MOX-Report No. 31/2023

Assessing ChatGPT for coding finite element methods

Orlando, G.

MOX, Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

mox-dmat@polimi.it https://mox.polimi.it

Assessing ChatGPT for coding finite element methods

Giuseppe Orlando(1)

(1) MOX - Dipartimento di Matematica, Politecnico di Milano
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
`giuseppe.orlando@polimi.it`

Keywords: ChatGPT, Finite element method, Artificial Intelligence, Programming languages, Natural language processing.

Abstract

ChatGPT is a language model trained by OpenAI to follow an instruction in a prompt and to provide a detailed response. We investigate the capabilities of ChatGPT to generate codes which implement the finite element method. The Finite element method (FEM) is a popular technique for the numerical solution of partial differential equations (PDEs). More specifically, we analyze the codes generated for two open source platforms: *deal.II*, a C++ software library, and FEniCS, for which we focus on its Python interface. We consider as benchmark problems the Poisson equation and a linear advection problem. The outcomes suggest that ChatGPT can be employed as initial building block to write finite element codes, but certain limitations and failures, which require further improvement of the machine learning model and a human supervision, are still present.

1 Introduction

Partial differential equations (PDEs) naturally arise in mathematical and engineering modelling [33]. Typical areas involving PDEs are e.g. continuum mechanics [35] and fluid mechanics [13]. The finite element method (FEM) is a widely employed technique for the numerical solution of PDEs. We refer to [22, 36] for an introduction and a review of this method. Several both open source libraries, such as *deal.II* [1], FEniCS [2], FreeFEM [3], and commercial softwares, such as COMSOL [4], are based on the finite element method to compute a numerical approximation of PDEs. Developing and implementing a finite element code is a challenging task, which requires a significant programming knowledge in order to handle several structures and data. An attempt to reduce such difficulties is represented by the so-called Domain-specific languages (DSLs) [19, 34], in which the designer formulates a problem in terms of PDEs and a "backend" takes care of generating the code for a specific language [34]. Recently, there has been a growing interest in the potential of Natural Language Processing (NLP) and Artificial Intelligence (AI) to automatically generate numerical algorithms. Among the various AI-based models, ChatGPT [5] is nowadays one of the most popular and successful. ChatGPT is a language model trained by OpenAI on large amounts of text data to follow an instruction and to provide natural language text with high detail and accuracy. However, the use of ChatGPT for mathematical capabilities and for programming numerical algorithms is still at early stages [20, 24]. In this work, we assess the ability of this model to generate finite element codes. To the best of our knowledge, this survey is not yet present in the literature and is performed here for the first time. More specifically, we generate programs implemented in the frameworks of *deal.II* [11, 12] and FEniCS [27, 28]. We consider as benchmark problems the Poisson equation and a linear advection problem. The paper is structured as follows. In Section 2, we analyze the codes generated by ChatGPT for *deal.II*, whereas in Section 3, we go through codes generated for FEniCS. Finally, some conclusions are reported in Section 4.

2 Finite element codes in *deal.II*

In this Section, we start investigating the capabilities of ChatGPT to generate finite element codes. It is worth to point out that the model language is able to produce completely different responses to the same question, since it has been massively trained over a variety of texts [24]. We now consider programs implemented in the framework of *deal.II*, a C++ software library for the creation of finite element codes [11, 12] widely used in several fields, such as Computational Fluid Dynamics (CFD) [17, 30, 31], and geodynamics [25]. We directly ask ChatGPT information about the library.

GO Hi ChatGPT, do you know deal.II?

Yes, I'm familiar with deal.II. It is a C++ software library for the numerical solution of partial differential equations using finite element methods. deal.II is an open-source project and is widely used in academic and industrial research for a variety of applications, such as fluid dynamics, solid mechanics, and electromagnetics. The library provides a modular framework for implementing complex numerical methods and supports parallel computing on a variety of architectures.

We then ask ChatGPT to write a code in *deal.II* which solves the Poisson equation [33]

$$-\Delta u = f \qquad \text{in } \Omega. \tag{1}$$

Equation (1) has to be supplied with suitable boundary conditions.

Write a code in deal.II which solves the Poisson equation.

Sure, here is an example code in deal.II that solves the Poisson equation:

```
#include <deal.II/base/function.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/utilities.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/full_matrix.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>

#include <iostream>
#include <fstream>
#include <cmath>

using namespace dealii;

// Define the Poisson equation class
template <int dim>
class PoissonEquation {
public:
  PoissonEquation ();
  void run ();
```

```
private:
  void setup_system ();
  void assemble_system ();
  void solve ();
  void output_results () const;

  Triangulation<dim> triangulation;
  FE_Q<dim> fe;
  DoFHandler<dim> dof_handler;

  SparsityPattern sparsity_pattern;
  SparseMatrix<double> system_matrix;

  Vector<double> solution;
  Vector<double> system_rhs;
};

// Constructor
template <int dim>
PoissonEquation<dim>::PoissonEquation ()
  : fe(1)
  , dof_handler(triangulation)
{}

// Member function for setting up the finite element system
template <int dim>
void PoissonEquation<dim>::setup_system () {
  dof_handler.distribute_dofs(fe);
  solution.reinit(dof_handler.n_dofs());
  system_rhs.reinit(dof_handler.n_dofs());

  DynamicSparsityPattern dsp(dof_handler.n_dofs(),
   dof_handler.n_dofs());
  DoFTools::make_sparsity_pattern(dof_handler, dsp);
  sparsity_pattern.copy_from(dsp);
  system_matrix.reinit(sparsity_pattern);
}

// Member function for assembling the finite element system
template <int dim>
void PoissonEquation<dim>::assemble_system () {
  QGauss<dim> quadrature_formula(fe.degree+1);
  FEValues<dim> fe_values(fe, quadrature_formula,
   update_values | update_gradients | update_JxW_values);

  const unsigned int dofs_per_cell = fe.dofs_per_cell;
  const unsigned int n_q_points = quadrature_formula.size();

  FullMatrix<double> cell_matrix(dofs_per_cell,
   dofs_per_cell);
```

```
 Vector<double> cell_rhs(dofs_per_cell);

 std::vector<types::global_dof_index> local_dof_indices(
  dofs_per_cell);

 typename DoFHandler<dim>::active_cell_iterator cell =
  dof_handler.begin_active(), endc = dof_handler.end();
 for (; cell!=endc; ++cell) {
  cell_matrix = 0;
  cell_rhs = 0;

  fe_values.reinit(cell);

  for (unsigned int q=0; q<n_q_points; ++q) {
   for (unsigned int i=0; i<dofs_per_cell; ++i) {
    for (unsigned int j=0; j<dofs_per_cell; ++j) {
     cell_matrix(i,j) += fe_values.shape_grad(i,q) *
  fe_values.shape_grad(j,q) * fe_values.JxW(q);
    }
```

As reported in [24], if the required code contains many lines, a server interruption often happens. We ask ChatGPT to continue the previous code

GO You stopped. Can you finish?

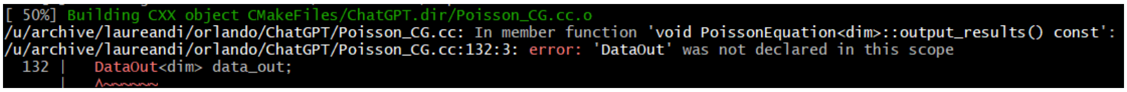
```
    const double rhs_value = 1.0;
    cell_rhs(i) += fe_values.shape_value(i,q) *
  rhs_value * fe_values.JxW(q);
   }
  }

  cell->get_dof_indices(local_dof_indices);
  for (unsigned int i=0; i<dofs_per_cell; ++i) {
   for (unsigned int j=0; j<dofs_per_cell; ++j) {
    system_matrix.add(local_dof_indices[i],
  local_dof_indices[j], cell_matrix(i,j));
   }
   system_rhs(local_dof_indices[i]) += cell_rhs(i);
  }
 }

 std::map<types::global_dof_index, double> boundary_values;
 VectorTools::interpolate_boundary_values(dof_handler, 0,
  ZeroFunction<dim>(), boundary_values);
 MatrixTools::apply_boundary_values(boundary_values,
  system_matrix, solution, system_rhs);
}
```

```
// Member function for solving the finite element system
template <int dim>
void PoissonEquation<dim>::solve () {
  SolverControl solver_control(1000, 1e-12);
  SolverCG<> solver(solver_control);

  PreconditionSSOR<> preconditioner;
  preconditioner.initialize(system_matrix, 1.2);

  solver.solve(system_matrix, solution, system_rhs,
   preconditioner);

  std::cout << "   " << solver_control.last_step()
            << " CG iterations needed to obtain convergence.
   "
            << std::endl;
}

// Member function for outputting the solution
template <int dim>
void PoissonEquation<dim>::output_results () const {
  DataOut<dim> data_out;
  data_out.attach_dof_handler(dof_handler);
  data_out.add_data_vector(solution, "solution");
  data_out.build_patches();

  std::ofstream output("solution.gpl");
  data_out.write_gnuplot(output);
}

// Member function for running the Poisson equation solver
template <int dim>
void PoissonEquation<dim>::run () {
  GridGenerator::hyper_cube(triangulation, -1, 1);
  triangulation.refine_global(4);

  std::cout << "Number of active cells: "
            << triangulation.n_active_cells()
            << std::endl
            << "Total number of cells: "
            << triangulation.n_cells()
            << std::endl;

  setup_system();
  assemble_system();
  solve();
  output_results();
}

// Main function
```

```
int main () {
  deallog.depth_console(2);

  PoissonEquation<2> poisson_equation;
  poisson_equation.run();

  return 0;
}
```

This code solves the Poisson equation on a 2D hypercube domain using linear finite elements. The solution is written to a file named "solution.gpl" in gnuplot format.

