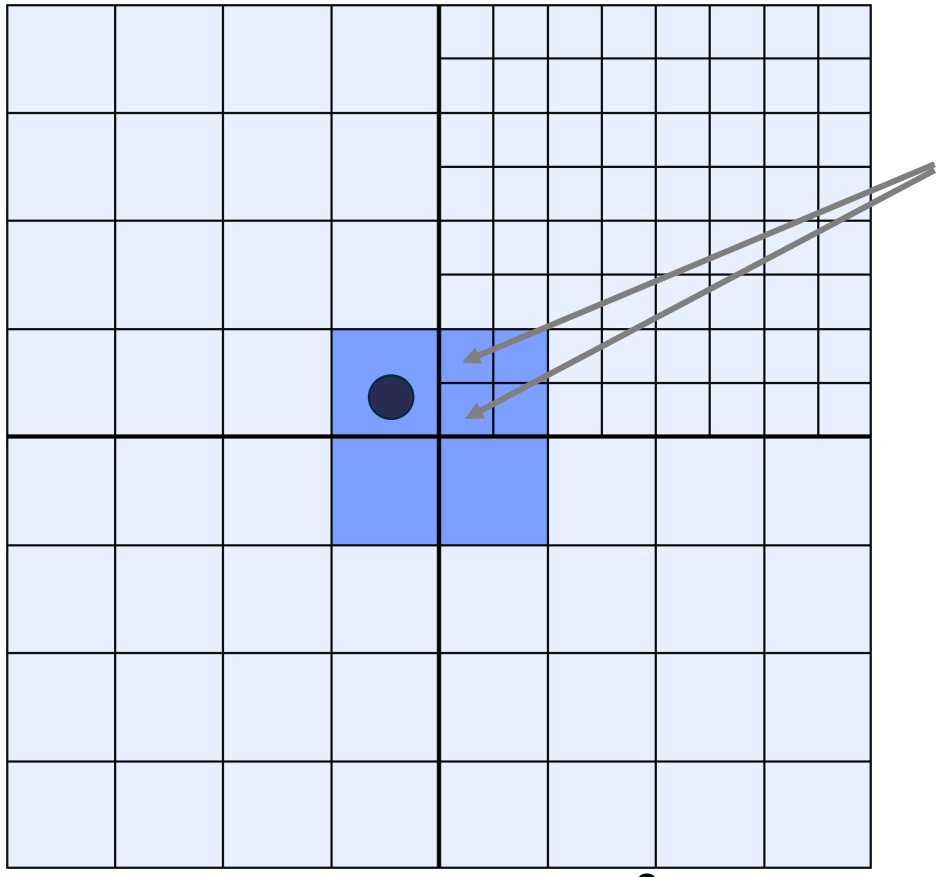


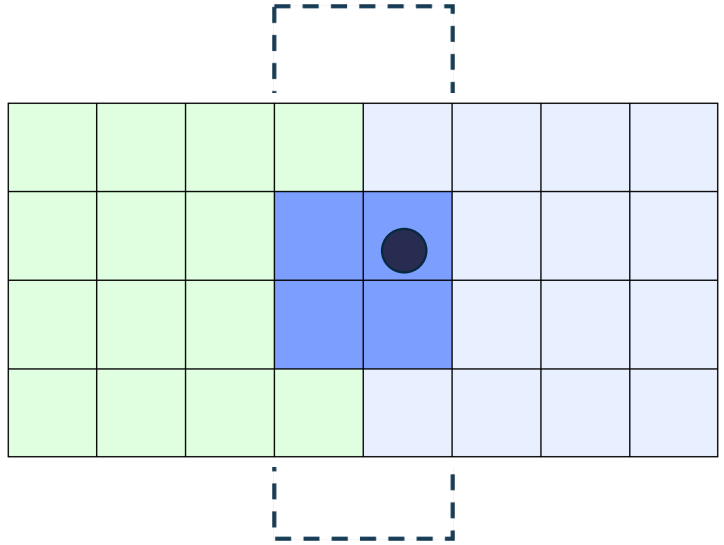


- More complicated when there is Adaptive Mesh Refinement (AMR)
- Halo filling is expensive because it is not necessarily nearest neighbor communication
- To avoid spurious forces on the refined side deposition needs to occur on two cells along each dimension

Communication Algorithm

- Commonly used method is to use halo cells during deposition and then do a reverse halo filling
- Works fine with Uniformly discretized mesh

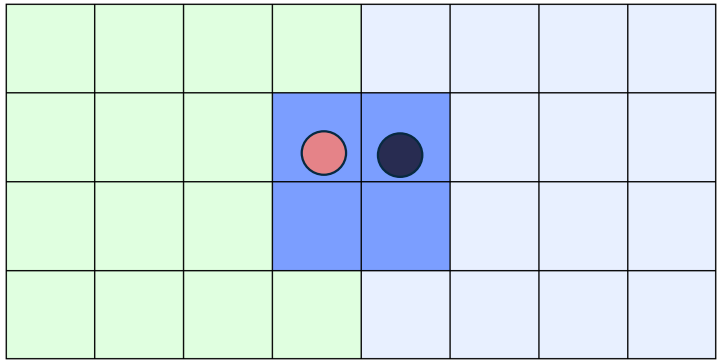




- More complicated when there is Adaptive Mesh Refinement (AMR)
- Halo filling is expensive because it is not necessarily nearest neighbor communication
- To avoid spurious forces on the refined side deposition needs to occur on two cells along each dimension

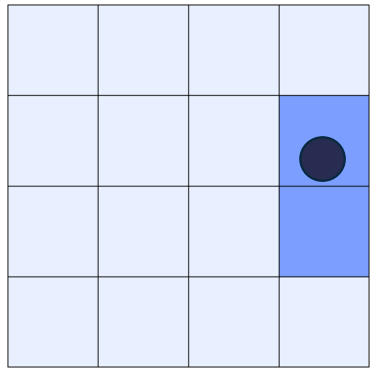
New Communication Algorithm

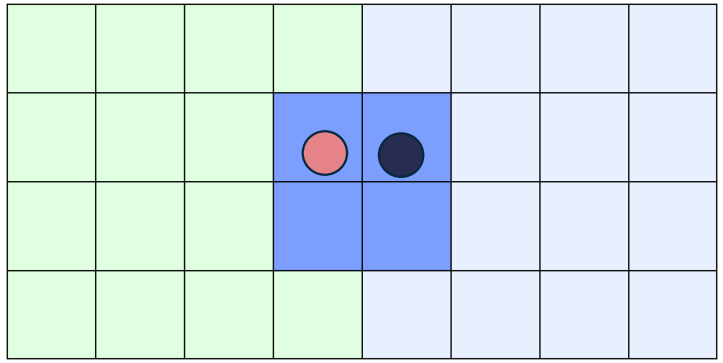
- Make virtual copies of the particle
- Send virtual copies instead of filled halo cells
- Deposit locally



New Communication Algorithm

- Make virtual copies of the particle
- Send virtual copies instead of filled halo cells
- Deposit locally

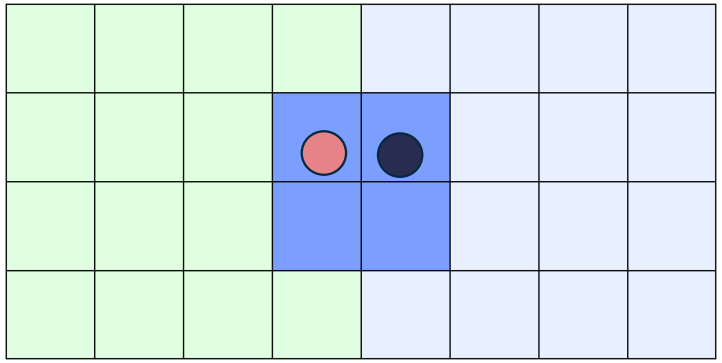


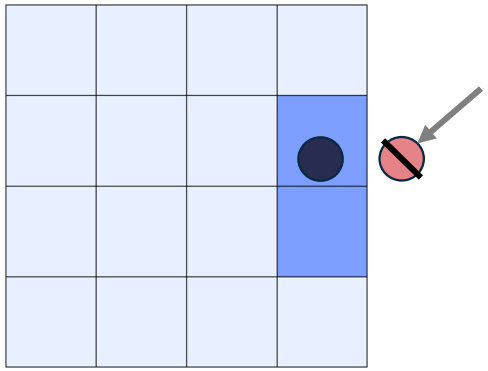


- Additional information is needed at physical boundaries

New Communication Algorithm

- Make virtual copies of the particle
- Send virtual copies instead of filled halo cells
- Deposit locally



