Generalized Multiscale Finite Element Method (GMsFEM) in Deal.II

Wenyuan Li*

May 10, 2023

Abstract

In the multiscale finite element method, we utilize two sets of refinements. The first refinement is applied to the global domain, resulting in a relatively coarse grid. Inside each coarse element, we perform an additional refinement to obtain the fine grid, which has a higher number of degrees of freedom than the coarse grid. The main objective of the multiscale finite element method is to construct coarse grid basis functions that can achieve the same accuracy as the fine grid basis functions. Various methods of constructing these basis functions have been thoroughly studied in the literature [4]. In this documentation, we will focus on the Generalized Multiscale Finite Element Method (GMsFEM) [3], implemented using the Deal.II library [2]. Additionally, this documentation serves as the final report for the MATH 676 course at Texas A&M University.

1 Multiscale Finite Element Methods

Our objective is to solve the partial differential equation:

$$\mathcal{L}(u) = f \quad \text{in} \quad [0,1]^d. \tag{1}$$

The refinement from the global domain to the coarse level is known as the global refinement times (g), while the refinement from the coarse grid to the fine grid is called the local refinement times (l). Under the refinement setting of deal.ii we have the following data for meshes.

	mesh size	dofs
fine	$h = 1/2^{g+l}$	$\mathcal{O}(2^{(g+l)d})$
coarse	$H = 1/2^{g}$	$\mathcal{O}(2^{gd})$

Our goal is to construct the coarse grid basis functions which can obtain the same accuracy as the fine grid basis functions when solving (1). As the degree of freedom is greatly reduced in the coarse grid, computation saving is provided. Figure 1 shows examples for a coarse grid, a fine grid, and a coarse basis function.

*Department of Mathematics, Texas A&M University, College Station, TX 77843, USA

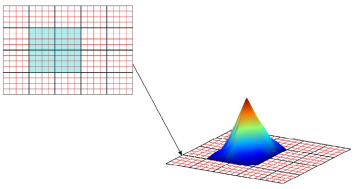


Figure 1: A first example

2 GMsFEM

In the coarse grid, every coarse node v_i has a coarse block ω_i which consists of the 4 coarse element which has v_i as one of their vertices. For every coarse block ω_i, we solve a local cell problem.

2.1 Local cell problem

We first define the following two bilinear inner products

$$a_{\omega_i}(v,w) = \int_{\omega_i} \kappa \nabla v \cdot \nabla w, \; s_{\omega_i}(v,w) = \int_{\omega_i} \widetilde{\kappa} v w,$$

where $\tilde{\kappa}$ can choose to be κ/H^2 or just κ in GMsFEM. At this step, we have two option, we can solve the following spectral problem in the fine grid basis function space (within ω_i), i.e. we find λ and v such that

$$a_{\omega_i}(v,w) = \lambda s_{\omega_i}(v,w), \quad \forall w \in V_H^{\omega_i}.$$

Or we can construct the snapshot basis $\psi_l^{\omega_i}$ by solving

$$-\nabla \cdot (\kappa(x) \nabla \psi_l^{\omega_i}) = 0, \text{ in } \omega_i; \quad \psi_l^{\omega_i} = \delta_l^h \text{ in } \partial\omega_i.$$

The characteristic funciton δ_l^h is 1 at the fine node v_l and is 0 at every other fine nodes on the boundary. Then in the snapshot space, we solve the spectral problem

$$a_{\omega_i}(v,w) = \lambda s_{\omega_i}(v,w), \quad \forall w \in V_{H,\text{ snap}}^{\omega_i}.$$

After that, we need to convert the vector v back to the fine grid coordinate system using the snapshot basis. The basic idea of snapshot basis is to find out the more important basis and use them to solve the spectral problem.

After solving the spectral problem, we rearrange the eigenvalues and select the eigenvectors corresponding to the smallest eigenvalues. We can choose J eigenvectors from one ω_i. Then we multiply the basis functions with a partition of unity and get the coarse basis functions.

2.2 Back to global domain

As the dofs in the local cell and the global domain has two totally different set of numbering, we need to construct a map and map the basis in local cells back to the global domain. Then we have the coarse space $V_H = \{\psi_j^{\omega_i} : 0 \le j \le J, \; \omega_i \subset \Omega\}$. After this mapping, we assemble every coarse basis function to a matrix R. Every column of R is a basis function and every entry of