The combination of the two parts produces a complete code. We let ChatGPT choose the domain, the right-hand side, and the boundary conditions, as well as technical details, such as the number of elements, the polynomial degree or the preconditioner. One can specify also these details in the request, as we will see later on. In the generated program, $\Omega = (-1,1)^2$, $f = 1$, and homogeneous Dirichlet boundary conditions have been considered. The code follows *deal.II* tutorials, see e.g. [6]. Nevertheless, as reported in Figure 1, a compilation error occurs since the class DataOut has not been declared.



```
[ 50%] Building CXX object CMakeFiles/ChatGPT.dir/Poisson_CG.cc.o
/u/archive/laureandi/orlando/ChatGPT/Poisson_CG.cc: In member function 'void PoissonEquation<dim>::output_results() const':
/u/archive/laureandi/orlando/ChatGPT/Poisson_CG.cc:132:3: error: 'DataOut' was not declared in this scope
  132 |   DataOut<dim> data_out;
      |   ^~~~~~~
```

Figure 1: Compilation error with the code automatically generated in *deal.II* to solve the Poisson equation.

Hence, we need to add the proper header which defines the missing data structure in order to compile and execute the program.

```
#include <deal.II/numerics/data_out.h> // Manually added
```

Figure 2 shows the numerical solution of the Poisson equation obtained with the generated code. This first test shows that ChatGPT is able to write a code which correctly implements the finite element method for the Poisson equation in *deal.II*. However, a human supervision is necessary to fulfil the proper dependencies.

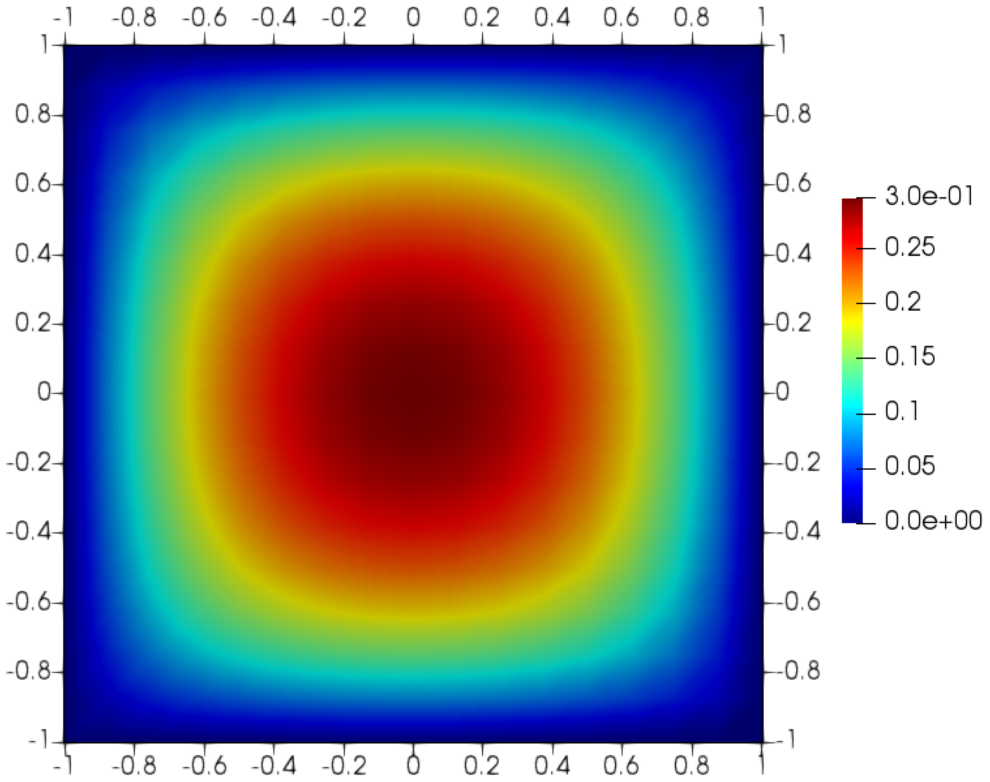


Figure 2: Solution of the Poisson equation (1) with the code generated by ChatGPT in *deal.II*, without specifying any detail in the request.

The previous code exploits only basic features of the library, which also includes more advanced tools, such as MPI parallelization and Adaptive Mesh Refinement (AMR). Code parallelization is a key element of High Performance Computing (HPC), see e.g. the discussion in [23], whereas AMR allows to enhance the resolution only in the regions of interest. We ask ChatGPT to modify the previous code so as to include MPI parallelization and AMR. Since the resulting code is much more complex, the issue of server interruption becomes severe. However, after a few attempts, we obtain the following code:

> Can you modify the previous code so as to include parallelization with MPI and Adaptive Mesh Refinement?

> ... You stopped. Can you finish?

> Apologies for the interruption, here is the complete code for solving Poisson's equation with MPI parallelization and Adaptive Mesh Refinement in deal.II:

```
#include <deal.II/base/logstream.h>
#include <deal.II/base/mpi.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/utilities.h>
#include <deal.II/base/function.h>
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/parameter_handler.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/grid_refinement.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_accessor.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_q.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/distributed/tria.h>
#include <deal.II/distributed/grid_refinement.h>

#include <fstream>
#include <iostream>

namespace PoissonEquation
{
  using namespace dealii;

  template <int dim>
  class RightHandSide : public Function<dim>
  {
  public:
    virtual double value(const Point<dim> &p, const unsigned
    int component = 0) const;
  };

  template <int dim>
  class PoissonEquation
  {
  public:
    PoissonEquation(const unsigned int degree, const
   unsigned int n_refinements, const bool
   use_adaptive_refinement);

    void run();

  private:
    void setup_system();
  void assemble_system();
  void solve();
  void refine_grid();
  void output_results(const unsigned int cycle) const;

  MPI_Comm mpi_communicator;

  const unsigned int degree;
  const unsigned int n_refinements;
  const bool use_adaptive_refinement;

  parallel::distributed::Triangulation<dim> triangulation;
  FE_Q<dim> fe;
  DoFHandler<dim> dof_handler;

  SparsityPattern sparsity_pattern;
  SparseMatrix<double> system_matrix;

  Vector<double> solution;
  Vector<double> system_rhs;

  ConditionalOStream pcout;
};

template <int dim>
double RightHandSide<dim>::value(const Point<dim> &p,
 const unsigned int /*component*/) const
{
  return 1.0;
}

template <int dim>
PoissonEquation<dim>::PoissonEquation(const unsigned int
 degree, const unsigned int n_refinements, const bool
 use_adaptive_refinement)
  : mpi_communicator(MPI_COMM_WORLD),
    degree(degree),
    n_refinements(n_refinements),
    use_adaptive_refinement(use_adaptive_refinement),
    fe(degree),
    dof_handler(triangulation),
    pcout(std::cout, Utilities::MPI::this_mpi_process(
 mpi_communicator) == 0)
{
}

template <int dim>
void PoissonEquation<dim>::setup_system()
{
  dof_handler.distribute_dofs(fe);
```

```
    DynamicSparsityPattern dsp(dof_handler.n_dofs());
    DoFTools::make_sparsity_pattern(dof_handler, dsp);
    sparsity_pattern.copy_from(dsp);

    system_matrix.reinit(sp
```

You stopped. Continue

```
}

template <int dim>
void PoissonEquation<dim>::assemble_system()
{
  const QGauss<dim> quadrature_formula(degree + 1);
  FEValues<dim> fe_values(fe, quadrature_formula,
   update_values | update_gradients | update_JxW_values);

  const unsigned int dofs_per_cell = fe.dofs_per_cell;
  const unsigned int n_q_points = quadrature_formula.size();

  FullMatrix<double> cell_matrix(dofs_per_cell,
   dofs_per_cell);
  Vector<double> cell_rhs(dofs_per_cell);

  std::vector<types::global_dof_index> local_dof_indices(
   dofs_per_cell);

  for (const auto &cell : dof_handler.active_cell_iterators
   ())
  {
    fe_values.reinit(cell);

    cell_matrix = 0;
    cell_rhs = 0;

    for (unsigned int q = 0; q < n_q_points; ++q)
    {
      for (unsigned int i = 0; i < dofs_per_cell; ++i)
      {
        const auto &phi_i = fe_values.shape_value(i, q);
        const auto &grad_phi_i = fe_values.shape_grad(i, q);

        for (unsigned int j = 0; j < dofs_per_cell; ++j)
        {
          const auto &phi_j = fe_values.shape_value(j, q);
          const auto &grad_phi_j = fe_values.shape_grad(j, q
   );
```

```
          cell_matrix(i, j) += (grad_phi_i * grad_phi_j) *
   fe_values.JxW(q);
        }

        cell_rhs(i) += phi_i * RightHandSide<dim>().value(
   fe_values.quadrature_point(q)) * fe_values.JxW(q);
      }
    }

    cell->get_dof_indices(local_dof_indices);

    for (unsigned int i = 0; i < dofs_per_cell; ++i)
    {
      for (unsigned int j = 0; j < dofs_per_cell; ++j)
      {
        system_matrix.add(local_dof_indices[i],
   local_dof_indices[j], cell_matrix(i, j));
      }
      system_rhs(local_dof_indices[i]) += cell_rhs(i);
    }
  }

  Constraints<double> constraints;
  DoFTools::make_hanging_node_constraints(dof_handler,
   constraints);
  constraints.close();

  constraints.condense(system_matrix);
  constraints.condense(system_rhs);
}

template <int dim>
void PoissonEquation<dim>::solve()
{
  SolverControl solver_control(system_matrix.n_rows(), 1e
   -12);
  SolverCG<Vector<double>> solver(solver_control);

  PreconditionSSOR<SparseMatrix<double>> preconditioner;
  preconditioner.initialize(system_matrix, 1.2);

  solver.solve(system_matrix, solution, system_rhs,
   preconditioner);

  constraints.distribute(solution);
}

template <int dim>
void PoissonEquation<dim>::refine_grid()
{
  if (use_adaptive_refinement)
  {
    Vector<float> estimated_error_per_cell(triangulation.
   n_active_cells());
    KellyErrorEstimator<dim>::estimate(dof_handler, QGauss<
   dim - 1>(degree + 1), {}, solution,
   estimated_error_per_cell);

    parallel::distributed::GridRefinement::
   refine_and_coarsen_fixed_number(triangulation,

                  estimated_error_per_cell,

                  0.3, // refine threshold

                  0.03 // coarsen threshold
    );

    triangulation.execute_coarsening_and_refinement();
  }
  else
  {
    triangulation.refine_global(1);
  }
}

template <int dim>
void Po
```