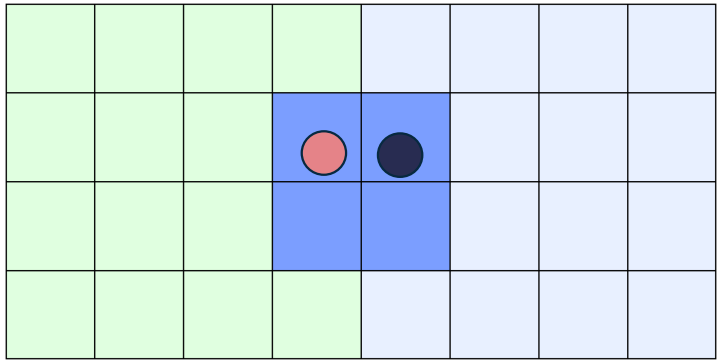


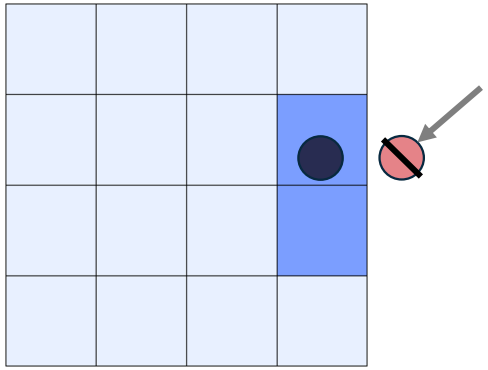
Outflow physical
boundary no contribution to potential

- Additional information is needed physical boundaries
- Some boundary conditions may obviate the need for virtual particles

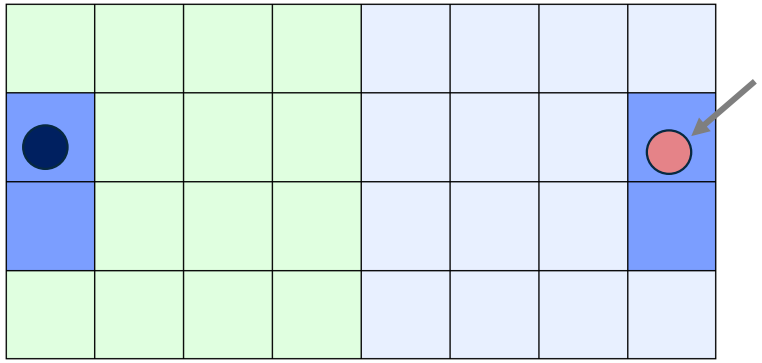
New Communication Algorithm

- Make virtual copies of the particle
- Send virtual copies instead of filled halo cells
- Deposit locally





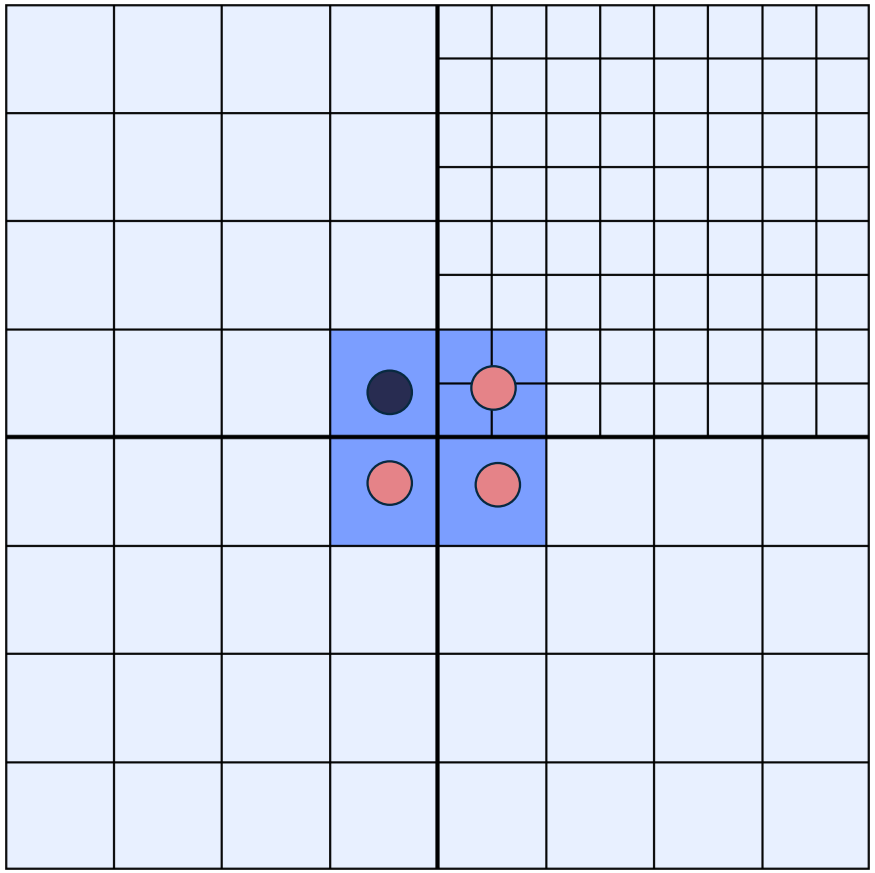
Outflow physical boundary no contribution to potential



Periodic boundary particles end up elsewhere

- Additional information is needed physical boundaries
- Some boundary conditions may obviate the need for virtual particles
- Periodic boundary conditions change the destination of the virtual particles

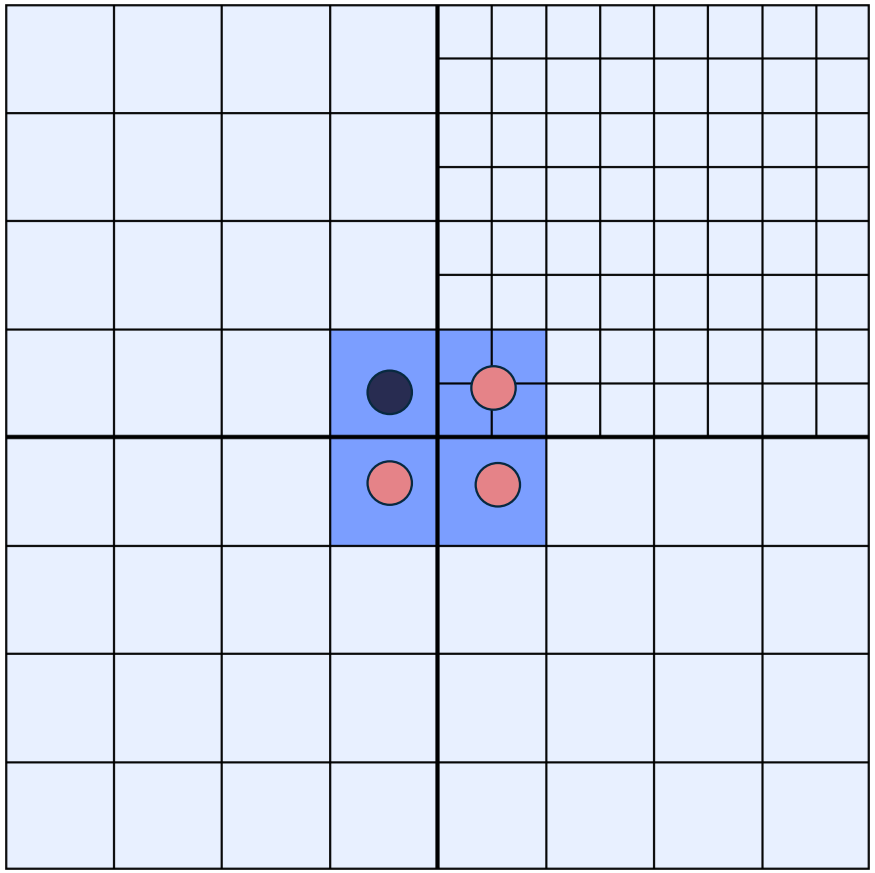
New Communication Algorithm



- Additional information is needed for AMR
  - Is the particle close to a fine-coarse boundary
  - If so where does it deposit on two cells instead of one

- I want LLMs to generate code for me
- And I want to do test-driven development
  - Everyone knows you can't assume correctly generated code
- So I need to break down the algorithm into smaller, testable steps
- And then I want to see if I can get LLM to also generate the test for me

New Communication Algorithm



- Additional information is needed for AMR
  - Is the particle close to a fine-coarse boundary
  - If so where does it deposit on two cells instead of one

- I want LLMs to generate code for me
- And I want to do test-driven development
  - Everyone knows you can't assume correctly generated code
- So I need to break down the algorithm into smaller, testable steps
- And then I want to see if I can get LLM to also generate the test for me.