R stands for their coordinates. Then we compute the coarse matrices in the following way. We have

$$
\begin{aligned}
Acoarse_{ij} &= a(\psi_j, \psi_i) = a(\sum_{k=1}^{N} \psi_{jk}\phi_k, \sum_{m=1}^{N} \psi_{im}\phi_m) \\
&= \sum_{k=1}^{N} \psi_{jk}\Big(\sum_{m=1}^{N} \psi_{im} a(\phi_k, \phi_m)\Big) = \sum_{k=1}^{N} \psi_{jk} \sum_{m=1}^{N} R_{mi} * Afine_{mk} \\
&= \sum_{k=1}^{N} \psi_{jk} \sum_{m=1}^{N} R'_{im} * Afine_{mk} = \sum_{k=1}^{N} \psi_{jk}(R' * Afine)_{ik} \\
&= \sum_{k=1}^{N} (R' * Afine)_{ik}\psi_{jk} = \sum_{k=1}^{N} (R' * Afine)_{ik} R_{kj} = (R' * Afine * R)_{ij}
\end{aligned}
$$

Similarly, we have

$$
\begin{aligned}
Acoarse &= R' * Afine * R \\
Mcoarse &= R' * Mfine * R \\
Fcoarse &= R' * Ffine
\end{aligned}
$$

In this way, we convert everything into the coarse space and the coarse matrices are used to find the coarse solution. After u_{coarse} is found, we will use Ru_{coarse} to map it back to the fine grid so that we can compare it with the fine grid solution.

The relative L^2 error is calculated by

$$(u_{\text{fine}} - u_{\text{coarse}}) * Mfine * (u_{\text{fine}} - u_{\text{coarse}})/(u_{\text{fine}} * Mfine * u_{\text{fine}}) * 100\%.$$

The relative energy error is computed by

$$(u_{\text{fine}} - u_{\text{coarse}}) * Afine * (u_{\text{fine}} - u_{\text{coarse}})/(u_{\text{fine}} * Afine * u_{\text{fine}}) * 100\%.$$

3 Code Explanation

The codes consist of two files, one is GMsFEM.cc and local_cell_problem.h.

3.1 GMsFEM.cc

3.1.1 Save and load data

The first two functions are copied from [1].

The following code is used to save a Eigen::MatrixXd matrix. You just have to provide the fileName (should end with ".csv") and the matrix. In this way, we can avoid duplicate computation.

```
// save the basis function
void saveData(std::string fileName, Eigen::MatrixXd  matrix)
{
    //https://eigen.tuxfamily.org/dox/structEigen_1_1IOFormat.html
    const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision, Eigen::
   DontAlignCols, ", ", "\n");
```

```
6
7       std::ofstream file(fileName);
8       if (file.is_open())
9       {
10          file << matrix.format(CSVFormat);
11          file.close();
12      }
13  }
```

The following function is used to load the saved matrix.

```
1   // load the basis functions
2   Eigen::MatrixXd openData(std::string fileToOpen)
3   {
4
5       // the input is the file: "fileToOpen.csv":
6       // a,b,c
7       // d,e,f
8       // This function converts input file data into the Eigen matrix format
9
10
11
12      // the matrix entries are stored in this variable row-wise. For example if
        we have the matrix:
13      // M=[a b c
14      //    d e f]
15      // the entries are stored as matrixEntries=[a,b,c,d,e,f], that is the
        variable "matrixEntries" is a row vector
16      // later on, this vector is mapped into the Eigen matrix format
17      std::vector<double> matrixEntries;
18
19      // in this object we store the data from the matrix
20      std::ifstream matrixDataFile(fileToOpen);
21
22      // this variable is used to store the row of the matrix that contains commas
23      std::string matrixRowString;
24
25      // this variable is used to store the matrix entry;
26      std::string matrixEntry;
27
28      // this variable is used to track the number of rows
29      int matrixRowNumber = 0;
30
31
32      while (getline(matrixDataFile, matrixRowString)) // here we read a row by
        row of matrixDataFile and store every line into the string variable
        matrixRowString
33      {
34          std::stringstream matrixRowStringStream(matrixRowString); //convert
        matrixRowString that is a string to a stream variable.
35
36          while (getline(matrixRowStringStream, matrixEntry, ',')) // here we read
         pieces of the stream matrixRowStringStream until every comma, and store the
         resulting character into the matrixEntry
37          {
38              matrixEntries.push_back(stod(matrixEntry));   //here we convert the
        string to double and fill in the row vector storing all the matrix entries
39          }
40          matrixRowNumber++; //update the column numbers
41      }
42
43      // here we convet the vector variable into the matrix and return the
```

```
    resulting object,
    // note that matrixEntries.data() is the pointer to the first memory
    location at which the entries of the vector matrixEntries are stored;
    return Eigen::Map<Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic,
    Eigen::RowMajor>>(matrixEntries.data(), matrixRowNumber, matrixEntries.size
    () / matrixRowNumber);

}
```

3.1.2 Right Hand Side Function

The following codes defines a class RightHandSide which contains one Function RightHandSide. The code should be self-explanatory. In here, we have two kind of source terms

$$f = 2\pi \sin(\pi x)\sin(\pi y),$$
$$f = 10\exp[-B * ((x-0.5)^2 + (y-0.5)^2)].$$