Continue

```
issonEquation<dim>::output_results(const unsigned int cycle)
    const
{
DataOut<dim> data_out;

data_out.attach_dof_handler(dof_handler);
data_out.add_data_vector(solution, "solution");

data_out.build_patches();

std::string filename = "solution-" + std::to_string(cycle) +
    "." + Utilities::MPI::to_string(triangulation.
   locally_owned_subdomain(), 4) + ".vtu";
std::ofstream output(filename);
data_out.write_vtu(output);
}
```

```
template <int dim>
void PoissonEquation<dim>::run()
{
for (unsigned int cycle = 0; cycle < n_cycles; ++cycle)
{
  std::cout << "Cycle " << cycle << ':' << std::endl;

  if (cycle == 0)
  {
    GridGenerator::hyper_cube(triangulation);
    triangulation.refine_global(2);
  }
  else
  {
    refine_grid();
  }

  setup_system();
  assemble_system();
  solve();
  output_results(cycle);
}
}

} // namespace Step8
} // namespace Poisson

int main(int argc, char *argv[])
{
try
{
Utilities::MPI::MPI_InitFinalize mpi(argc, argv);

if (argc != 2)
{
  std::cerr << "Usage: " << argv[0] << " <number of cycles>"
    << std::endl;
  std::exit(1);
}

const unsigned int n_cycles = std::atoi(argv[1]);

Poisson::Step8::PoissonEquation<2> poisson_equation(n_cycles
   , true);

poisson_equation.run();

}
catch (std::exception &exc)
{
```

```
std::cerr << std::endl
<< std::endl
<< "----------------------------------------------------"
<< std::endl;
std::cerr << "Exception on processing: " << std::endl
<< exc.what() << std::endl
<< "Aborting!" << std::endl
<< "----------------------------------------------------"
<< std::endl;

return 1;

}
catch (...)
{
std::cerr << std::endl
<< std::endl
<< "----------------------------------------------------"
<< std::endl;
std::cerr << "Unknown exception!" << std::endl
<< "Aborting!" << std::endl
<< "----------------------------------------------------"
<< std::endl;
return 1;
}

return 0;
}
```

The language model completed our request, even though the first block is clearly incomplete. A knowledge of the employed infrastructure is at this stage necessary to go through the code. Indeed, while the program adopts MPI functionalities, such as `parallel::distributed::Triangulation<dim> triangulation` or `mpi_communicator`, the proper variables to distribute data among the processors are not employed. For this purpose, PETSc [7] or Trilinos [8] capabilities should be used, see e.g.[9]. We ask ChatGPT to modify the code:

I apologize again, but it seems that the previous code does not employ a proper data structure for parallelization. Am I right? Can you correct?

You are correct, the previous code does not employ a proper data structure for parallelization. To enable parallelization using MPI, we need to use a distributed triangulation instead of a regular triangulation. Here is an updated code that uses a distributed triangulation and parallel data structures:

```
#include <deal.II/base/conditional_ostream.h>
#include <deal.II/base/function.h>
#include <deal.II/base/index_set.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/multithread_info.h>
#include <deal.II/base/mpi.h>
#include <deal.II/base/parameter_handler.h>
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/utilities.h>

#include <deal.II/distributed/grid_refinement.h>
#include <deal.II/distributed/tria.h>

#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_renumbering.h>
#include <deal.II/dofs/dof_tools.h>

#include <deal.II/fe/fe_q.h>

#include <deal.II/grid/grid_generator.h>
#include <deal.II/grid/manifold_lib.h>

#include <deal.II/lac/dynamic_sparsity_pattern.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/vector.h>

#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/error_estimator.h>
#include <deal.II/numerics/vector_tools.h>

#include <iostream>
#include <fstream>
#include <iomanip>
#include <sstream>

namespace Poisson
{
  namespace Step8
  {
    using namespace dealii;

    template <int dim>
    class PoissonEquation
    {
    public:
      PoissonEquation(const unsigned int n_cycles, const
   bool adaptive_mesh_refinement);

      void run();
```

```
private:
  void setup_system();
  void assemble_system();
  void solve();
  void refine_grid();
  void output_results(const unsigned int cycle) const;

  const unsigned int n_cycles;
  const bool adaptive_mesh_refinement;

  MPI_Comm mpi_communicator;
  const unsigned int n_mpi_processes;
  const unsigned int this_mpi_process;

  ConditionalOStream pcout;

  Triangulation<dim> triangulation;
  FE_Q<dim> fe;
  DoFHandler<dim> dof_handler;

  IndexSet locally_owned_dofs;
  IndexSet locally_relevant_dofs;

  std::vector<types::global_dof_index> locally_owned_dofs_indices;

  SparsityPattern sparsity_pattern;
  SparseMatrix<double> system_matrix;

  Vector<double> solution;
  Vector<double> system_rhs;
};

template <int dim>
PoissonEquation<dim>::PoissonEquation(const unsigned int n_cycles, const bool adaptive_mesh_refinement)
  : n_cycles(n_cycles), adaptive_mesh_refinement(adaptive_mesh_refinement),
    mpi_communicator(MPI_COMM_WORLD),
    n_mpi_processes(Utilities::MPI::n_mpi_processes(mpi_communicator)),
    this_mpi_process(Utilities::MPI::this_mpi_process(mpi_communicator)),
    pcout(std::cout, (this_mpi_process == 0))
{
}

template <int dim>
void PoissonEquation<dim>::setup_system()
{
  TimerOutput::Scope t(computing_timer, "setup");

  dof_handler.distribute_dofs(fe);

  locally_owned_dofs = dof_handler.locally_owned_dofs();
  DoFTools::extract_locally_relevant_d
```

Continue

```
ofs(dof_handler, locally_relevant_dofs);
locally_relevant_dofs = locally_owned_dofs;
locally_owned_dofs_indices = locally_owned_dofs.indices();

DynamicSparsityPattern dsp(locally_owned_dofs);
DoFTools::make_sparsity_pattern(dof_handler, dsp);
SparsityTools::distribute_sparsity_pattern(dsp,
                                            dof_handler.
   n_locally_owned_dofs_per_processor(),
                                            mpi_communicator
   ,

   locally_owned_dofs_indices,

   locally_relevant_dofs);

sparsity_pattern.copy_from(dsp);

system_matrix.reinit(locally_owned_dofs, locally_owned_dofs,
    sparsity_pattern, mpi_communicator);
solution.reinit(locally_owned_dofs, mpi_communicator);
system_rhs.reinit(locally_owned_dofs, mpi_communicator);
}

template <int dim>
void PoissonEquation<dim>::assemble_system()
{
TimerOutput::Scope t(computing_timer, "assembly");

system_matrix = 0;
system_rhs = 0;

QGauss<dim> quadrature_formula(fe.degree + 1);
FEValues<dim> fe_values(fe, quadrature_formula,
   update_values | update_gradients | update_JxW_values);

const unsigned int dofs_per_cell = fe.dofs_per_cell;
const unsigned int n_q_points = quadrature_formula.size();
```

```
FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell)
   ;
Vector<double> cell_rhs(dofs_per_cell);

std::vector<types::global_dof_index> local_dof_indices(
   dofs_per_cell);

for (const auto &cell : dof_handler.active_cell_iterators())
  {
    if (!cell->is_locally_owned())
      {
        continue;
      }

    cell_matrix = 0;
    cell_rhs = 0;

    fe_values.reinit(cell);

    for (unsigned int q_index = 0; q_index < n_q_points; ++
   q_index)
      {
        const double JxW = fe_values.JxW(q_index);

        for (unsigned int i = 0; i < dofs_per_cell; ++i)
          {
            const unsigned int component_i = fe.
   system_to_component_index(i).first;

            for (unsigned int j = 0; j < dofs_per_cell; ++j)
              {
                const unsigned int component_j = fe.
   system_to_component_index(j).first;

                cell_matrix(i, j) +=
                  fe_values.shape_grad(i, q_index)[
   component_i] * fe_values.shape_grad(j, q_index)[
   component_j] * JxW;
              }

            cell_rhs(i) += fe_values.shape_value(i, q_index)
    * 1.0 * JxW;
          }
      }

    cell->get_dof_indices(local_dof_indices);

    for (unsigned int i = 0; i < dofs_per_cell; ++i)
      {
        for (unsigned int j = 0; j < dofs_per_cell; ++j)
```

```
          {
            system_matrix.add(local_dof_indices[i],
   local_dof_indices[j], cell_matrix(i, j));
          }

        system_rhs(local_dof_indices[i]) += cell_rhs(i);
      }
  }

system_matrix.compress(VectorOperation::add);
system_rhs.compress(VectorOperation::add);
}

template <int dim>
void PoissonEquation<dim>::solve()
{
TimerOutput::Scope t(computing_timer, "solve");

SolverControl solver_control(system_matrix.m(), 1e-12 *
   system_rhs.l2_norm());
SolverCG<Vector<double>> solver(solver_control,
   mpi_communicator);
```