This development is for Flash-X, a well-established code, which means I can make assumptions about available utilities. But I also need to figure out how to fake those utilities for the purpose of testing the generated code

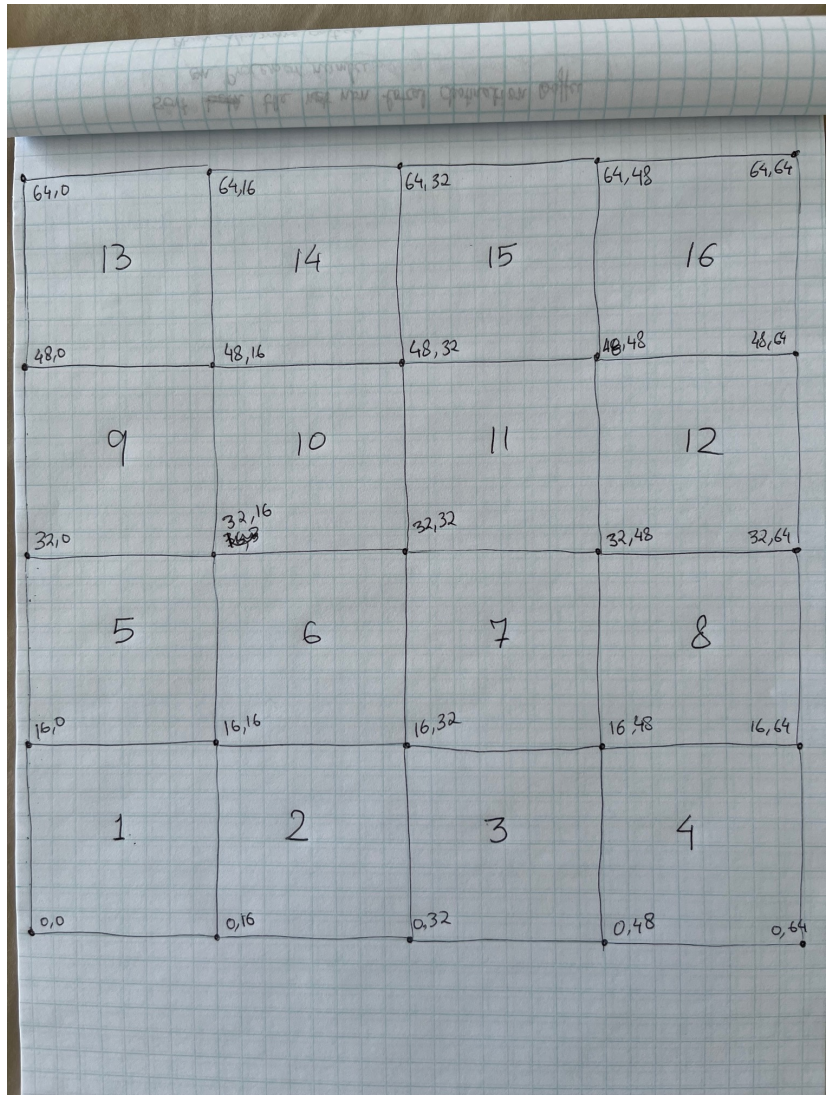
Development and Testing Methodolgy

- ❑ Set up the testing environment
- ❑ Define constants that exist in the code
- ❑ Create a uniformly discretized mesh with easy to inspect numbers
- ❑ Create simplified versions of the utilities available in the code

- ❑ For the purpose of testing overall I specified a uniform grid with 4x4 blocks for 2D and 4^3 blocks for 3D, blocks have 8 cells along each dimension, and physical size being 0.0-64.0 along each dimension
- ❑ I have a copy of the mesh on paper, and I manually inspect the output of tests and verify against my paper setup.

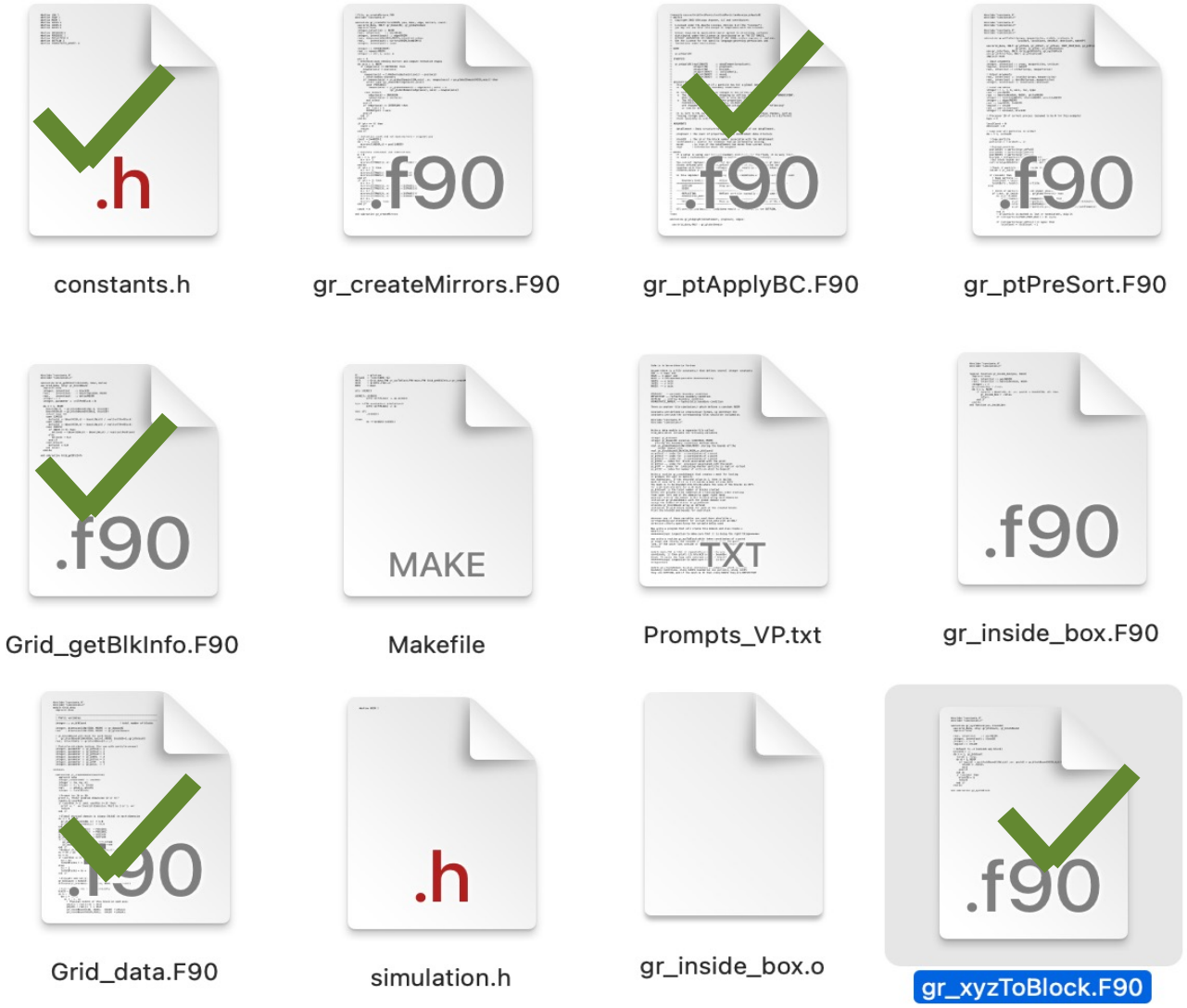
Development and Testing Methodolgy

- ❑ Set up the testing environment
- ❑ Define constants that exist in the code
- ❑ Create a uniformly discretized mesh with easy to inspect numbers
- ❑ Create simplified versions of the utilities available in the code



- ❑ For the purpose of testing overall I specified a uniform grid with 4x4 blocks for 2D and 4^3 blocks for 3D, blocks have 8 cells along each dimension, and physical size being 0.0-64.0 along each dimension
- ❑ I have a copy of the mesh on paper, and I manually inspect the output of tests and verify against my paper setup.

View of the working folder



- All the routines with a check mark exist in the main code
  - Here simpler version for testing are generated by AI
  - The test is also generated by AI

Prompt Example – Mesh generation

Grid_data Module

integer gr_blkCount
integer gr_domainBC sized as (LOW:HIGH, MDIM) storing the boundary conditions as defined before
real gr_globalDomain(LOW:HIGH,MDIM) storing the bounds of the global domain size
real gr_blockBound(LOW:HIGH,MDIM,gr_blkCount)

Prompt for creating the mesh – constants and variables are already known to the LLM

Write a routine gr_createDomain that creates a mesh. It prompts the user to specify the dimensions. If the returned value is 2, then it builds mesh of size 64^2, if it is 3 it builds a mesh of size 64^3. The mesh is to be divided into blocks where the size of the blocks is 16^2 for a 2D mesh and 16^3 for a 3D mesh gr_blkCount is the total number of blocks created blocks are assumed to be numbered in a lexicographic order starting from lower left end of the domain to upper right hand. physical size of the domain is 0.0 to 64.0 along each dimension

initialize gr_globalDomain with the global domain size
assign the number of blocks to gr_blkCount, allocate gr_blockBound array as defined
initialize it with block bounds for each of the created blocks

Adding Utilities and Testing Them

Prompts for faking the utilities

- Now write a program that will create this domain and also create a makefille
- Next write a routine gr_xyzToBlock which takes coordinates of a point as input and returns the blockID of the block on which the point lies. If the point lies outside of the global domain it returns -1 for blockID
- modify main.f90 so that it repeatedly prompts the user for a coordinate, it then prints its blockID and the boundbox for the block. It exits the loop with returned value of blockID is -1
- modify gr_createDomain to also initialize gr_domainBC, which contains boundary conditions. Along IAXIS boundaries are periodic, along JAXIS they are OUTFLOW, and if the mesh is 3D then along KAXIS they are REFLECTIVE
- Now write a stand-alone subroutine that takes blockID as input and returns the bound box in a real array bbox(LOW:HIGH,MDIM) and the delta -- the physical size of the cell in real array(MDIM). delta(KAXIS) is set to 0 for a 2D mesh

Adding Utilities and Testing Them

Prompts f

At each of these prompts main.F90 was modified to exercise the new functionality and Makefile was modified to add new files. All done with LLM.