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

template <int dim>
double RightHandSide<dim>::value(const Point<dim> & p,
                                 const unsigned int /*component*/) const
{
  // f = 2 \pi sin(\pi x) sin(\pi y)
  double return_value = 2.0 * M_PI;
  for (unsigned int i = 0; i < dim; ++i)
    return_value *= sin(M_PI * p(i));

  // double return_value;
  // return_value = 10.0 * exp( - 500.0 * ((p(0) - 0.5) * (p(0) - 0.5) + (p(1) -
    0.5) * (p(1) - 0.5)));

  return return_value;
}
```

3.1.3 Class GMsFEM

The main class GMsFEM contains two public function GMsFEM() and run(). There are five private functions. The function buildPOU() is used to generate the partition of unity which is needed in the local cell problem. We construct the coarse basis functions in build_coarse_basis(). Fine grid solution is calculated in fine_sol(). The fine grid solution is considered as the reference solution to compute the accuracy of our method. The coarse_sol() function will utilize the coarse basis calculated before to compute the coarse solution. The output_results() function just do some routine output work.

```
template <int dim>
class GMsFEM
{
public:
```

```
  GMsFEM();
  void run();

private:
  void buildPOU();
  void build_coarse_basis();
  void fine_sol();
  void coarse_sol();
  void output_results() const;
}
```

We have a central control panel which can be used to adjust some parameters. The current code only support cube_start = 0 and cube_end = 1. The loc_refine_times and global_refine_times are explained in Section 1. The total_refine_times is the number of refinement to get the fine grid starting from the global domain. If compute_snapshot_flag = 0, we will use the first option discussed in Section 2.1 and the second option when it equals 1. We have an option to decide whether you want to compute the R matrix or load it from a file. Remember to change the file names before running the code. n_of_loc_basis controls how many basis functions we use from each local cell. The max_number_of_basis_computed is used to test error with respect to different n_of_loc_basis. It is needed when loading the R matrix from a file. This variable is useless if compute_Rms = true.

```
// central control panel
  const double cube_start = 0;
  const double cube_end = 1;
  int loc_refine_times = 3;
  int global_refine_times = 3;
  int total_refine_times = loc_refine_times + global_refine_times;

  int compute_snapshot_flag = 0;   // 0: don't compute; 1: compute
  bool compute_Rms = true;   // true: compute, false: load from file
  // don't forget to change the name!
  std::string saveFileName = "Rms.csv";
  std::string loadFileName = "Rms44.csv";
  // the first number is global refine times, the second number is local refine
    times

  unsigned int max_number_of_basis_computed = 5;
  unsigned int n_of_loc_basis = 5;
// central control panel
```

There are some other private variables I want to explain a little bit. Please see the comments.

```
  // number of coarse element in one row
  int Nx = (int) pow(2, global_refine_times);
  // the length of the coarse side
  double coarse_side = (cube_end - cube_start) / Nx;

  // number of fine element in a coarse element in a row
  int nx = (int) pow(2, loc_refine_times);
  // the length of the fine side
  double fine_side = coarse_side / nx;

  // the dof of the coarse sol
  int coarse_size = (Nx - 1) * (Nx - 1) * n_of_loc_basis;
  // the dof of the fine sol
  int fine_size = (Nx * nx + 1) * (Nx * nx + 1);
```

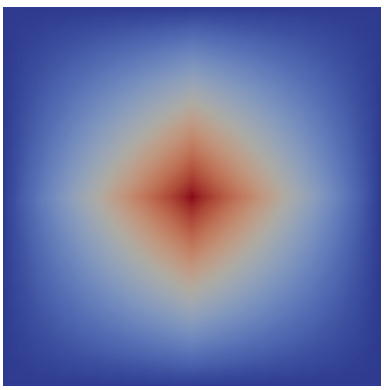
GMsFEM() The GMsFEM() function pass 1 to the finite element and indicates that we will use Q_1 basis function. Then it pass the triangulation to the dof_handler.

```
template <int dim>
GMsFEM<dim>::GMsFEM()
  : fe(1)
  , dof_handler(triangulation)
{}
```

buildPOU() The code here generates a partition of unity that is needed to construct the coarse basis function.

```
// build bilinear basis functions
template <int dim>
void GMsFEM<dim>:: buildPOU()
{
  int size1 = (int) round(pow(2, loc_refine_times + 1)) + 1;
  int size2 = (int) round(pow(2, loc_refine_times)) + 1;
  double side = 1.0 / (size2 - 1);

  POU.resize(size1, size1);
  Eigen::MatrixXd topleft(size2, size2);
  Eigen::MatrixXd topright(size2, size2);
  Eigen::MatrixXd botleft(size2, size2);
  Eigen::MatrixXd botright(size2, size2);
  for (int i = 0; i < size2; i++) {
    for (int j = 0; j < size2; j++) {
      double y = (size2 - 1 - i) * side;
      double x = j * side;
      topleft(i, j) = x * (1.0 - y);
      topright(i, j) = (1.0 - x) * (1.0 - y);
      botleft(i, j) = x * y;
      botright(i, j) = (1.0 - x) * y;
    }
  }