GO Continue

```
PreconditionSSOR<SparseMatrix<double>> preconditioner;
preconditioner.initialize(system_matrix, 1.2);

solver.solve(system_matrix, solution, system_rhs,
   preconditioner);

constraints.distribute(solution);

solution.update_ghost_values();

output_results();
}

template <int dim>
void PoissonEquation<dim>::refine_grid()
{
TimerOutput::Scope t(computing_timer, "refine");

Vector<float> estimated_error_per_cell(triangulation.
   n_active_cells());

KellyErrorEstimator<dim>::estimate(dof_handler,
                                                QGauss<dim - 1>(fe.
    degree + 1),
                                                {},
                                                solution,
                                                estimated_error_per_cell
    );

GridRefinement::refine_and_coarsen_fixed_number(
    triangulation,

    estimated_error_per_cell,
                                                                0.5,
                                                                0.03);

triangulation.execute_coarsening_and_refinement();

setup_system();

assemble_system();
}

template <int dim>
void PoissonEquation<dim>::output_results() const
{
TimerOutput::Scope t(computing_timer, "output");

DataOut<dim> data_out;

data_out.attach_dof_handler(dof_handler);

data_out.add_data_vector(solution, "solution");

data_out.build_patches();

std::string filename = "solution-";

filename += Utilities::int_to_string(triangulation.
    n_global_active_cells(), 4);

filename += ".vtu";

std::ofstream output(filename);
data_out.write_vtu(output);
}

template <int dim>
void PoissonEquation<dim>::run()
{
TimerOutput::Scope t(computing_timer, "total");

make_grid();
```

```
setup_system();

assemble_system();

solve();

for (unsigned int cycle = 0; cycle < 6; ++cycle)
  {
    refine_grid();

    assemble_system();

    solve();
  }

computing_timer.print_summary();

computing_timer.print(std::cout, TimerOutput::summary,
   TimerOutput::cpu_times);
}

} // namespace Step10
} // namespace dealii

int main(int argc, char *argv[])
{
try
{
using namespace dealii;
using namespace Step10;

Utilities::MPI::MPI_InitFinalize mpi_initialization(argc,
   argv, numbers::invalid_unsigned_int);

PoissonEquation<2> poisson_equation_solver;
poisson_equation_solver.run();
}

catch (std::exception &exc)
{
std::cerr << std::endl
<< std::endl
<< "----------------------------------------------------" <<
    std::endl;
std::cerr << "Exception on processing: " << std::endl
<< exc.what() << std::endl
<< "Aborting!" << std::endl
<< "----------------------------------------------------" <<
    std::endl;
```

```
return 1;
}

catch (...)
{
std::cerr << std::endl
<< std::endl
<< "----------------------------------------------------" <<
    std::endl;
std::cerr << "Unknown exception!" << std::endl
<< "Aborting!" << std::endl
<< "----------------------------------------------------" <<
    std::endl;
return 1;
}

return 0;
}
```

The situation is improved, since proper structures like `locally_relevant_dofs` and `locally_owned_dofs` are used to distribute the unknowns among the processors. However, neither PETSc nor Trilinos have been employed, and, therefore, the mission of generating a complete code which employs MPI and AMR is failed. The outcomes of this test suggest that ChatGPT can be used as starting point for long and complex codes, but human supervision and contribution are still necessary. This consideration is even more valid in terms of programming conventions, e.g. indentation style, which should be significantly improved in relation to the provided output.

We now consider another important class of time dependent problems, namely hyperbolic equations [33]. In particular, we focus on the linear advection equation, which reads as follows:

$$\frac{\partial u}{\partial t} + \mathbf{c} \cdot \nabla u = 0 \qquad \text{in } \Omega, \tag{2}$$

supplied with suitable initial and boundary conditions, see e.g. the discussion in [32]. Here, u is the advected scalar field and $\mathbf{c}$ is the advection velocity. The so-called Discontinuous Galerkin (DG) method has been widely employed for the spatial discretization of hyperbolic problems, see among many others [15, 16, 29]. This choice is justified by the fact that hyperbolic problems are typically characterized by discontinuities in the solution [32]. The DG method is a finite element method with discontinuous basis functions and can therefore handle discontinuous solutions. A detailed description of the DG method is out of the scope of the present work and we refer to [14, 21] for an introduction to this method. A numerical flux has to be employed at the interface between two adjacent elements. A common choice for the linear advection problem is the upwind flux [14]. We then ask ChatGPT to generate a code which solves a linear advection problem in two dimensions using the DG method and an upwind flux:

Write a code in deal.II which solves an unsteady linear advection problem in 2D using the Discontinuous Galerkin method and an upwind flux.

Here is an example code in deal.II which solves an unsteady linear advection problem in 2D using the Discontinuous Galerkin method and an upwind flux:

```
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/lac/precondition.h>
#include <deal.II/grid/tria.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/vector_tools.h>
#include <deal.II/numerics/matrix_tools.h>
#include <fstream>
#include <iostream>

using namespace dealii;

template<int dim>
class AdvectionProblem
{
public:
  AdvectionProblem ();
  void run ();

private:
  void setup_system ();
  void assemble_system ();
  void solve ();
  void output_results () const;

  Triangulation<dim>     triangulation;
  FE_DGQ<dim>            fe;
  DoFHandler<dim>        dof_handler;

  Vector<double>         solution;
  Vector<double>         old_solution;
  SparseMatrix<double>   system_matrix;

  double                  time_step;
  double                  final_time;
  unsigned int            n_time_steps;
  unsigned int            fe_degree;
  double                  advection_speed;
};

template<int dim>
AdvectionProblem<dim>::AdvectionProblem ()
  :
  fe_degree (2),
  dof_handler (triangulation)
{}

template<int dim>
void AdvectionProblem<dim>::setup_system ()
{
  dof_handler.distribute_dofs (fe);

  solution.reinit (dof_handler.n_dofs());
  old_solution.reinit (dof_handler.n_dofs());

  system_matrix.reinit (dof_handler.n_dofs(), dof_handler.
   n_dofs());
}

template<int dim>
void AdvectionProblem<dim>::assemble_system ()
{
  QGauss<dim>  quadrature_formula(fe_degree+1);
  FEValues<dim> fe_values (fe, quadrature_formula,
                           update_values | update_gradients
   |
                           update_quadrature_points |
   update_JxW_values);

  const unsigned int   dofs_per_cell = fe.dofs_per_cell;
  const unsigned int   n_q_points    = quadrature_formula.
   size();

  FullMatrix<double>   cell_matrix (dofs_per_cell,
   dofs_per_cell);
  Vector<double>       cell_rhs (dofs_per_cell);

  std::vector<unsigned int> local_dof_indices (dofs_per_cell
   );

  for (const auto& cell : dof_handler.active_cell_iterators
   ())
    {
      cell_matrix = 0;
      cell_rhs = 0;
      fe_values.reinit (cell);

      for (unsigned int q=0; q<n_q_points; ++q)
        {
          const double advection_direction = advection_speed
    > 0 ? 1 : -1;

          for (unsigned int i=0; i<dofs_per_cell; ++i)
            {
              for (unsigned int j=0; j<dofs_per_cell; ++j)
                {
                  cell_matrix(i,j) += advection_direction *
                    fe_values.shape_grad (i, q) *
                    fe_values.shape (j, q) *
                    fe_values.JxW(q);
                }

              cell_rhs(i) += fe_values.shape_value (i, q) *
                old_solution(cell->vertex_dof_index(i,0))
```

Continue

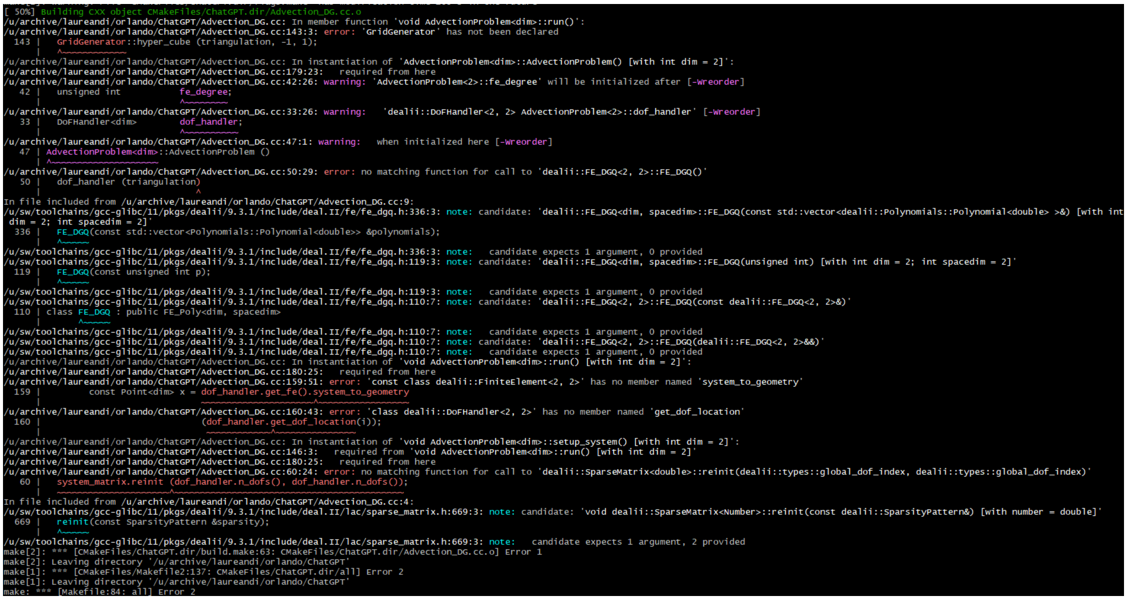
```
        * fe_values.JxW(q);
      }
    }

  cell->get_dof_indices (local_dof_indices);

  for (unsigned int i=0; i<dofs_per_cell; ++i)
    {
      for (unsigned int j=0; j<dofs_per_cell; ++j)
        {
          system_matrix.add (local_dof_indices[i],
                             local_dof_indices[j],
                             cell_matrix(i,j));
        }

      solution(local_dof_indices[i]) = old_solution(
   local_dof_indices[i]) +
        time_step*cell_rhs(i);
    }
  }
}

template<int dim>
void AdvectionProblem<dim>::solve ()
{
```

```
SparseMatrix<double> system_matrix_copy (system_matrix);
Vector<double> solution_copy (solution);

SolverControl solver_control (1000, 1e-12);
SolverCG<> cg (solver_control);

PreconditionIdentity preconditioner;

cg.solve (system_matrix_copy, solution_copy, solution,
   preconditioner);
}

template<int dim>
void AdvectionProblem<dim>::output_results () const
{
DataOut<dim> data_out;

data_out.attach_dof_handler (dof_handler);
data_out.add_data_vector (solution, "solution");
data_out.build_patches ();

std::ofstream output ("solution.gpl");
data_out.write_gnuplot (output);
}

template<int dim>
void AdvectionProblem<dim>::run ()
{
GridGenerator::hyper_cube (triangulation, -1, 1);
triangulation.refine_global (4);

setup_system ();

advection_speed = 1.0;

time_step = 1.0/(double)fe_degree;
final_time = 1.0;
n_time_steps = (unsigned int)(final_time/time_step);

for (unsigned int n=0; n<=n_time_steps; ++n)
{
std::cout << "Time step " << n << std::endl;

  if (n == 0)
    {
      for (unsigned int i=0; i<solution.size(); ++i)
        {
          const Point<dim> x = dof_handler.get_fe().
   system_to_geometry
             (dof_handler.get_dof_location(i));
          solution(i) = std::sin(2.0*numbers::PI*x(0))*std::
```

```
   sin(2.0*numbers::PI*x(1));
        }
    }
  else
    {
      old_solution = solution;
      assemble_system ();
      solve ();
    }

  if (n % 10 == 0)
    {
      output_results ();
    }
  }
}

int main ()
{
deallog.depth_console (0);

AdvectionProblem<2> advection_problem;
advection_problem.run ();

return 0;
}
```