- Now writ … makefille
- Next writ … pint as input and returns the blockID … of the global domain it returns -1 for block…
- modify m… nate, it then prints its blockID and the boundbox for the block. It exits the loop with returned value of blockID is -1
- modify gr_createDomain to also initialize gr_domainBC, which contains boundary conditions. Along IAXIS boundaries are periodic, along JAXIS they are OUTFLOW, and if the mesh is 3D then along KAXIS they are REFLECTIVE
- Now write a stand-alone subroutine that takes blockID as input and returns the bound box in a real array bbox(LOW:HIGH,MDIM) and the delta -- the physical size of the cell in real array(MDIM). delta(KAXIS) is set to 0 for a 2D mesh

Adding Utilities and Testing Them

Prompts f

- Now writ … makefille
- Next writ … the blockID … eturns -1 for block
- modify m … ckID and the boun…
- modify gr_createDomain to also initialize gr… s. Along IAXIS boundaries are periodic, along JAXIS… n along KAXIS they are REFLECTIVE
- Now write a stand-alone subroutine that tak… in a real array bbox(LOW:HIGH,MDIM) and the delta… IM). delta(KAXIS) is set to 0 for a 2D mesh

At each of these prompts main.F90 was modified to exercise the new functionality and Makefile was modified to add new file… with LLM.

The generated code almost always had compilation errors, but they were easy to fix. And usually arose from ambiguity in the prompt, though not always.

Adding Utilities and Testing Them

Prompts f

- Now writ … makefille
- Next writ … the blockID … eturns -1 for block
- modify m … ckID and the boun…
- modify gr_… …s. Along IAXIS boun… …n along KAXIS they…
- Now write a … in a real array bbox(… …M). delta(KAXI…

At each of these prompts main.F90 was modified to exercise the new functionality and Makefile was modified to add new file with LLM.

The generated code almost always had compilation errors, but they were easy to fix. And …rose from ambiguity …ompt, though not

The whole exercise took less than two hours, including making my paper version for verification.

The Algorithm

Took several iterations of changes in design and interaction with LLM to figure out

Portion of the algorithm for creating mirrors

- Check if the particle is close to a block boundary
  - If true check if it is close to a physical boundary
  - If true apply boundary conditions
    - If particle left the domain there is no need for virtual particles
    - If particle got reflected back no need for virtual particles along corresponding axis
- If virtual particles are needed create mirror positions
  - If the particle is close to the boundary along 1 axis only, 1 mirror is needed
  - If it is close to two boundaries 3 mirrors are needed
  - If it is close to three boundaries 7 mirrors are needed

- Part of the prompt that specifies what to do

loop over all axes and create a new value for the coordinate along each axis. If it is INTERIOR, newpos(axis)=pos(axis)

If not INTERIOR newpos(axis) = 2.0*bbox(edgetype(axis), axis) - pos(axis)

If the newpos(axis) is outside the physical domain, and if the boundary condition is periodic then newpos(axis) = newpos(axis) = gr_globalDomain(3 - edge(axis), axis) - (gr_globalDomain(edge(axis), axis) - newpos(axis))

if it is any other boundary condition change edgetype(axis) to INTERIOR and change newpos(axis) = pos(axis)

if after handling boundary conditions edgetype is still not INTERIOR then increment ptr by 1, and set ISTRUE(ptr) = axis

```
for all i 1 to ptr

  mirror(ISTRUE(i),i)=newpos(ISTRUE(i))

if ptr > 1 then
   mirror(ISTRUE(1),i+1)=newpos(ISTRUE(1))
   mirror(ISTRUE(2),i+1)=newpos(ISTRUE(2)

if ptr > 2 then
   mirror(ISTRUE(1),i+2)=newpos(ISTRUE(1))
   mirror(ISTRUE(3),i+2)=newpos(ISTRUE(3))
   mirror(ISTRUE(2),i+3)=newpos(ISTRUE(2))
   mirror(ISTRUE(3),i+3)=newpos(ISTRUE(3))
   mirror(:, i+4) = newpos(:)
```

Observations about Code generation

Promising Results – a lot of manual work still needed

- Prompts need to be precise, specific, and unambiguous
  - It is like coding in a natural language
  - Natural languages are imprecise by definition
    - Makes communicating complicated requirements to LLM difficult
  - Decomposition of requirements into smaller chunks becomes unavoidable
    - Implies more thought to be given to code design and componentization
    - In the long run good for code maintainability
- Debugging generated code directly is not necessary
  - It is better to reason about the deficiency in logic through testing
    - One can look at the reasoning shown by the LLM as it analyzes
    - One can also ask for extensive inline documentation
    - Often inspecting the documentation can lead to understanding the deficiency in logic
  - If one can, debugging the prompt is better

More Observations about Code generation

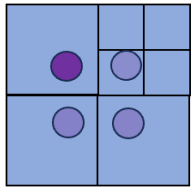
- I found it easier to abandon working code when I thought of a better approach
- Experimented with four different ways of implementing before settilng down on the current one
- With every iteration the code got cleaner and smaller

More Observations about Code generation

- I found it easier to abandon working code when I thought of a better approach
- Experimented with four different ways of implementing before

The Algorithm – Make Virtual Copies

- For each particle check if the Particle is within one cell of block boundary for Left/Right Edge along each dimension
  - Where true create a virtual copy of the particle
  - Find the neighboring block along each edge where a virtual particle is created
  - Find the refinement level of the neighbor
- For each virtual particle update the attributes
  - Assign as overlapping with the neighboring block
  - If neighbor is at same refinement level value to be deposited into one cell
  - If neighbor is at finer level, value to be deposited in 2^d cells
  - If neighbor is at coarser level, virtual particle deposits into one cell, real particle deposits into 2^d cells.



```
subroutine checkEdges(bndbox,
del, pos, xedge, yedge, zedge)

 -- given the coordinates of the
particle in pos, bounding box of
the block in bndbox, and size of
the cell in del, it returns values
in x/y/z/edge that indicate whether
a virtual particle is needed
```

```
subroutine processParticles (numParticles,
particleProps, src, dest, local)

 -- src array contains the unprocessed particles,
dest has particles that need to move and local has
particles that stay after processing. This
subroutine calls checkedges, and another routine
that gets info about negh, creates virtual
particles as needed and puts them into the right
array
```

More Observations about Code generation

- I found it easier to abandon working code when I thought of a better approach
- Experimented with

The Algorithm – Make Virtu

- For each particle check if the Particle is within one cell of block boundary for Left/Right Edge along each dimension
  - Where true create a virtual copy of the particle
  - Find the neighboring block along each edge where a virtual particle is created
  - Find the refinement level of the neighbor
- For each virtual particle update the attributes
  - Assign as overlapping with the neighboring block
  - If neighbor is at same refinement level value to be deposited into one cell
  - If neighbor is at finer level, value to be deposited in 2^d cells
  - If neighbor is at coarser level, virtual particle deposits into one cell, real particle deposits into 2^d cells.

Prompt – checkEdges

Write a subroutine in Fortran that returns three integer values returned in variables xedge, vedge and zedge

assume constants LOW=1, HIGH=2, MDIM=3

IAXIS=1, JAXIS = 2, KAXIS =3 are defining physical dimensions

Inputs to the function are:

real array bndbox(LOW:HIGH,MDIM) -- bndBox(LOW,:) is the lower left-hand corner and bndBox(HIGH,:) is the upper right-hand corner coordinates of a block

del(MDIM) are size of individual cells in the block along the three dimensions

pos(MDIM) is the coordinate of a point

xedge contains values for IAXIS, vedge for JAXIS and zedge for KAXIS

Each of the output arguments can take one of five values

10 if the point is in the box and one cell or more away from the both left and right edges

11 if the point is in the box and less than a cell away from left edge

12 if the point is in the box and less than a cell away from right edge

13 if the point is outside the box and less than a cell away from left edge

14 if the point is outside the box and less than a cell away from right edge

15 if the point is outside the box and more than a cell away from both edges

```
that gets info about negh, creates virtual
particles as needed and puts them into the right
array
```

More Observations about Code generation

- I found it easier to abandon working code when I thought of a better approach
- Experimented with four different ways of implementing before setting down on the current one
- With every iteration the code got cleaner and smaller

With plenty of inline comments in the code and preserved prompts I have almost complete specification of the code
Excellent for maintenance

Use-case 2 – Code Translation

- It all started with MCFM, a Monte Carlo code that gives predictions for a wide range of processes at hadron colliders
- It needs to be integrated into a new framework Pepper which is a GPU based code developed to handle the next generation of computational work for the colliders
- Pepper is written in C++ and aims to obtain performance portability with Kokkos
- MCFM is Fortran – scientists want it converted to C++
- MCFM has nearly 500 source files spread across multiple directories, with around 50-200 lines per file. Most files fit within the LLM context window
- **https://neucol.github.io/pages/software**