  // partition of unity

  POU.block(0, 0, size2, size2) = topleft;
  POU.block(0, size2, size2, size2 - 1) = topright.rightCols(size2 - 1);
  POU.block(size2, 0, size2 - 1, size2) = botleft.bottomRows(size2 - 1);
  POU.bottomRightCorner(size2 - 1, size2 - 1) = botright.bottomRightCorner(size2
     - 1, size2 - 1);

}
```

The generated function is



This function is 1 at the center and 0 on the boundary. The functions in the top-left, top-right, bottom-left, and bottom-right regions are Q_1 functions.

build_coarse_basis Please see the comments.

```
1
2  template <int dim>
3  void GMsFEM<dim>:: build_coarse_basis()
4  {
5  // initialize the size for Rms
6  Rms.resize(fine_size, coarse_size);
7
8  // decide to compute Rms or load Rms from file
9  if (compute_Rms == true) {
10 std::cout << "Start to build coarse grid basis functions. " << std::endl;
11
12 // get the interior coarse degrees of freedom
13 // and store it in coarse_centers
14 std::vector<Point<dim>> coarse_centers;
15 for (int i = 1; i < Nx; i++) {
16   for (int j = 1; j < Nx; j++) {
17
18     // coordinates of coarse nodes
19     Point<dim> coarse_center(i * coarse_side, j * coarse_side);
20     coarse_centers.push_back(coarse_center);
21   }
22 }
23
24 // define the type for convenience
25 using iterator_type = Triangulation<2>::cell_iterator;
26 using active_type   = Triangulation<2>::active_cell_iterator;
27
28 // coarse_patches stores the fine cells for each local cell problem
29 std::vector<std::vector<active_type>> coarse_patches;
30 // one coarse_center = one coarse_patch = one local cell problem
31 coarse_patches.resize(coarse_centers.size());
32
33 // collect the fine cells of each $\omega_i$ into coarse_patches
34 for (active_type cell : triangulation.active_cell_iterators()) {
35   // std::cout << "cell center: " << cell->center() << std::endl;
36   Point<dim> cell_center = cell->center();
37   for (unsigned long i = 0; i < coarse_centers.size(); i++ ) {
38     Point<dim> coarse_center = coarse_centers[i];
39     // we do the collection based on the distance
40     // between the cell_center and the coarse_center
41     if (abs(cell_center[0] - coarse_center[0]) < coarse_side &&
42       abs(cell_center[1] - coarse_center[1]) < coarse_side ) {
43         coarse_patches[i].push_back(cell);
44
45       }
46   }
47 }
48
49 // index for storing the coarse basis in the correct column
50 int Rms_i = 0;
51
52 // loop over the coarse_centers and solve the local cell problem
53 for (unsigned long i = 0; i < coarse_centers.size(); i++)
54   {
55     // get the corresponding coarse patch
56     std::vector<active_type> coarse_patch = coarse_patches[i];
```

```
    // get the coordinate of the coarse_center
    Point<dim> coarse_center = coarse_centers[i];
    // initialize the patch to global mapping
    std::map<active_type, active_type> patch_to_global_triangulation_map;
    // initialize the triangulation for patch
    Triangulation<dim> patch_triangulation;

    // use the coarse_patch to build triangulation and also construct the map
    GridTools::build_triangulation_from_patch<Triangulation<2>>(
      coarse_patch, patch_triangulation, patch_to_global_triangulation_map);


    // call the local cell problem solver with patch_triangulation
    Local<dim> local_cell_problem;
    local_cell_problem.setUp(patch_triangulation, n_of_loc_basis, POU,
    coarse_center, fine_side, coarse_side, compute_snapshot_flag);
    Eigen::MatrixXd loc_basis_return = local_cell_problem.run();
    // obtain the local coarse basis

    // change it to vector<Vector<>> so that we can use the map
    std::vector<Vector<double>> loc_basis;
    for (int j = 0; j < loc_basis_return.cols(); j++) {
      Vector<double> loc_basis_col(loc_basis_return.rows());

      for (int i = 0; i < loc_basis_return.rows(); i++) {
        loc_basis_col(i) = loc_basis_return(i, j);
      }
      loc_basis.push_back(loc_basis_col);
    }

    // for each basis from the local cell problem
    for (unsigned int i = 0; i < n_of_loc_basis; i++) {

      // initialize a global basis_function
      Vector<double> basis_function;
      basis_function.reinit(dof_handler.n_dofs());

      // initialize a temp_values vector for each cell
      Vector<double> temp_values;
      temp_values.reinit(dof_handler.get_fe().n_dofs_per_cell());