This code sets up a 2D hyper-cube and refines it globally 4 times. The ‘setup_system()‘ function initializes the finite element space, creates the vectors for the solution and the old solution, and initializes the sparse system matrix. The ‘assemble_system()‘ function loops over all cells in the triangulation and assembles the local contribution to the system matrix and right-hand side vector. The ‘solve()‘ function solves the linear system using a preconditioned conjugate gradient method. The ‘output_results()‘ function writes the solution to a file in gnuplot format. The ‘run()‘ function sets up the problem parameters, initializes the solution at time zero, and loops over all time steps, assembling and solving the system at each time step and outputting the solution every 10 time steps.

Notice that, in *deal.II* tutorials, only a steady state advection using the DG method is presented [10]. The language model is therefore able to employ autonomously variables related to a time discretization scheme, such as `old_solution`, and to update the solution for the following time step. However, the compilation of the previous program leads to several errors reported in Figure 3.



```
[ 50%] Building CXX object CMakeFiles/ChatGPT.dir/Advection_DG.cc.o
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc: In member function 'void AdvectionProblem<dim>::run()':
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:143:3: error: 'GridGenerator' has not been declared
  143 |   GridGenerator::hyper_cube (triangulation, -1, 1);
      |   ^~~~~~~~~~~~~
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc: In instantiation of 'AdvectionProblem<dim>::AdvectionProblem() [with int dim = 2]':
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:179:23:   required from here
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:42:26: warning: 'AdvectionProblem<2>::fe_degree' will be initialized after [-Wreorder]
   42 |   unsigned int           fe_degree;
      |                          ^~~~~~~~~
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:33:26: warning:   'dealii::DoFHandler<2, 2> AdvectionProblem<2>::dof_handler' [-Wreorder]
   33 |   DoFHandler<dim>        dof_handler;
      |                          ^~~~~~~~~~~
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:47:1: warning:   when initialized here [-Wreorder]
   47 | AdvectionProblem<dim>::AdvectionProblem ()
      | ^~~~~~~~~~~~~~~~~~~~~
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:50:29: error: no matching function for call to 'dealii::FE_DGQ<2, 2>::FE_DGQ()'
   50 |   dof_handler (triangulation)
      |                             ^
In file included from /u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:9:
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:336:3: note: candidate: 'dealii::FE_DGQ<dim, spacedim>::FE_DGQ(const std::vector<dealii::Polynomials::Polynomial<double> >&) [with int
 dim = 2; int spacedim = 2]'
  336 |   FE_DGQ(const std::vector<Polynomials::Polynomial<double>> &polynomials);
      |   ^~~~~~
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:336:3: note:   candidate expects 1 argument, 0 provided
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:119:3: note: candidate: 'dealii::FE_DGQ<dim, spacedim>::FE_DGQ(unsigned int) [with int dim = 2; int spacedim = 2]'
  119 |   FE_DGQ(const unsigned int p);
      |   ^~~~~~
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:119:3: note:   candidate expects 1 argument, 0 provided
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:110:7: note: candidate: 'dealii::FE_DGQ<2, 2>::FE_DGQ(const dealii::FE_DGQ<2, 2>&)'
  110 | class FE_DGQ : public FE_Poly<dim, spacedim>
      |       ^~~~~~
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:110:7: note:   candidate expects 1 argument, 0 provided
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:110:7: note: candidate: 'dealii::FE_DGQ<2, 2>::FE_DGQ(dealii::FE_DGQ<2, 2>&&)'
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/fe/fe_dgq.h:110:7: note:   candidate expects 1 argument, 0 provided
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc: In instantiation of 'void AdvectionProblem<dim>::run() [with int dim = 2]':
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:180:25:   required from here
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:159:51: error: 'const class dealii::FiniteElement<2, 2>' has no member named 'system_to_geometry'
  159 |         const Point<dim> x = dof_handler.get_fe().system_to_geometry
      |                              ~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:160:43: error: 'class dealii::DoFHandler<2, 2>' has no member named 'get_dof_location'
  160 |                              (dof_handler.get_dof_location(i));
      |                               ~~~~~~~~~~~~^~~~~~~~~~~~~~~
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc: In instantiation of 'void AdvectionProblem<dim>::setup_system() [with int dim = 2]':
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:146:3:   required from 'void AdvectionProblem<dim>::run() [with int dim = 2]'
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:180:25:   required from here
/u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:60:24: error: no matching function for call to 'dealii::SparseMatrix<double>::reinit(dealii::types::global_dof_index, dealii::types::global_dof_index)'
   60 |   system_matrix.reinit (dof_handler.n_dofs(), dof_handler.n_dofs());
      |   ~~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
In file included from /u/archive/laureandi/orlando/ChatGPT/Advection_DG.cc:4:
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/lac/sparse_matrix.h:669:3: note: candidate: 'void dealii::SparseMatrix<Number>::reinit(const dealii::SparsityPattern&) [with number = double]'
  669 |   reinit(const SparsityPattern &sparsity);
      |   ^~~~~~
/u/sw/toolchains/gcc-glibc/11/pkgs/dealii/9.3.1/include/deal.II/lac/sparse_matrix.h:669:3: note:   candidate expects 1 argument, 2 provided
make[2]: *** [CMakeFiles/ChatGPT.dir/build.make:63: CMakeFiles/ChatGPT.dir/Advection_DG.cc.o] Error 1
make[2]: Leaving directory '/u/archive/laureandi/orlando/ChatGPT'
make[1]: *** [CMakeFiles/Makefile2:137: CMakeFiles/ChatGPT.dir/all] Error 2
make[1]: Leaving directory '/u/archive/laureandi/orlando/ChatGPT'
make: *** [Makefile:84: all] Error 2
```

Figure 3: Compilation error with the code automatically generated in *deal.II* to solve the linear advection problem.

Moreover, as well as the server interruption, other issues are present in the implementation. First of all, the upwind flux is not employed. Afterwards, we notice that the advection velocity is a scalar instead of being a vector with two components. Finally, in the `assemble_system` function, the vector storing the numerical solution is modified. We point out these drawbacks to the language model so as to verify the proposed modifications:

> GO The previous code is characterized by some issues: ‘GridGenerator‘ has not been declared, ‘system_to_geometry‘ and ‘get_dof_location’ functions do not exist and ‘system_matrix.reinit‘ accepts a wrong number of arguments. Moreover, the upwind flux is not proper implemented since no loop over faces is present, the advection velocity should be a vector and not a scalar, and in the ‘assemble_system‘ function you should not modify the ‘solution‘ variable. Try to correct.

> You are correct, thank you for bringing these issues to my attention. Here is a corrected version of the code:

```
#include <deal.II/base/quadrature_lib.h>
#include <deal.II/base/function.h>
#include <deal.II/base/logstream.h>
#include <deal.II/base/timer.h>
#include <deal.II/lac/vector.h>
#include <deal.II/lac/sparse_matrix.h>
```

```
#include <deal.II/lac/precondition.h>
#include <deal.II/lac/solver_cg.h>
#include <deal.II/grid/tria.h>
#include <deal.II/grid/grid_generator.h>
#include <deal.II/dofs/dof_handler.h>
#include <deal.II/dofs/dof_tools.h>
#include <deal.II/fe/fe_dgq.h>
#include <deal.II/fe/fe_values.h>
#include <deal.II/numerics/data_out.h>
#include <deal.II/numerics/vector_tools.h>

#include <iostream>
#include <fstream>
#include <cmath>

using namespace dealii;

template<int dim>
class AdvectionProblem
{
public:
  AdvectionProblem ();
  void run ();

private:
  void setup_system ();
  void assemble_system ();
  void solve ();
  void output_results () const;

  Triangulation<dim>   triangulation;
  FE_DGQ<dim>          fe;
  DoFHandler<dim>      dof_handler;

  Vector<double>       solution;
  Vector<double>       old_solution;

  double               advection_speed;
  double               time_step;
  double               final_time;
  unsigned int         n_time_steps;

  void upwind_flux(const double &v, const Tensor<1, dim> &
   normal,
                   const double &u_minus, const double &
   u_plus, double &flux) const;
};

template<int dim>
AdvectionProblem<dim>::AdvectionProblem ()
  :
```

```
  fe(1),
  dof_handler(triangulation)
{}

template<int dim>
void AdvectionProblem<dim>::upwind_flux(const double &v,
   const Tensor<1, dim> &normal,
                                        const double &
   u_minus, const double &u_plus, double &flux) const
{
  if (v > 0)
    {
      flux = v * u_minus;
    }
  else
    {
      flux = v * u_plus;
    }
  flux *= normal[0];
  if (dim>1)
    flux += v * normal[1] * (u_minus + u_plus) / 2.;
}

template<int dim>
void AdvectionProblem<dim>::setup_system ()
{
  dof_handler.distribute_dofs (fe);

  std::cout << "   Number of degrees of freedom: "
            << dof_handler.n_dofs()
            << std::endl;

  solution.reinit (dof_handler.n_dofs());
  old_solution.reinit (dof_handler.n_dofs());
}

template<int dim>
void AdvectionProblem<dim>::assemble_system ()
{
  QGauss<dim>   quadrature_formula(fe.degree+1);
  const unsigned int n_q_points = quadrature_formula.size();

  FEValues<dim> fe_values (fe, quadrature_formula,
                           update_values     |
   update_gradients |
                           update_quadrature_points   |
   update_JxW_values);

  Vector<double>       cell_rhs;
  FullMatrix<double>   cell_matrix;
```

```
std::vector<unsigned int> local_dof_indices(fe.
 dofs_per_cell);

for (const auto &cell : dof_handler.active_cell_iterators
 ())
  {
    fe_values.reinit (cell);

    cell_rhs.reinit (fe.dofs_per_cell);
    cell_matrix.reinit (fe.dofs_per_cell, fe.dofs_per_cell
 );
```