Initial Exploration

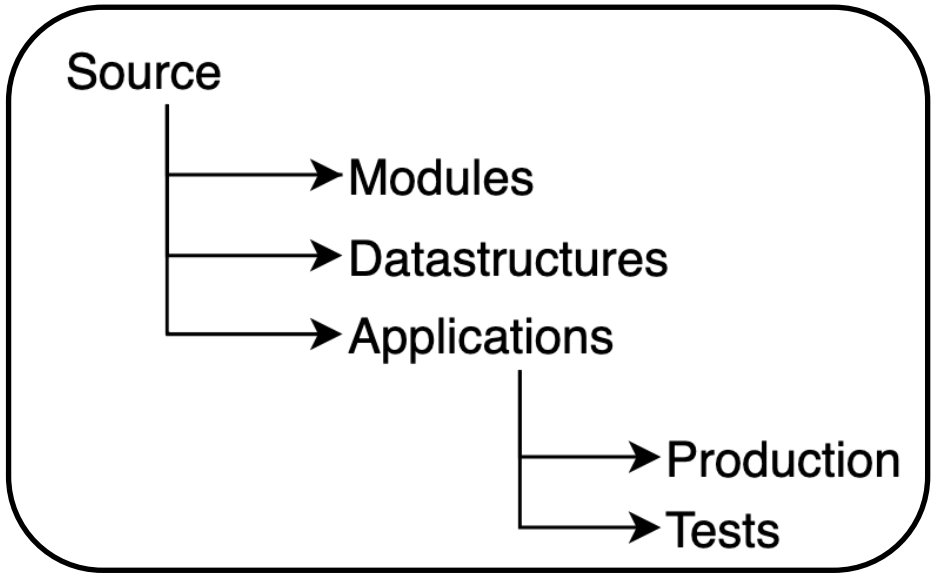
- All of this was happening when ChatGPT had just started making waves, and we decided it would be worth exploring code translation as a use case
- It starting with cutting and pasting code snippets of mostly arithmetic into the chat window and asking for translation
  - Turned out the translation was syntactically correct
- But translation is not about only syntax, and code is not just arithmetic
  - We clearly needed more
- We also thought scripting can be a part of the solution, along with a human in the loop feature

Initial Exploration

- All of this was happening when ChatGPT had just started making waves, and we decided it would be worth exploring code translation as a use case
- It starting with cutting and pasting code snippets of mostly arithmetic into the chat window and asking for translation
  - Turned out the translation was syntactically correct
- But translation is not about only syntax, and code is not just arithmetic
  - We clearly needed more
- We also thought scripting can be a part of the solution, along with a human in the loop feature

What resulted from the desire to do code translation without the need to understand the code, and to do it rapidly is **CodeScribe**

Overview of The Code Translation Task

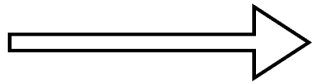


- The code is organized into a directory structure shown above
- Tests (a subset of Production Applications) that provide coverage for all sections of the code can be used as the test-suite to ensure correctness for any refactoring of the code
- Tests are the reason why CodeScribe became a viable solution

Example of Fortran module conversion to C++ headers and source.

example_mod.f90

```
module example_mod
  use types
  use nf_mod
  implicit none

public
    integer :: kdot
    integer,parameter:: nloop=2, fn=-5
    real(dp),parameter:: zip=0._dp
    real(dp),parameter:: one=1._dp
    complex(dp),parameter:: im=(zip,one)
    logical :: myflag
    real(dp) :: tau(-nf:nf), ln(nf)
    real(dp) :: le
    real(dp) :: bxm, bxn
    save

end module example_mod
```



example_mod.hpp

```
#ifndef EXAMPLE_MOD
#define EXAMPLE_MOD

#include<nf_mod.hpp>

namespace example_mod {
  using namespace nf_mod;

  extern int kdot;
  const int nloop = 2;
  const int fn = -5;
  const double zip = 0.0;
  const double one = 1.0;
  extern bool myflag;
  const std::complex<double> im(zip, one);
  extern double tau[2*nf+1], l[nf];
  extern double le;
  extern double bxm, bxn;
}
#endif
```

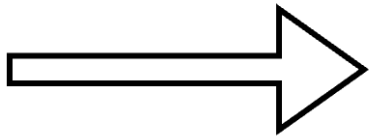
example_mod.cpp

```
#include<example_mod.hpp>
#include<nf_mod.hpp>

namespace example_mod {
  using namespace nf_mod;

  int kdot;
  bool myflag;
  double tau[2*nf+1], l[nf];
  double le;
  double bxm, bxn;
}
```

Example of FORTRAN subroutine conversion to C++.

couplz.f

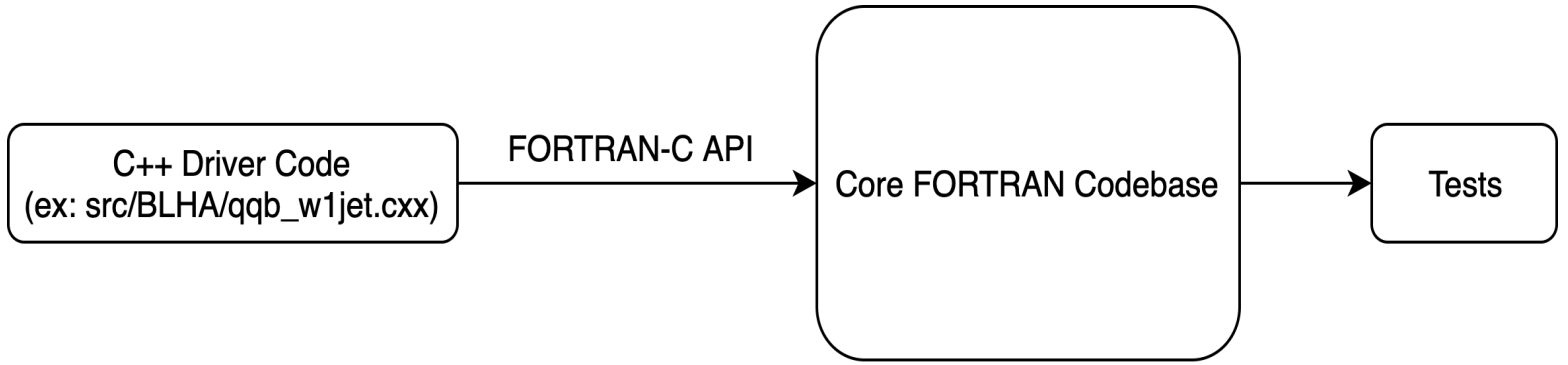
```
subroutine couplz(xw)
  use constants_mod
  use nf_mod
  use zcouple_mod
  use ewcharge_mod
  real(dp):: xw
  !!  ...
  !!  ...
  return
end subroutine couplz
```



couplz.cpp

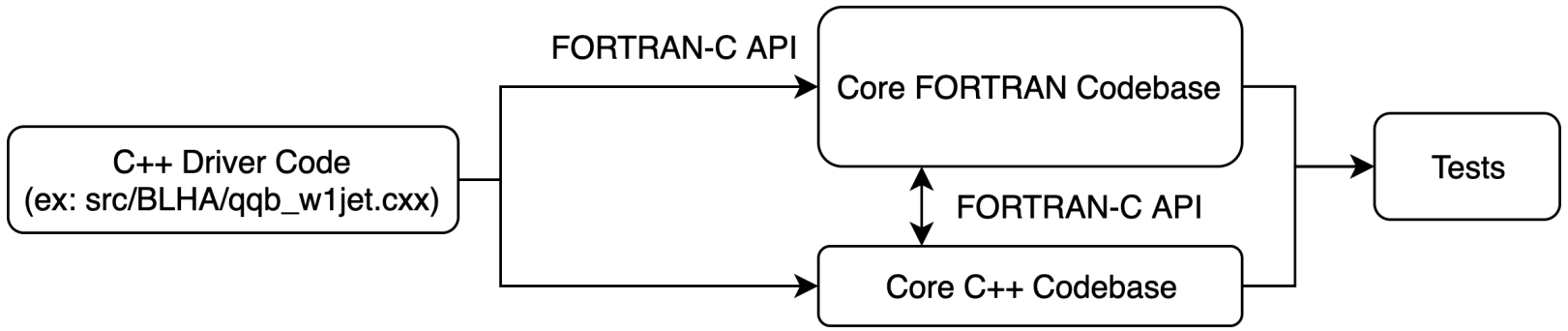
```
#include<constants_mod.hpp>
#include<nf_mod.hpp>
#include<zcouple_mod.hpp>
#include<ewcharge_mod.hpp>

void couplz(double xw) {
  using namespace constants_mod;
  using namespace nf_mod;
  using namespace zcouple_mod;
  using namespace ewcharge_mod;
  // ...
  // ...
  return;
}
```