      // get the dof_handler and triangulation used in the local_cell_problem
      const auto &patch_dof_handler = local_cell_problem.get_dof_handler();
      const auto &patch_triangulation =
          local_cell_problem.get_triangulation();

      // for each pair inside the map
      for(auto pair : patch_to_global_triangulation_map) {
        // build patch_cell for each pair
        // pair.first is the cell in local cell problem
        const typename DoFHandler<dim>::cell_iterator patch_cell(
            &patch_triangulation, pair.first->level(),
            pair.first->index(), &patch_dof_handler);

        // pair.second is the corresponding cell in the global domain
        const typename DoFHandler<dim>::cell_iterator global_cell(
            &dof_handler.get_triangulation(), pair.second->level(),
            pair.second->index(), &dof_handler);

        // obtain the values from loc_basis[i] to temp_values
        patch_cell->get_dof_values(loc_basis[i], temp_values);
        // assign temp_values to the global basis_function
```

```
        global_cell->set_dof_values(temp_values, basis_function);

      }
      // in this way, we map every local basis function to a global basis function.

      // collect the basis function to Rms
      Eigen::VectorXd basis_function_temp((Nx * nx + 1) * (Nx * nx + 1));
      for (unsigned int k = 0; k < basis_function.size(); k++) {
        basis_function_temp(k) = basis_function[k];
      }
      Rms.col(Rms_i) = basis_function_temp;

      // The following code are used to check if the mapping from local to global is correct.
      // // check basis functions

      // Vector<double> basis(Rms.rows());

      // for (unsigned int j = 0; j < Rms.rows(); j++) {
      //   basis[j] = Rms(j, Rms_i);
      // }
      // // std::cout << "Rms_i = " << Rms_i << std::endl;
      // // std::cout << basis << std::endl;

      // DataOut<dim> data_out;
      //data_out.attach_dof_handler(dof_handler);
      // data_out.add_data_vector(basis, "basis");
      // data_out.build_patches();
      // std::ofstream out("basis-global.vtk");
      // data_out.write_vtk(out);
      // // check basis functions

      Rms_i += 1;


    }
    // // check basis functions
    //  if (Rms_i >= 2) {
    //     break;
    //   }
    // // check basis functions

  }

// save Rms to avoid computing it again
saveData(saveFileName, Rms);

std::cout << "Finish building coarse basis. " << std::endl;

} else {
// load Rms from a file
std::cout << "Start loading coarse basis. " << std::endl;
Eigen::MatrixXd Rms_loaded = openData(loadFileName);
std::cout << "Finish loading coarse basis. " << std::endl;


// test accuracy with different number of local basis
if (n_of_loc_basis < max_number_of_basis_computed) {
  Eigen::MatrixXd Rms_temp;
  Rms_temp.resize(fine_size, coarse_size);

```

```
  int Rms_temp_i = 0;
  for (int i = 0; i < Rms_loaded.cols(); i += max_number_of_basis_computed) {
    for (unsigned int j = 0; j < n_of_loc_basis; j++) {
      Rms_temp.col(Rms_temp_i) = Rms_loaded.col(i + j);
      Rms_temp_i++;
    }

  }

  Rms = Rms_temp;
} else if (n_of_loc_basis == max_number_of_basis_computed) {
  Rms = Rms_loaded;
} else {
  std::cout << "n_of_loc_basis > max_number_of_basis_computed !" << std::endl;
  std::cout << "can not do it with loading!" << std::endl;
  std::cout << "change compute_Rms to true !" << std::endl;
}
}
// test accuracy with different number of local basis

}
```

fine_sol() This function consists of make_fine_grid, setup_system, assemble_system, and solve. Everything is nearly the same as the functions in the dealii tutorial. So I will skip the explanation here.

coarse_sol() Please see the comments.

```
// get coarse grid matrices and solutions and calculate error
template <int dim>
void GMsFEM<dim>::coarse_sol()
{
  // convert Rms to dealii type FullMatrix Rms1
  Rms1.reinit(fine_size, coarse_size);
  for (int i = 0; i < Rms.rows(); i++) {
    for (int j = 0; j < Rms.cols(); j++) {
      Rms1(i, j) = Rms(i, j);
    }
  }

  // convert sparse matrix to full matrix
  FullMatrix<double> AfineDense;
  AfineDense.copy_from(Afine);

  std::cout <<"check point 1, this step takes time, keep waiting."<< std::endl;

  // Acoarse = R^T * Afine * R
  FullMatrix<double> Acoarse(coarse_size, coarse_size);
  FullMatrix<double> R_T_Afine(coarse_size, fine_size);
  Rms1.Tmmult(R_T_Afine, AfineDense);
  R_T_Afine.mmult(Acoarse, Rms1);
  std::cout << "check point 2" << std::endl;

  // F_coarse = R^T * F_fine;
  Vector<double> rhs_coarse(coarse_size);
  Rms1.Tvmult(rhs_coarse, rhs_fine);