Continue

```
for (unsigned int q=0; q<n_q_points; ++q)
  {
    const double u = fe_values.get_value(q, old_solution);
    Tensor<1, dim> advection_direction;
    advection_direction[0] = advection_speed;
    if (dim>1)
      advection_direction[1] = 0.;
    Tensor<1, dim> normal = fe_values.normal_vector(q);

    double flux;
    if (advection_speed >= 0)
      {
        upwind_flux(advection_speed, normal, u, fe_values.
 get_value(q, old_solution, fe_values.
 get_neighbor_face_index(0)), flux);
      }
    else
      {
        upwind_flux(advection_speed, normal, fe_values.
 get_value(q, old_solution, fe_values.
 get_neighbor_face_index(0)), u, flux);
      }
    for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
      {
        cell_rhs(i) += fe_values.shape_value(i,q) * flux *
  fe_values.JxW(q);
        for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
          {
            cell_matrix(i,j) += advection_direction *
 fe_values.shape_grad(i,q) *
                                fe_values.shape_value(j,q)
  * fe_values.JxW(q);
          }
        }
    }

  cell->get_dof_indices (local_dof_indices);
  for (unsigned int i=0; i<fe.dofs_per_cell; ++i)
    {
      for (unsigned int j=0; j<fe.dofs_per_cell; ++j)
        {
          system_matrix.add(local_dof_indices[i],
   local_dof_indices[j],
                            cell_matrix(i,j));
        }
      system_rhs(local_dof_indices[i]) += cell_rhs(i);
    }
  }
}

template<int dim>
void AdvectionProblem<dim>::solve ()
{
SparseDirectUMFPACK A_direct;
A_direct.initialize(system_matrix);
A_direct.vmult(solution, system_rhs);
}

template<int dim>
void AdvectionProblem<dim>::output_results () const
{
DataOut<dim> data_out;

data_out.attach_dof_handler (dof_handler);
data_out.add_data_vector (solution, "u");

data_out.build_patches ();

std::ofstream output ("solution.vtk");
data_out.write_vtk (output);
}

template<int dim>
void AdvectionProblem<dim>::run ()
{
advection_speed = 1.0;
time_step = 0.01;
final_time = 1.0;
n_time_steps = static_cast<unsigned int>(final_time/
   time_step);

GridGenerator::hyper_cube(triangulation, -1, 1);
triangulation.refine_global(4);
```

```
setup_system ();

Vector<double> tmp(dof_handler.n_dofs());
Vector<double> forcing_terms(dof_handler.n_dofs());

for (unsigned int i=0; i<n_time_steps; ++i)
{
std::cout << "Time step " << i << std::endl;
if (i == 0)
  {
    VectorTools::interpolate(dof_handler, ZeroFunction<dim
   >(),
                             old_solution);
    VectorTools::interpolate(dof_handler, ZeroFunction<dim
   >(),
                             solution);
  }
else
  {
    old_solution = solution;
  }

system_matrix.reinit (dof_handler.n_dofs(), dof_handler.
   n_dofs());
system_rhs.reinit (dof_handler.n_dofs());

assemble_system ();

solve ();

output_results ();
}
std::cout << " Solution computed." << std::endl;
}
```

Continue

```
int main ()
{
try
{
deallog.depth_console (0);
AdvectionProblem<2> advection_problem;
advection_problem.run ();
}

catch (std::exception &exc)
{
std::cerr << std::endl << std::endl
<< "------------------------------------------------------"
<< std::endl;
std::cerr << "Exception on processing: " << std::endl << exc
   .what()
<< std::endl << "Aborting!" << std::endl
<< "------------------------------------------------------"
<< std::endl;

  return 1;
}
catch (...)
{
std::cerr << std::endl << std::endl
<< "------------------------------------------------------"
<< std::endl;
std::cerr << "Unknown exception!" << std::endl << "Aborting!
   " << std::endl
<< "------------------------------------------------------"
<< std::endl;
return 1;
}

return 0;
}
```

The code is significantly changed and the auxiliary routine `upwind_flux` is proposed. However, one can easily check that the output variable `flux` is not employed in the `assemble_system` function. Moreover, other changes not related to the reported issues have been performed. In particular, we notice that a direct solver is employed instead of the Conjugate Gradient method (see e.g. [32] for a discussion on the differences between direct and iterative methods to solve a linear system), and the initial datum is initialized to zero, meaning that no field is actually transported. This test shows that the automatic implementation of complex numerical algorithms in a framework for which several functions are needed is still far away from being efficient and requires a relevant improvement of the machine learning model. Hence, ChatGPT can be employed to generate the initial building block of a long code, but human programming capabilities remain necessary.

3 Finite element codes in FEniCS

In this Section, we investigate finite element codes generated by ChatGPT in the framework of FEniCS [27, 28]. More specifically, we focus on its Python interface so as to assess the capability of the model to deal with different programming languages. As done in Section 2 for *deal.II*, we ask ChatGPT information about FEniCS

Hey ChatGPT, do you know FEniCS?

Yes, I am familiar with FEniCS. FEniCS is a popular open-source finite element analysis software for solving partial differential equations (PDEs). It provides a flexible and efficient platform for implementing and solving a wide range of PDE problems. It is written in Python and C++ and includes a high-level interface for easy problem specification and a low-level interface for advanced customization. FEniCS is widely used in the scientific and engineering communities for simulating complex physical phenomena in various fields such as fluid mechanics, solid mechanics, electromagnetics, and many others.

We then require to generate a code which solves the Poisson equation. For the sake of comparison with the results in Section 2, we ask to consider the domain $\Omega = (-1,1)^2$, homogeneous Dirichlet boundary conditions, $f = 1$ and linear finite elements.

Write a code which solves the Poisson equation in FEniCS through its Python interface. Consider $(-1,1)^2$ as domain, homogeneous Dirichlet boundary conditions, source term equal to 1 and linear finite elements.

Sure! Here's an example code that solves the Poisson equation in FEniCS through its Python interface using linear finite elements:

```
from fenics import *

# Define mesh and function space
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, ’P’, 1)

# Define boundary conditions
u_D = Constant(0.0)
def boundary(x, on_boundary):
    return on_boundary
bc = DirichletBC(V, u_D, boundary)

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Constant(1.0)
a = dot(grad(u), grad(v)) * dx
L = f * v * dx

# Compute solution
u = Function(V)
```

```
solve(a == L, u, bc)

# Plot solution
plot(u)

# Save solution to file in VTK format
vtkfile = File(’poisson.pvd’)
vtkfile << u
```

In this code, we first define the mesh and the function space using ‘UnitSquareMesh‘ and ‘FunctionSpace‘. We then define the homogeneous Dirichlet boundary conditions using ‘DirichletBC‘. Next, we define the variational problem using the trial and test functions ‘u‘ and ‘v‘, the source term ‘f‘, and the bilinear and linear forms ‘a‘ and ‘L‘. Finally, we solve the linear system using ‘solve‘, passing in the variational problem, the solution ‘u‘, and the boundary conditions ‘bc‘. We plot the solution using ‘plot‘, and save it to file in VTK format using ‘File‘ and ‘vtkfile « u‘.

Apparently, the code is implemented in a correct way. However, one can notice that the domain $\Omega = (0,1)^2$ has been employed instead of $\Omega = (-1,1)^2$. We ask therefore to correct the program

I am sorry, but the domain requested is not the unit square. The bottom left corner has coordinates (-1,-1). Please, try to correct the code

I apologize for the mistake in my previous response. Here is the corrected code to solve the Poisson equation in FEniCS with the specified domain and boundary conditions:

```
from fenics import *

# Define mesh and function space
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, ’P’, 1)

# Define boundary conditions
u_D = Constant(0.0)
def boundary(x, on_boundary):
    return on_boundary
bc = DirichletBC(V, u_D, boundary)

# Define variational problem
u = TrialFunction(V)
```

```
v = TestFunction(V)
f = Constant(1.0)
a = dot(grad(u), grad(v)) * dx
L = f * v * dx

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# Plot solution
plot(u)

# Save solution to file in VTK format
vtkfile = File('poisson.pvd')
vtkfile << u
```

To change the domain to (-1, 1) x (-1, 1), we can modify the first line of the code to create a mesh of that domain:

```
mesh = UnitSquareMesh(10, 10) # Change to
mesh = RectangleMesh(Point(-1, -1), Point(1, 1), 10, 10)
```

This will create a mesh with 10 x 10 elements in the domain (-1, 1) x (-1, 1).

The implemented code is still not correct, but a workaround to the issue is proposed. Figure 4 shows the numerical solution obtained with the correct program. The results are in agreement with those in Figure 2.