Our Approach – A Step-by-step Process



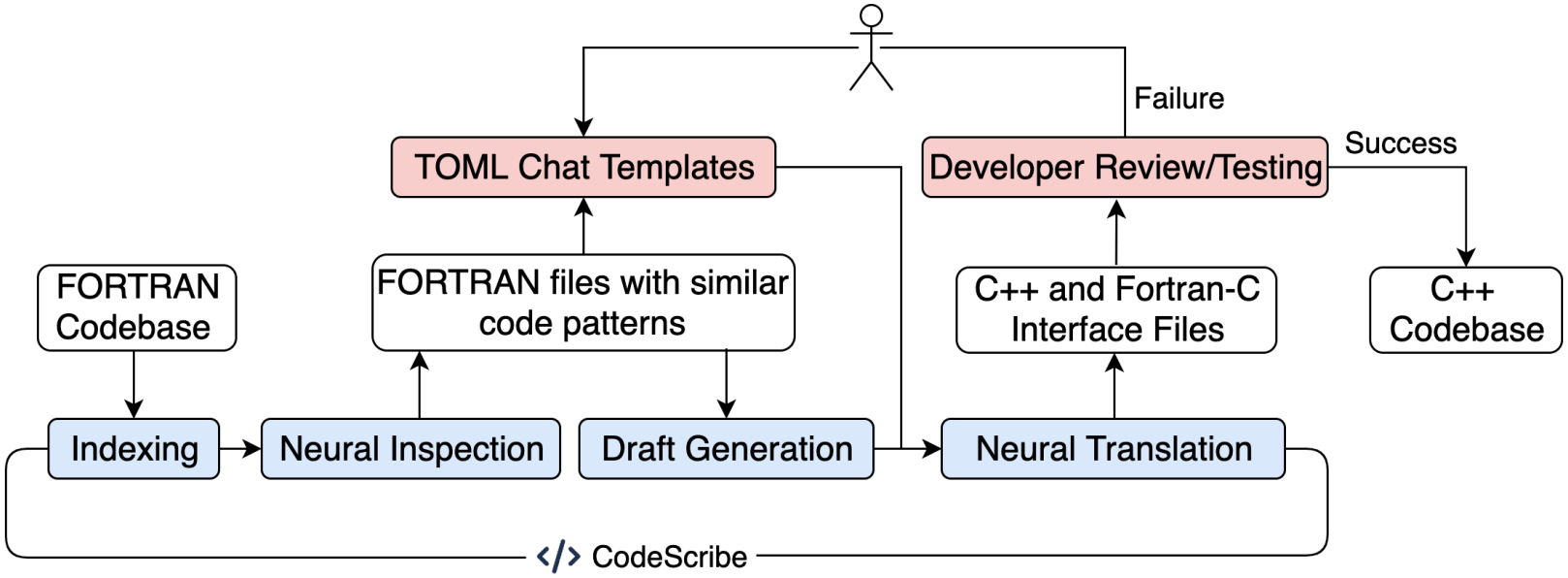
- Determine small groups of modules and data structures that are interdependent on one another but independent of the source code otherwise
- Develop prompts to teach LLM the rules for conversion from Fortran to C++
- Write corresponding Fortran-C-API to integrate the generated code with the application which still has a lot of Fortran code
- Run the relevant portion of the test-suite
- Debug manually and/or refine prompts

Our Approach – A Step-by-step Process



- Determine small groups of modules and data structures that are interdependent on one another but independent of the source code otherwise
- Develop prompts to teach LLM the rules for conversion from Fortran to C++
- Write corresponding Fortran-C-API to integrate the generated code with the application which still has a lot of Fortran code
- Run the relevant portion of the test-suite
- Debug manually and/or refine prompts

Tool Developed For Conversion -- CodeScribe



A customized Python engine to experiments with different approaches for code conversion and test performance of different models.

akashdhruv/CodeScribe

Index command

- Maps the source tree by analyzing file hierarchies, dependencies, and constructs, creating YAML files for efficient navigation
- YAML files store metadata and are compiled into an inverse dictionary to support accurate code queries
- The Index command prevents LLM hallucinations by providing a structural map to guide accurate code translation that is used as a RAG database

```yaml
# Structure of scribe.yaml

directory: src
files:
  file1.f90:
    modules:
      - module1
    subroutines:
      - subroutine1
      - subroutine2
    functions:
      - function1

  file2.f90:
    modules: []
    subroutines:
      - subroutineA
    functions:
      - functionB
```

Inspect command and TOML Chat completion Templates

- Enables interactive queries about source code.
- This command is used to build the chat completion templates:
  - User starts with describing the rules of conversion and then provides a dummy example of the source code
  - Assistant provides syntactically correct code
  - User appends the actual source code which is then passed on the LLM to complete the conversation as the assistant

```
# Structure of seed_prompt.toml

[[chat]]
role = "user"
content = "Rules and syntax-related instructions for code conversion>"

[[chat]]
role = "assistant"
content = "I am ready. Please give me a test problem."

[[chat]]
role = "user"
content = "<Template of contents in a source file>"

[[chat]]
role = "assistant"
content = "<Desired contents of the converted file. Syntactically correct code>"

[[chat]]
role = "user"
content = "<Append code from a source file>"
```

This is useful for identifying patterns in files and building chat completion prompts for different patterns

````
[[chat]]
role = "user"
content = """
You are a code conversion tool for a scientific computing application.
The application is organized as different source files in a directory structure.
I will give you a Fortran file which you will convert to a C++ source code file.

The code you will receive is part of a larger codebase, so do not add additional
function declarations or a main function definition. Just perform the conversion
process line-by-line.

Here are some rules I want you to follow:

1. The input code is a Fortran subroutine or function which needs to be converted
   to a C++ function. Replace the Fortran structure:

   ```
   use <module-name>
   subroutine <func-name>(xw)
    real(dp):: xw
    ...
   end subroutine
   ```

   with the C++ equivalent:

   ```
   #include <module-name.hpp>
   using namespace <module-name>;

   void <func-name>(double& xw) {
     ...
     return;
   }

   extern "C" {
     void <func-name>_wrapper(double* xw) {
       <func-name>(*xw);
     }
   }
   ```

2. Replace any use `<module-name>` statements with `#include <module-name.hpp>`
   and using `namespace <module-name>;`. Place the `#include` statements at the top
   of the file. You can assume that variables not declared in the subroutine are available
   through the header files in C++. Ignore `use types` and other irrelevant modules.

3. Convert Fortran types as follows:

   `real(dp)` to `double`
   `complex(dp)` to `std::complex<double>` Include the `<complex>` header in case of complex types.

   Convert Fortran arrays to C++ using the `FArray` template classes which support Fortran-like
   indexing. For example, replace:

   ```
   real(dp), dimension(nx, ny) :: a
   ```

   with:

   ```
   FArray2D<double> a(nx, ny);
   ```

   Include the <FArray.hpp> header and use the appropriate class (FArray1D, FArray2D, FArray3D) depending on
````

DRAFT command

- ❑ Creates a file specific prompt for LLM consumption.
- ❑ Created file has ”scribe” suffix.
- ❑ Basic types and multidimensional arrays are converted from Fortran to C++.
- ❑ YAML Index files are used to provide context about the use of external functions.
- ❑ Provides additional datapoint to LLM by doing trivial code conversions

```
# Excerpt from <results>/rag-sensitivity/with-function-context/target.scribe
scribe-prompt: Write corressponding extern "C" with _wrapper added to the name.
               Refer to the template for treating Farray and scalars

scribe-prompt: When variables are used as function. They should be treated as
               external or statement functions. External functions are available
               in header files

scribe-prompt: Statement functions should be converted to equivalent lambda functions
               in C++. Include [&] in capture clause to use variables by reference

scribe-prompt: Lsm1 is an external function
scribe-prompt: L0 is an external function
scribe-prompt: lnrat is an external function
scribe-prompt: L1 is an external function

<draft-cpp-code>
..
</draft-cpp-code>
```

```
!
!  Copyright (C) 2019-2022, respective authors of MCFM.
!  SPDX-License-Identifier: GPL-3.0-or-later

      function A51(j1,j2,j3,j4,j5,za,zb)

c  Amplitudes taken from Appendix IV of
c  Z.~Bern, L.~J.~Dixon and D.~A.~Kosower,
c  %``One loop amplitudes for e+ e- to four partons,''
c  Nucl.\ Phys.\ B {\bf 513}, 3 (1998)
c  [hep-ph/9708239].
c  Modified to remove momentum conservation relations

      use types
      use constants_mod
      use mxpart_mod
      use sprods_com_mod
      use scale_mod
      use epinv_mod
      use epinv2_mod
      implicit none
      complex(dp):: A51
      complex(dp):: za(mxpart,mxpart),zb(mxpart,mxpart)
      integer:: j1,j2,j3,j4,j5
      complex(dp):: Vcc,Fcc,Vsc,Fsc,l12,l23,L0,L1,Lsm1,A5lom
      complex(dp):: lnrat,zab2
      real(dp):: s123

      zab2(j1,j2,j3,j4)=za(j1,j2)*zb(j2,j4)+za(j1,j3)*zb(j3,j4)

c     -i * A5tree  Eq.(IV.1)
      s123=s(j1,j2)+s(j2,j3)+s(j3,j1)
      A5lom=-za(j3,j4)*zab2(j3,j1,j2,j5)/(za(j1,j2)*za(j2,j3)*s123)

c--leading N  Eq. (IV.2)
      l12=lnrat(musq,-s(j1,j2))
      l23=lnrat(musq,-s(j2,j3))
      Vcc=
     & -(epinv*epinv2+epinv*l12+half*l12**2)
     & -(epinv*epinv2+epinv*l23+half*l23**2)
     & -two*(epinv+l23)-four

c--Eq. (IV.3)
      Fcc=zab2(j3,j1,j2,j5)/(za(j1,j2)*za(j2,j3)*s123)
     & *(za(j3,j4)*Lsm1(-s(j1,j2),-s123,-s(j2,j3),-s123)
     & +two*za(j3,j1)*zab2(j4,j2,j3,j1)
     &   *L0(-s(j2,j3),-s123)/s123)

c--Eq. (IV.4)
      Vsc =half*(epinv+l23)+one
c--Eq. (IV.5)
      Fsc =-za(j3,j4)*za(j3,j1)*zb(j1,j5)
     & /(za(j1,j2)*za(j2,j3))*L0(-s(j2,j3),-s123)/s123
     & +half*za(j3,j1)**2*zb(j1,j5)*zab2(j4,j2,j3,j1)
     & /(za(j1,j2)*za(j2,j3))*L1(-s(j2,j3),-s123)/s123**2

      A51=(Vcc+Vsc)*A5Lom+Fcc+Fsc

      return
      end
```