  // solve sol_coarse using coarse matrices
  std::cout << "Start to solve coarse solution." << std::endl;
```

```
  Vector<double> sol_coarse_temp(coarse_size);

  SolverControl            solver_control_coarse(10000, 1e-8);
  SolverCG<Vector<double>> solver_coarse(solver_control_coarse);
  solver_coarse.solve(Acoarse, sol_coarse_temp, rhs_coarse, PreconditionIdentity
    ());

  std::cout << "   " << solver_control_coarse.last_step()
            << " CG iterations needed in coarse method to obtain convergence."
    << std::endl;

  // convert the coarse grid solution back to fine grid;
  // sol_coarse = R * sol_coarse_temp
  sol_coarse.reinit(fine_size);
  Rms1.vmult(sol_coarse, sol_coarse_temp);

  // difference = sol_fine - sol_coarse
  Vector<double> difference = sol_fine;
  difference.add(-1.0, sol_coarse);

  // get error; relative L2error =
  // difference' * Mfine * difference / (sol_fine' * Mfine * sol_fine)
  double L2error;
  double numerator;
  Vector<double> Mfine_times_difference(fine_size);
  Mfine.vmult(Mfine_times_difference, difference);
  numerator = difference * Mfine_times_difference;

  double denominator;
  Vector<double> Mfine_times_sol_fine(fine_size);
  Mfine.vmult(Mfine_times_sol_fine, sol_fine);
  denominator = sol_fine * Mfine_times_sol_fine;

  L2error = numerator / denominator;

  std::cout << "the relative L2 error is : " << L2error << std:: endl;


  // get error; relative Energy error =
  // difference' * Afine * difference / (sol_fine' * Afine * sol_fine)
  double Eerror;

  double Enumerator;
  Vector<double> Afine_times_difference(fine_size);
  Afine.vmult(Afine_times_difference, difference);
  Enumerator = difference * Afine_times_difference;

  double Edenominator;
  Vector<double> Afine_times_sol_fine(fine_size);
  Afine.vmult(Afine_times_sol_fine, sol_fine);
  Edenominator = sol_fine * Afine_times_sol_fine;

  Eerror = Enumerator / Edenominator;

  std::cout << "the relative Energy error is : " << Eerror << std:: endl;


}
```

output_results() This function outputs the plots for fine and coarse grid solutions. The procedure is quite standard.

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

  data_out.attach_dof_handler(dof_handler);

  data_out.add_data_vector(sol_fine, "sol_fine");

  data_out.build_patches();

  std::ofstream out("sol_fine.vtk");

  data_out.write_vtk(out);



  DataOut<dim> data_out1;

  data_out1.attach_dof_handler(dof_handler);

  data_out1.add_data_vector(sol_coarse, "sol_coarse");

  data_out1.build_patches();

  std::ofstream out1("sol_coarse.vtk");

  data_out1.write_vtk(out1);


}
```

run() The run() function runs the above functions in order. Remember that we should run fine_sol() before build_coarse_basis() as we need the refined triangulation.

```
template <int dim>
void GMsFEM<dim>::run()
{
  std::cout << "Solving problem in " << dim << " space dimensions."
            << std::endl;

  buildPOU();
  fine_sol();
  build_coarse_basis();
  coarse_sol();
  output_results();
}
```

main() The main function creates a GMsFEM object named GMsFEM_2d and call the run function.

```
int main()
{
  {
    GMsFEM<2> GMsFEM_2d;
    GMsFEM_2d.run();
```

```
  }

  return 0;
}
```

3.2 local_cell_problem.h

3.2.1 BoundaryValues

In this code, we use the Dirichlet boundary condition. The GMsFEM also works for other boundary conditions.

3.2.2 kappa()

This function defines the κ that we used in the equation

$$-\nabla \cdot (\kappa(x)\nabla u) = f.$$

The plot for the kappa that we use is shown in the numerical results section.

3.2.3 class Local

```

template <int dim>
class Local
{
public:
  Local();
  Eigen::MatrixXd run();

  // accept the variables from the GMsFEM class
  // and assign them to the variables inside this class
  void setUp(Triangulation<dim> &input_triangulation, unsigned int
    input_n_of_loc_basis,
              Eigen::MatrixXd input_POU, Point<dim> input_coarse_center,
              double input_fine_side, double input_coarse_side, int
    input_compute_snapshot_flag)
    {
      triangulation.copy_triangulation(input_triangulation);
      n_of_loc_basis = input_n_of_loc_basis;
      POU = input_POU;
      coarse_center = input_coarse_center;
      fine_side = input_fine_side;
      coarse_side = input_coarse_side;
      compute_snapshot_flag = input_compute_snapshot_flag;
    }

  // return the triangulation and dof_handler used in the local cell
  // they are needed for the mapping between local and global
  const Triangulation<dim> &get_triangulation() const { return triangulation; }
  const DoFHandler<dim> &get_dof_handler() const { return dof_handler; }

private:
  void make_grid();
  void setup_system();
  void assemble_system();
```

```
  void solve();
  void output_results() const;

  // definition of needed variables
  Triangulation<dim> triangulation;  // consider &
  FE_Q<dim>          fe;
  DoFHandler<dim>    dof_handler;

  SparsityPattern      sparsity_pattern;
  SparseMatrix<double> Alocal;
  SparseMatrix<double> Slocal;

  Vector<double> system_rhs;

  Eigen::MatrixXd POU;
  Point<dim> coarse_center;
  double fine_side;
  double coarse_side;
  int compute_snapshot_flag;
  unsigned int n_of_loc_basis;
  Eigen::MatrixXd loc_basis;
};
```