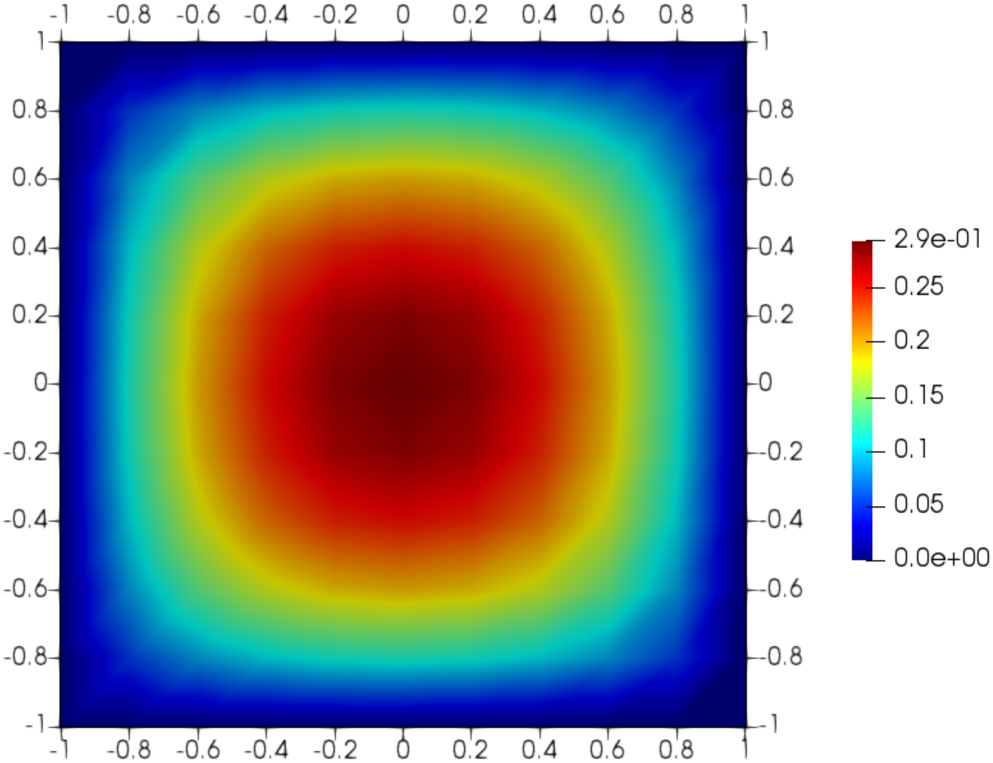


Figure 4: Solution of the Poisson equation (1) with the code generated by ChatGPT in FEniCS.

We then focus on the linear advection equation (2). As done in Section 2, we ask ChatGPT to generate a code for a linear advection problem in two dimensions using the Discontinuous Galerkin method and the upwind flux.

Write a code in FEniCS through its Python interface which solves an unsteady linear advection problem in 2D using the Discontinuous Galerkin method and an upwind flux. Save the solution at every time step in VTK.

Sure, here's a code that solves the 2D linear advection problem using the Discontinuous Galerkin method with an upwind flux and saves the solution at every time step in VTK using FEniCS through its Python interface.

```
from fenics import *
from dolfin import *
from mshr import *

# Define the mesh and function space
mesh = RectangleMesh(Point(0, 0), Point(1, 1), 32, 32)
```

```
V = FunctionSpace(mesh, ’DG’, 1)

# Define the upwind flux
w = Constant((1.0, 0.0))
u = Expression(’sin(pi*x[0])*cos(pi*x[1])’, degree=2)

# Define the initial condition
u0 = interpolate(u, V)

# Define the time step and final time
dt = 0.01
T = 1.0

# Define the variational problem
u = TrialFunction(V)
v = TestFunction(V)
a = u*v*dx + dt*dot(w, grad(u))*jump(v)*dS
L = u0*v*dx

# Define the boundary condition
bc = DirichletBC(V, Constant(0.0), ’on_boundary’)

# Define the file to save the solution at every time step
vtkfile = File(’solution.pvd’)

# Solve the problem
u = Function(V)
t = 0.0
while t < T:
    t += dt
    solve(a == L, u, bc)
    vtkfile << (u, t)
    u0.assign(u)
```

In this code, we first define the mesh and function space using the ‘RectangleMesh‘ and ‘FunctionSpace‘ classes, respectively. We then define the upwind flux ‘w‘ and the initial condition ‘u0‘.

Next, we define the time step ‘dt‘ and final time ‘T‘. We also define the variational problem using the ‘TrialFunction‘, ‘TestFunction‘, ‘a‘, and ‘L‘ variables. The boundary condition is defined using the DirichletBC class.

Finally, we define the file to save the solution at every time step using the ‘File‘ class and solve the problem using a loop over time steps. At each time step, we solve the variational problem using the ‘solve‘ function and save the solution to the file using the ‘vtkfile « (u, t)‘ syntax. We then update the initial condition for the next time step using ‘u0.assign(u)‘.

Unlike the codes in Section [2], one can notice that surface integrals denoted with `dS` label are present in the variational formulation and should represent the contribution due to the upwind flux. However, as evident from the description generated by the language model after the program, the upwind flux is represent by `w`, which is a constant vector with first component equal to 1 and second component equal to 0. The expression of the upwind flux $\hat{\mathbf{F}}$ is instead given by [14]

$$\hat{\mathbf{F}} = \boldsymbol{\beta}\,\{\{u\}\} + \frac{1}{2}\,|\boldsymbol{\beta}\cdot\mathbf{n}|\,[[u]]\,, \tag{3}$$

where $\{\{u\}\}$ and $[[u]]$ denote the average and jump operators, respectively, as defined in [14]. We ask therefore ChatGPT to correct itself:

> The upwind flux is not correctly implemented. Try to correct.

> You are correct, thank you for pointing that out. Here's the corrected code with the implementation of the upwind flux:

```
from fenics import *
from dolfin import *
from mshr import *

# Define the mesh and function space
mesh = RectangleMesh(Point(0, 0), Point(1, 1), 32, 32)
V = FunctionSpace(mesh, ’DG’, 1)

# Define the upwind flux
u = Expression(’sin(pi*x[0])*cos(pi*x[1])’, degree=2)
w = Expression((’u’, ’0’), u=u, degree=2)

# Define the initial condition
u0 = interpolate(u, V)

# Define the time step and final time
dt = 0.01
T = 1.0

# Define the variational problem
u = TrialFunction(V)
v = TestFunction(V)
a = u*v*dx + dt*dot(w, grad(u))*jump(v)*dS - dot(w(’+’), n(’
    +’))*u(’+’)*v(’+’)*dS + dot(w(’-’), n(’-’))*u(’-’)*v(’-’
    )*dS
L = u0*v*dx

# Define the boundary condition
bc = DirichletBC(V, Constant(0.0), ’on_boundary’)
```

```
# Define the file to save the solution at every time step
vtkfile = File('solution.pvd')

# Solve the problem
u = Function(V)
t = 0.0
while t < T:
    t += dt
    u0.assign(u)
    w.u = u0
    solve(a == L, u, bc)
    vtkfile << (u, t)
```

In this code, we define the upwind flux using an ‘Expression‘ object with a variable ‘u‘ representing the previous time step solution. We then modify the variational problem a to include the upwind flux term using the ‘('+')‘ and ‘('-')‘ notation to denote the values of ‘u‘ on either side of the cell face. We also add a term for the upwind flux boundary condition at the cell faces using the ‘n‘ variable representing the outward normal vector.

Note that we also move the ‘u0.assign(u)‘ line before the ‘w.u = u0‘ line to ensure that we are using the previous time step solution in the upwind flux calculation.

With these corrections, the code should now correctly implement the upwind flux for the 2D linear advection problem.

The definition of `w` has been modified, but it does not correspond yet to an upwind flux. Moreover, in the variational formulation, the variable `n` has not been defined. Hence, the goal of automatically generating a program which solves a linear advection problem employing the DG method and the upwind flux is not achieved. For the sake of comparison, we report a possible modification of the original code so as to fulfil the request. We consider `w` as advection velocity and we focus on the *rotating cosine hill* problem described in [18, 26], which we summarize here for the convenience of the reader. The computational domain is $\Omega = (-0.5, 0.5)^2$ and the initial condition is

$$u_0(x,y) = \begin{cases} \frac{1}{4}\left(1+\cos\left(\pi X\right)\right)\left(1+\cos\left(\pi Y\right)\right) & \text{if } X^2+Y^2 \leq 1 \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

where $X = \frac{x-\frac{1}{6}}{0.2}$ and $Y = \frac{y-\frac{1}{6}}{0.2}$. The advection velocity is $\boldsymbol{\beta} = [-y, x]^T$.

```
from fenics import *
from dolfin import *
from mshr import *
```

```
# Define the mesh and function space
mesh = RectangleMesh(Point(-0.5, -0.5), Point(0.5, 0.5), 32,
    32)
V = FunctionSpace(mesh, 'DG', 1)

# Define the advecting field and the initial condition
w = Expression(('-x[1]', 'x[0]'), degree=2) # Modify the
    advection velocity
u = Expression('0.25*(1.0 + cos(pi*(x[0] - 1.0/6.0)/0.2))
    *(1.0 + cos(pi*(x[1] - 1.0/6.0)/0.2))*\
                (((x[0] - 1.0/6.0)*(x[0] - 1.0/6.0)
    /(0.2*0.2) + (x[1] - 1.0/6.0)*(x[1] - 1.0/6.0)/(0.2*0.2)
    ) <= 1.0)',\
                degree=2)  # Modify the initial condition

# Define the initial condition
u0 = interpolate(u, V)

# Define the time step and final time
dt = 0.01
T = 1.0

# Define the variational problem
u = TrialFunction(V)
v = TestFunction(V)
n = FacetNormal(mesh) # Added definition of outward unit
    normal vector
# Modify the variational formulation
a = u*v*dx - dt*u*dot(w, grad(v))*dx \
  + dt*avg(u)*dot(w, jump(v, n))*dS \
  + dt*0.5*abs(dot(w, n('+')))*dot(jump(u,n), jump(v,n))*dS
L = u0*v*dx

# Define the boundary condition
bc = DirichletBC(V, Constant(0.0), 'on_boundary')

# Define the file to save the solution at every time step
vtkfile = File('solution.pvd')

# Solve the problem
u = Function(V)
t = 0.0
vtkfile << (u0, t) # Save also initial condition
while t < T:
    t += dt
    solve(a == L, u, bc)
    u0.assign(u)
    vtkfile << (u0, t)
```

The core of the program can be preserved and the main modification regards the variational form a, so as to include the upwind flux. Figure 5 shows the contour plots of the advected filed at $t = 0\,\mathrm{s}$ and $t = 1\,\mathrm{s}$. One can easily notice that the body is rotating as expected.