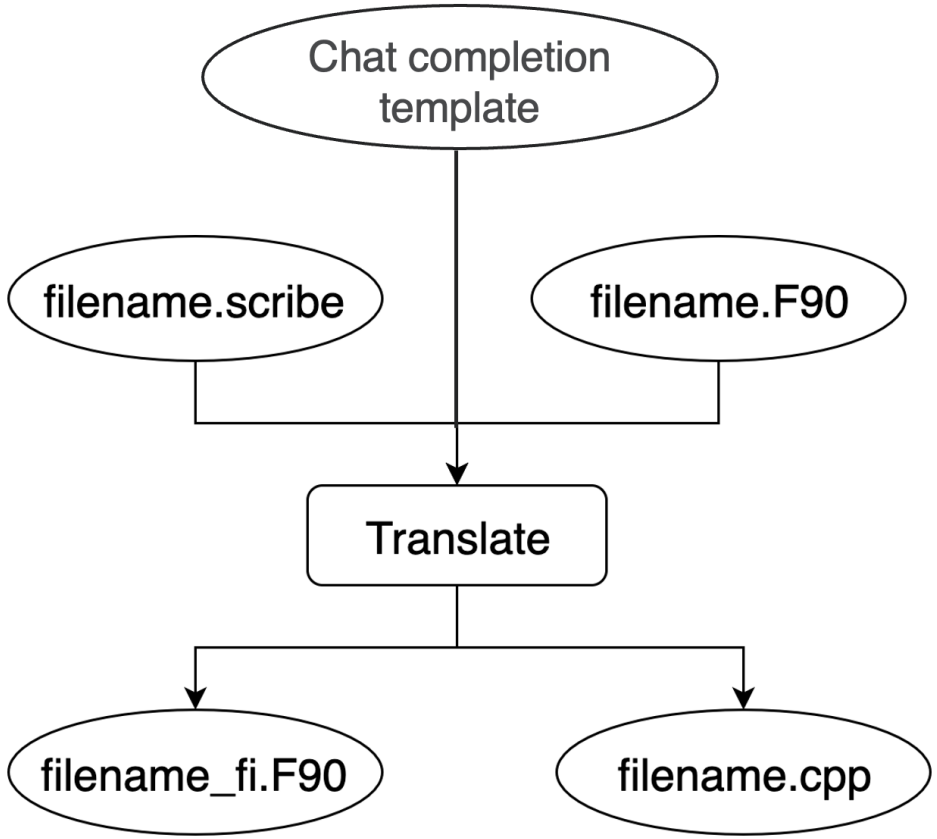
```
scribe-prompt: Write corressponding extern "C" with _wrapper added to the name. Refer to the template for treating Farray and scalars
scribe-prompt: When variables are used as function. They should be treated as external or statement functions. External functions are available in header files
scribe-prompt: Statement functions should be converted to equivalent lambda functions in C++. Include [&] in capture clause to use variables by reference
scribe-prompt: Lsm1 is an external function
scribe-prompt: L0 is an external function
scribe-prompt: lnrat is an external function
scribe-prompt: L1 is an external function

#include <cmath>
#include <complex>
#include <constants_mod.hpp>
#include <epinv2_mod.hpp>
#include <epinv_mod.hpp>
#include <mxpart_mod.hpp>
#include <scale_mod.hpp>
#include <sprods_com_mod.hpp>
#include <types.hpp>

function A51(j1,j2,j3,j4,j5,za,zb)

using namespace types;
using namespace constants_mod;
using namespace mxpart_mod;
using namespace sprods_com_mod;
using namespace scale_mod;
using namespace epinv_mod;
using namespace epinv2_mod;

complex<double> A51
FArray<std::complex<double>> za(mxpart,mxpart),zb(mxpart,mxpart)
int j1,j2,j3,j4,j5
complex<double> Vcc,Fcc,Vsc,Fsc,l12,l23,L0,L1,Lsm1,A5lom
complex<double> lnrat,zab2
double s123

zab2(j1,j2,j3,j4)=za(j1,j2)*zb(j2,j4)+za(j1,j3)*zb(j3,j4)

s123=s(j1,j2)+s(j2,j3)+s(j3,j1)
A5lom=-za(j3,j4)*zab2(j3,j1,j2,j5)/(za(j1,j2)*za(j2,j3)*s123)

l12=lnrat(musq,-s(j1,j2))
l23=lnrat(musq,-s(j2,j3))
Vcc=
\ -(epinv*epinv2+epinv*l12+half*pow(l12,2))
\ -(epinv*epinv2+epinv*l23+half*pow(l23,2))
\ -two*(epinv+l23)-four

Fcc=zab2(j3,j1,j2,j5)/(za(j1,j2)*za(j2,j3)*s123)
\ *(za(j3,j4)*Lsm1(-s(j1,j2),-s123,-s(j2,j3),-s123)
\ +two*za(j3,j1)*zab2(j4,j2,j3,j1)
\   *L0(-s(j2,j3),-s123)/s123)

Vsc =half*(epinv+l23)+one
Fsc =-za(j3,j4)*za(j3,j1)*zb(j1,j5)
\ /(za(j1,j2)*za(j2,j3))*L0(-s(j2,j3),-s123)/s123
\ +half*za(j3,j1)**2*zb(j1,j5)*zab2(j4,j2,j3,j1)
\ /(za(j1,j2)*za(j2,j3))*L1(-s(j2,j3),-s123)/pow(s123,2)

A51=(Vcc+Vsc)*A5Lom+Fcc+Fsc

return
end
```

```
!
!  Copyright (C) 2019-2022, respective authors of MCFM.
!  SPDX-License-Identifier: GPL-3.0-or-later

      function A51(j1,j2,j3,j4,j5,za,zb)

c  Amplitudes taken from Appendix IV of
c  Z.~Bern, L.~J.~Dixon and D.~A.~Kosower,
c  %``One loop amplitudes for e+ e- to four partons,''
c  Nucl.\ Phys.\ B {\bf 513}, 3 (1998)
c  [hep-ph/9708239].
c  Modified to remove momentum conservation relations

      use types
      use constants_mod
      use mxpart_mod
      use sprods_com_mod
      use scale_mod
      use epinv_mod
      use epinv2_mod
      implicit none
      complex(dp):: A51
      complex(dp):: za(mxpart,mxpart),zb(mxpart,mxpart)
      integer:: j1,j2,j3,j4,j5
      complex(dp):: Vcc,Fcc,Vsc,Fsc,l12,l23,L0,L1,Lsm1,A5lom
      complex(dp):: lnrat,zab2
      real(dp):: s123

      zab2(j1,j2,j3,j4)=za(j1,j2)*zb(j2,j4)+za(j1,j3)*zb(j3,j4)

c    -i * A5tree  Eq.(IV.1)
      s123=s(j1,j2)+s(j2,j3)+s(j3,j1)
      A5lom=-za(j3,j4)*zab2(j3,j1,j2,j5)/(za(j1,j2)*za(j2,j3)*s123)

c--leading N  Eq. (IV.2)
      l12=lnrat(musq,-s(j1,j2))
      l23=lnrat(musq,-s(j2,j3))
      Vcc=
     & -(epinv*epinv2+epinv*l12+half*l12**2)
     & -(epinv*epinv2+epinv*l23+half*l23**2)
     & -two*(epinv+l23)-four

c--Eq. (IV.3)
      Fcc=zab2(j3,j1,j2,j5)/(za(j1,j2)*za(j2,j3)*s123)
     & *(za(j3,j4)*Lsm1(-s(j1,j2),-s123,-s(j2,j3),-s123)
     & +two*za(j3,j1)*zab2(j4,j2,j3,j1)
     &   *L0(-s(j2,j3),-s123)/s123)

c--Eq. (IV.4)
      Vsc =half*(epinv+l23)+one
c--Eq. (IV.5)
      Fsc =-za(j3,j4)*za(j3,j1)*zb(j1,j5)
     & /(za(j1,j2)*za(j2,j3))*L0(-s(j2,j3),-s123)/s123
     & +half*za(j3,j1)**2*zb(j1,j5)*zab2(j4,j2,j3,j1)
     & /(za(j1,j2)*za(j2,j3))*L1(-s(j2,j3),-s123)/s123**2

      A51=(Vcc+Vsc)*A5Lom+Fcc+Fsc

      return
      end
```

```
#include <complex>
#include <constants_mod.hpp>
#include <epinv2_mod.hpp>
#include <epinv_mod.hpp>
#include <mxpart_mod.hpp>
#include <scale_mod.hpp>
#include <sprods_com_mod.hpp>
#include <types.hpp>
#include <FArray.hpp>

using namespace types;
using namespace constants_mod;
using namespace mxpart_mod;
using namespace sprods_com_mod;
using namespace scale_mod;
using namespace epinv_mod;
using namespace epinv2_mod;

std::complex<double> A51(int j1, int j2, int j3, int j4, int j5, FArray2D<std::complex<double>>& za, FArray2D<std::complex<double>>& zb) {
    std::complex<double> Vcc, Fcc, Vsc, Fsc, l12, l23, A5lom;
    std::complex<double> zab2(mxpart, mxpart);
    double s123;

    zab2(j1,j2) = za(j1,j2)*zb(j2,j4) + za(j1,j3)*zb(j3,j4);

    s123 = s(j1,j2) + s(j2,j3) + s(j3,j1);
    A5lom = -za(j3,j4) * zab2(j3,j1) / (za(j1,j2) * za(j2,j3) * s123);

    l12 = lnrat(musq,-s(j1,j2));
    l23 = lnrat(musq,-s(j2,j3));

    Vcc = -(epinv * epinv2 + epinv * l12 + half * std::pow(l12, 2))
         -(epinv * epinv2 + epinv * l23 + half * std::pow(l23, 2))
         -two * (epinv + l23) - four;

    Fcc = zab2(j3,j1) / (za(j1,j2) * za(j2,j3) * s123)
         * (za(j3,j4) * Lsm1(-s(j1,j2),-s123,-s(j2,j3),-s123)
         + two * za(j3,j1) * zab2(j4,j2) * L0(-s(j2,j3),-s123) / s123);

    Vsc = half * (epinv + l23) + one;
    Fsc = -za(j3,j4) * za(j3,j1) * zb(j1,j5)
         / (za(j1,j2) * za(j2,j3)) * L0(-s(j2,j3),-s123) / s123
         + half * std::pow(za(j3,j1), 2) * zb(j1,j5) * zab2(j4,j2)
         / (za(j1,j2) * za(j2,j3)) * L1(-s(j2,j3),-s123) / std::pow(s123, 2);

    return (Vcc + Vsc) * A5lom + Fcc + Fsc;
}

extern "C" {
    std::complex<double> A51_wrapper(int j1, int j2, int j3, int j4, int j5, std::complex<double>* zaf, std::complex<double>* zbf) {
        FArray2D<std::complex<double>> za(zaf, mxpart, mxpart);
        FArray2D<std::complex<double>> zb(zbf, mxpart, mxpart);
        return A51(j1, j2, j3, j4, j5, za, zb);
    }
}
```