The functions Local(), make_grid(), setup_system(), assemble_system() are standard and similar to the functions in the dealii tutorial. The only thing I want to mention is that

$$a_{\omega_i}(v,w) = \int_{\omega_i} \kappa \nabla v \cdot \nabla w, \; s_{\omega_i}(v,w) = \int_{\omega_i} \tilde{\kappa} v w.$$

The corresponding codes are

```
cell_matrixA(i, j) +=
    (current_coefficient *               // kappa(x_q)
     fe_values.shape_grad(i, q_index) * // grad phi_i(x_q)
     fe_values.shape_grad(j, q_index) * // grad phi_j(x_q)
     fe_values.JxW(q_index));           // dx

cell_matrixS(i, j) +=
    (current_coefficient *                // kappa(x_q)
     fe_values.shape_value(i, q_index) * // phi_i(x_q)
     fe_values.shape_value(j, q_index) * // phi_j(x_q)
     fe_values.JxW(q_index));            // dx
```

solve() This function is used to obtain the local basis functions.

```

template <int dim>
void Local<dim>::solve()
{

  // this is a map with key-value = boundary_index-boundary_value
  std::map<types::global_dof_index, double> boundary_values;

  // create a temporary object to avoid changing Alocal
  SparseMatrix<double> Alocaltemp;
  Alocaltemp.reinit(sparsity_pattern);
  Alocaltemp.copy_from(Alocal);

  // convert the dealii type matrix Alocal, Slocal to Eigen::MatrixXd type
  FullMatrix<double> AlocalDense(Alocal.m(), Alocal.n());
  AlocalDense.copy_from(Alocal);
```

```
  FullMatrix<double> SlocalDense(Slocal.m(), Slocal.n());
  SlocalDense.copy_from(Slocal);
  Eigen::MatrixXd Alocal0(Alocal.m(), Alocal.n());
  Eigen::MatrixXd Slocal0(Alocal.m(), Alocal.n());
  for (unsigned long i = 0; i < AlocalDense.m(); i++) {
    for (unsigned long j = 0; j < AlocalDense.n(); j++) {

      Alocal0(i, j) = AlocalDense[i][j];
      Slocal0(i, j) = SlocalDense[i][j];
    }
  }

  // this is for the \tilde{\kappa} in the s(\cdot,\cdot)
  // but actually this does not affect the spectral problem at all
  Slocal0 = Slocal0 / coarse_side / coarse_side;


  Eigen::MatrixXd loc_basis0;

  // compute snapshot basis and solve the spectral problem in the snapshot space
  if (compute_snapshot_flag == 1) {
    // this is needed to know the index of the boundary nodes
    // we can also use Functions::ZeroFunction<2>(), for BoundaryValues<dim>()
    VectorTools::interpolate_boundary_values(dof_handler,
                                             0,
                                             BoundaryValues<dim>(),
                                             boundary_values);


    // define the coordinate matrix for the snapshot bases
    Eigen::MatrixXd Rsnap(Alocal.m(), boundary_values.size());
    int j = 0;    // column index for Rsnap

    // define the \delta_l^h (see Section 2)
    for (auto keyValuePair = boundary_values.begin(); keyValuePair !=
   boundary_values.end(); keyValuePair++) {
      keyValuePair->second = 1.0;
      for (auto otherPair = boundary_values.begin(); otherPair !=
   boundary_values.end(); otherPair++) {
        if (otherPair->first == keyValuePair->first) continue;
        otherPair->second = 0.0;
      }

      // for each \delta_l^h, we solve Au = 0 and obtain the snapshot basis
      // this is actually expensive, we can find A(interior, interior)^{-1}
      // and use it for different boundary
      // not sure how to do this in dealii, will figure it out.
      Vector<double> solution;
      solution.reinit(dof_handler.n_dofs());

      MatrixTools::apply_boundary_values(boundary_values,
                                         Alocaltemp,
                                         solution,
                                         system_rhs, false);

      SolverControl            solver_control(2000, 1e-5);
      SolverCG<Vector<double>> solver(solver_control);

      solver.solve(Alocaltemp, solution, system_rhs, PreconditionIdentity());

      for (auto i = 0; i < Rsnap.rows(); i++) {
        Rsnap(i,j) = solution[i];
```

```
      }
      j++;

    }



    // Asnap = R^T * Alocal0 * R (see Section 2)
    Eigen::MatrixXd Asnap = Rsnap.transpose() * Alocal0 * Rsnap;
    Eigen::MatrixXd Ssnap = Rsnap.transpose() * Slocal0 * Rsnap;


    // to ensure the matrices are symmetric
    // they are symmetric originally, only some machine error difference
    Asnap = (Asnap + Asnap.transpose()) / 2;
    Ssnap = (Ssnap + Ssnap.transpose()) / 2;

    // we use this eigen solver in Eigen
    // it solves Av = \lambda S v
    // it will arrange the eigenvalue to be increasing
    Eigen::GeneralizedSelfAdjointEigenSolver<Eigen::MatrixXd> ges;

    ges.compute(Asnap, Ssnap);

    // collect the first n_of_loc_basis eigenvectors
    // and use Rsnap to map it back to the fine grid
    loc_basis0 = Rsnap * ges.eigenvectors().leftCols(n_of_loc_basis);



  } else {

    // if we don't compute snapshot,
    // we can solve the spectral problem directly in the fine grid.
    Alocal0 = (Alocal0 + Alocal0.transpose()) / 2;
    Slocal0 = (Slocal0 + Slocal0.transpose()) / 2;
    Eigen::GeneralizedSelfAdjointEigenSolver<Eigen::MatrixXd> ges;
    ges.compute(Alocal0, Slocal0);

    loc_basis0 = ges.eigenvectors().leftCols(n_of_loc_basis);


  }



  // these codes are used to find the coordinates of the dofs
  MappingQ<dim> mapping(1);
  std::map<types::global_dof_index, Point<dim>> support_points;
  auto fe_collection = dof_handler.get_fe_collection();

  DoFTools::map_dofs_to_support_points(mapping, dof_handler, support_points);

  // we need to multiply the eigenvectors with the partition of unity
  // we use the coordinates of the coarse_center and the dof to locate the
    position of the dof.
  // And find the correct POU value to multiply
  Eigen::VectorXd POUvector(support_points.size());

  // the dof may locates in the left or bottom of the coarse_center
  // the index in POU are all positive
  // we need to shift the position
```

```
  int move_position = POU.cols() / 2;

  for (auto support_point : support_points) {
    Point<dim> coordinates = support_point.second;
    int positionx = (int) round((coordinates(0) - coarse_center[0]) / fine_side)
     + move_position;
    int positiony = (int) round((coordinates(1) - coarse_center[1]) / fine_side)
     + move_position;
    POUvector(support_point.first) = POU(POU.rows() - 1 - positiony, positionx);

  }

  // multiply the eigenvectors with the partition of unity
  loc_basis = loc_basis0.array().colwise() * POUvector.array();
}
```