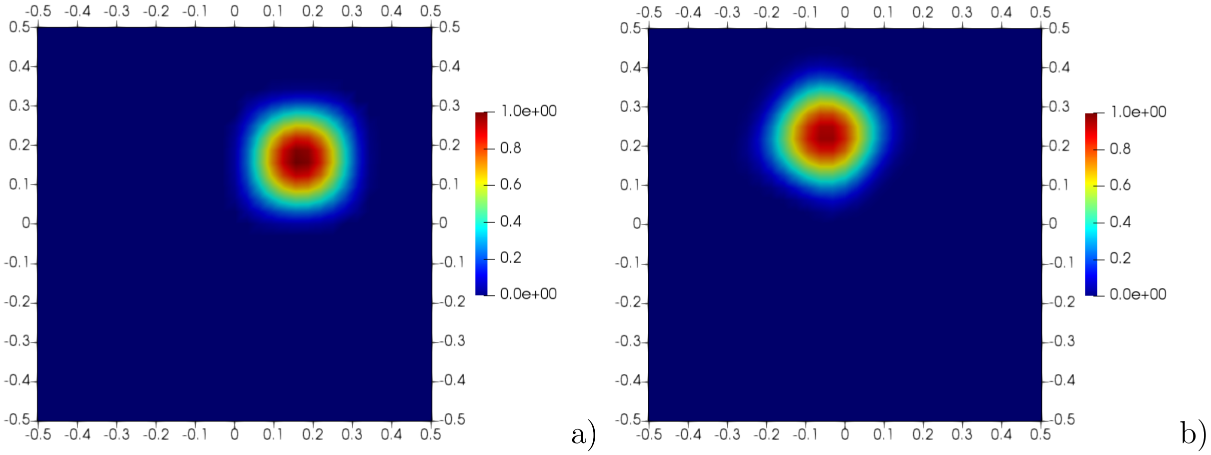


Figure 5: *Rotating cosine hill* problem. The results are obtained with the corrected code implemented in FEniCS, a) initial condition, b) results at $t = 1\,\mathrm{s}$.

4 Conclusions

We have investigated the use of ChatGPT to generate finite element codes in the frameworks of *deal.II* and FEniCS. The outcomes suggest that ChatGPT can be employed as initial building block to implement the required algorithms. However, a human supervision is still necessary to overcome some limitations and failures, such as inclusion of missing libraries or wrong interpretation of some requests. The language model is efficient for short and simple codes, whereas a major challenge regards the generation of long codes and complex algorithms, which are typically required for relevant numerical simulations. In this case, the knowledge of the employed framework and the human contribution remain essential, and a significant improvement of the machine learning model is required to achieve the goal of an automatic development of complex scientific computing codes. In future work, we aim to investigate the application of the AI-based model to DSLs.

References

[1] `https://dealii.org/`.

[2] `https://fenicsproject.org/`.

[3] `https://freefem.org/`.

[4] `https://comsol.com/`.

[5] `https://chat.openai.com/chat`.

[6] `https://dealii.org/current/doxygen/deal.II/step_3.html`.

[7] `https://petsc.org/`.

[8] https://trilinos.github.io/.

[9] https://dealii.org/current/doxygen/deal.II/step_40.html.

[10] https://dealii.org/current/doxygen/deal.II/step_12.html.

[11] D. Arndt, W. Bangerth, M. Feder, M. Fehling, R. Gassmöller, T. Heister, L. Heltai, M. Kronbichler, M. Maier, P. Munch, J.-P. Pelteret, S. Sticko, B. Turcksin, and D. Wells. The deal II library, version 9.4. *Journal of Numerical Mathematics*, 30:231–246, 2022.

[12] W. Bangerth, R. Hartmann, and G. Kanschat. deal II: a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software (TOMS)*, 33:24–51, 2007.

[13] J.M. Cimbala and Y.A. Cengel. *Fluid mechanics: fundamentals and applications.* McGraw-Hill Higher Education, 2006.

[14] B Cockburn. Discontinuous Galerkin methods. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift für Angewandte Mathematik und Mechanik: Applied Mathematics and Mechanics*, 83(11):731–754, 2003.

[15] B. Cockburn, S.Y. Lin, and C.W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. III. One-dimensional systems. *Journal of Computational Physics*, 84:90–113, 1989.

[16] B. Cockburn and C.W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws. II. General framework. *Mathematics of Computation*, 52:411–435, 1989.

[17] M. Corti, A. Zingaro, L. Dedé, and A.M. Quarteroni. Impact of atrial fibrillation on left atrium haemodynamics: A computational fluid dynamics study. *Computers in Biology and Medicine*, 150:106143, 2022.

[18] J. Donea and A. Huerta. *Finite element methods for flow problems.* John Wiley & Sons, 2003.

[19] M. Fowler. *Domain-specific languages.* Pearson Education, 2010.

[20] S. Frieder, L. Pinchetti, R.R. Griffiths, T. Salvatori, T. Lukasiewicz, P.C. Petersen, A. Chevalier, and J. Berner. Mathematical capabilities of ChatGPT. Technical Report 2301.13867, arXiv, 2023.

[21] F.X. Giraldo. *An Introduction to Element-Based Galerkin Methods on Tensor-Product Bases.* Springer Nature, 2020.

[22] T.J.R. Hughes. *The finite element method: linear static and dynamic finite element analysis.* Courier Corporation, 2012.

[23] G. Karniadakis, G.E. Karniadakis, and R.M. Kirby II. *Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation*, volume 2. Cambridge University Press, 2003.

[24] A. Kashefi and T. Mukerji. ChatGPT for Programming Numerical Methods. Technical Report 2303.12093, arXiv, 2023.

[25] M. Kronbichler, T. Heister, and W. Bangerth. High Accuracy Mantle Convection Simulation through Modern Numerical Methods. *Geophysical Journal International*, 191:12–29, 2012.

[26] R. Kumar and B.H. Dennis. Bubble-enriched least-squares finite element method for transient advective transport. *Differential Equations and Nonlinear Mechanics*, 2008, 2008.

[27] A. Logg and G.N. Wells. DOLFIN: automated finite element computing. *ACM Transactions on Mathematical Software*, 37, 2010.

[28] A. Logg, G.N. Wells, and J. Hake. DOLFIN: a C++/Python finite element library. In K.-A. Mardal A. Logg and G. N. Wells, editors, *Automated Solution of Differential Equations by the Finite Element Method*, volume 84 of *Lecture Notes in Computational Science and Engineering*, chapter 10. Springer, 2012.

[29] G. Orlando. A filtering monotonization approach for DG discretizations of hyperbolic problems. *Computers & Mathematics with Applications*, 129:113–125, 2023.

[30] G. Orlando, P.F. Barbante, and L. Bonaventura. An efficient IMEX-DG solver for the compressible Navier-Stokes equations for non-ideal gases. *Journal of Computational Physics*, 471:111653, 2022.

[31] G. Orlando, A. Della Rocca, P.F. Barbante, L. Bonaventura, and N. Parolini. An efficient and accurate implicit DG solver for the incompressible Navier-Stokes equations. *International Journal for Numerical Methods in Fluids*, 94:1484–1516, 2022.

[32] A. Quarteroni and A. Valli. *Numerical approximation of partial differential equations*, volume 23. Springer Science & Business Media, 2008.

[33] S. Salsa. *Partial differential equations in action: from modelling to theory*, volume 99. Springer, 2016.

[34] K. Szenes, N. Discacciati, L. Bonaventura, and W. Sawyer. Domain-specific implementation of high order discontinuous galerkin methods in spherical geometry. Technical Report 2303.11767, arXiv.

[35] E.B. Tadmor, R.E. Miller, and R.S. Elliott. *Continuum mechanics and thermodynamics: from fundamental concepts to governing equations.* Cambridge University Press, 2012.

[36] O.C. Zienkiewicz, R.L. Taylor, and J.Z. Zhu. *The finite element method: its basis and fundamentals.* Elsevier, 2005.

MOX Technical Reports, last issues

Dipartimento di Matematica
Politecnico di Milano, Via Bonardi 9 - 20133 Milano (Italy)

Gambarini, M.; Ciaramella, G.; Miglio, E.; Vanzan, T.
Robust optimization of control parameters for WEC arrays using stochastic methods

30/2023 Antonietti, P. F.; Bonizzoni, F.; Verani, M.
A DG-VEM method for the dissipative wave equation

28/2023 Zingaro, A.; Vergara, C.; Dede', L.; Regazzoni, F.; Quarteroni, A.
A comprehensive mathematical model for myocardial perfusion

29/2023 Carbonaro, D.; Mezzadri, F.; Ferro, N.; De Nisco, G.; Audenino, A.L.; Gallo, D.; Chiastra, C.; Morbiducci, U.; Perotto, S.
Design of innovative self-expandable femoral stents using inverse homogenization topology optimization

Fumagalli, A.; Panzeri, L.; Formaggia, L.; Scotti, A.; Arosio, D.
A mixed-dimensional model for direct current simulations in presence of a thin high-resistivity liner

27/2023 Beirao da Vega, L.; Canuto, C.; Nochetto, R.H.; Vacca, G.; Verani, M.
Adaptive VEM for variable data: convergence and optimality

26/2023 Artoni, A.; Antonietti, P.F.; Corradi, R.; Mazzieri, I.; Parolini, N.; Rocchi, D.; Schito P.; Semeraro, F.F.;
AeroSPEED: a high order acoustic solver for aeroacoustic applications

25/2023 Bonetti, S.; Botti, M.; Mazzieri, I.; Antonietti, P.F.
Numerical modelling of wave propagation phenomena in thermo-poroelastic media via discontinuous Galerkin methods

24/2023 Costa, G.; Cavinato, L.; Fiz, F.; Sollini, M.; Chiti, A.; Torzilli, G.; Ieva, F.; Viganò, L.
Mapping Tumor Heterogeneity via Local Entropy Assessment: Making Biomarkers Visible

23/2023 Bertoletti, A.; Cannistrà, M.; Diaz Lema, M.; Masci, C.; Mergoni, A.; Rossi, L.; Soncin, M.
The Determinants of Mathematics Achievement: A Gender Perspective Using Multilevel Random Forest