Translate command

- Translation uses a chat template seed prompt, appending source and draft code to produce converted code
- The LLM outputs the C++ source and Fortran-C interface enclosed within, saving them as filename.cpp and filename_fi.f90
- Interaction with the LLM uses APIs, local models, or manual JSON-based copy-pasting, as outlined in the Inspect command

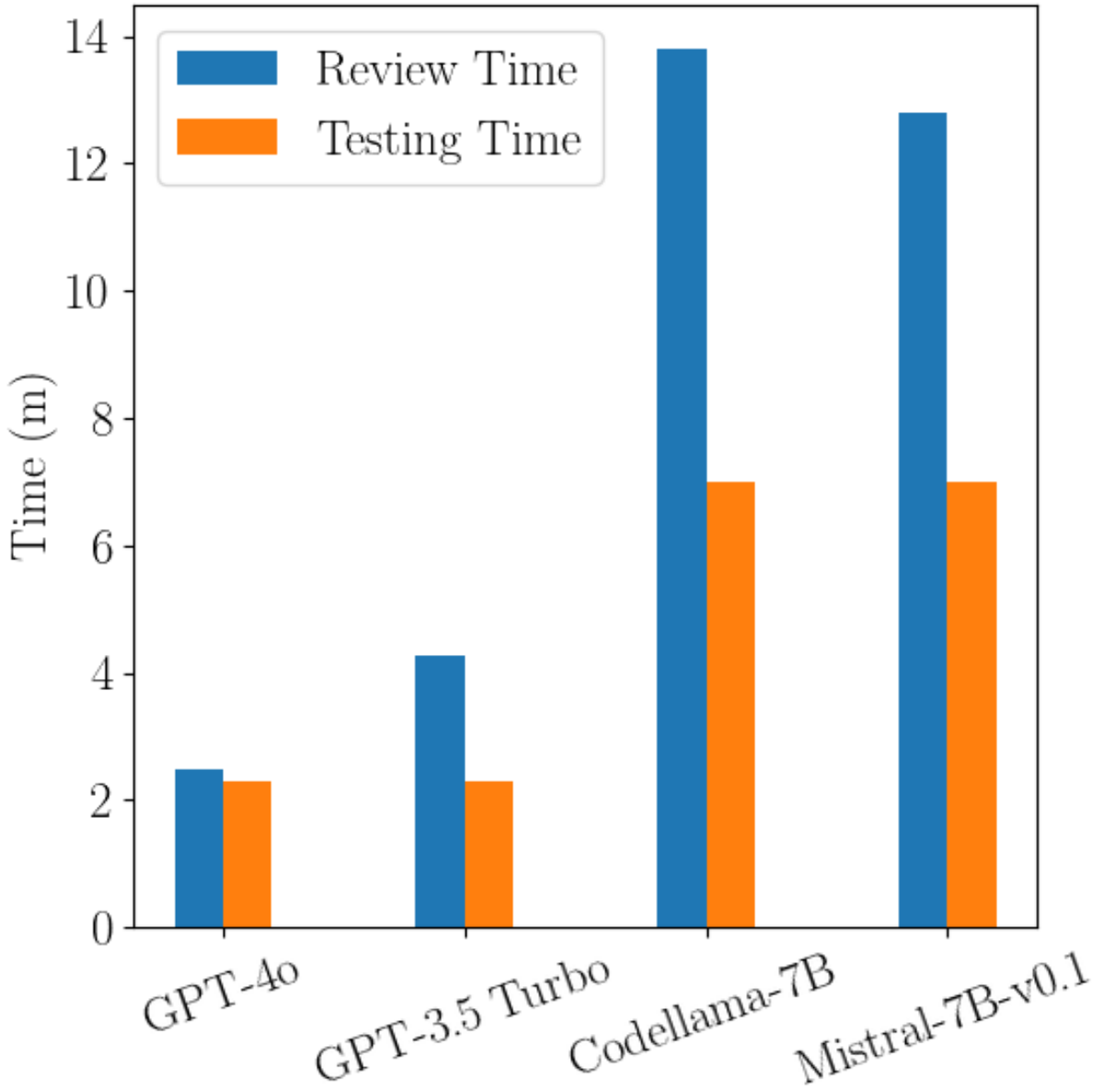


A. Dhruv and A. Dubey, “Leveraging Large Language Models for CodeTranslation and Software Development in Scientific Computing,”Proceedings of PASC-25
https://dl.acm.org/doi/abs/10.1145/3732775.3733572
On arXiv : doi: 10.48550/arXiv.2410.24119.

Model Sensitivity Study

The time shown in the figure is the time for developer review and testing per file. Higher times reflect less correct code, and therefore more iterations for getting to correct code

- CodeLlama-7B and Mistral-7B required significant manual review and testing
- GPT-3.5 Turbo and GPT-4o demonstrated strong adherence to chat template, reducing developer time
- Overall, GPT-4o delivered the best performance due to its high parameter count and multi-step reasoning capabilities



Observations About Code Translation

Excellent Results – significant time saved in translation

- ❑ Pure syntactical conversion is almost 100% right, not so much the whole code
- ❑ Models with high parameter count (~trillion) and multi-step reasoning capabilities are desirable for these tasks
- ❑ Model fine-tuning can potentially alleviate a lot of issues for low parameter count models. Fine-tuning is a common practice for application specific LLM use
- ❑ Ramp-on methodology – wholesale conversions almost always wrong and difficult to analyze
  - ➢ A related question -- how can people be trained to use tools effectively?

CONCLUSIONS

- LLMs have a great deal of promise in coding related tasks
- A judicious combination of scripts, LLMs and human-in-the-loop have already helped in reducing tedious tasks
- They are still very far from being reliable assistants for non-trivial coding on their own
- We have had no luck so far with code refactoring – explaining the full context of the existing code has been too difficult so far