output_results() This function in this local cell problem is used to check if the local basis functions are correct.

run() This function sets up the order of running every function and returns the local basis functions to the GMsFEM class.

4 Numerical Results

First we want to verify that our mapping from local cell to the global domain is correct. Figure 2 shows a basis function in the global domain and the local cell. From these two plots, we find that the function restricted to cell of the global domain exactly match the basis in the local cell in the right plot. These two plots show that our mapping is correct.

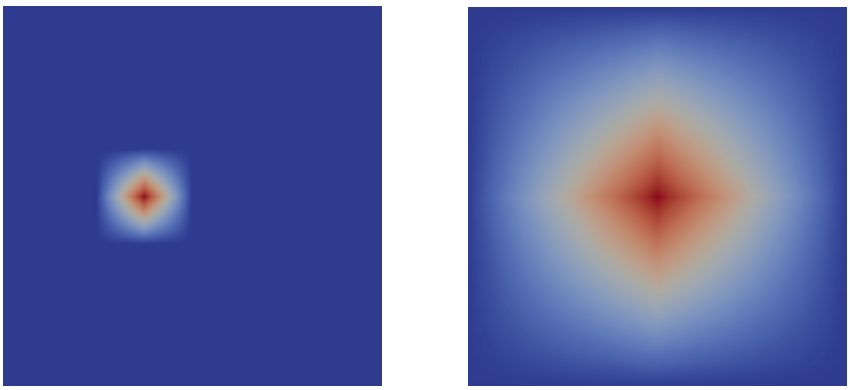


Figure 2: Global ⇔ Local

We will compare the coarse and fine solution and compute the relative error. The left plot in Figure 3 presents the $\kappa(x)$ that we use. $\kappa(x)$ contains many high contrast regions. The κ value in the yellow regions is 10000 while is 1 in the blue areas. The right plot shows the source term f. The exact formula for f is

$$f = 2\pi \sin(\pi x) \sin(\pi y).$$

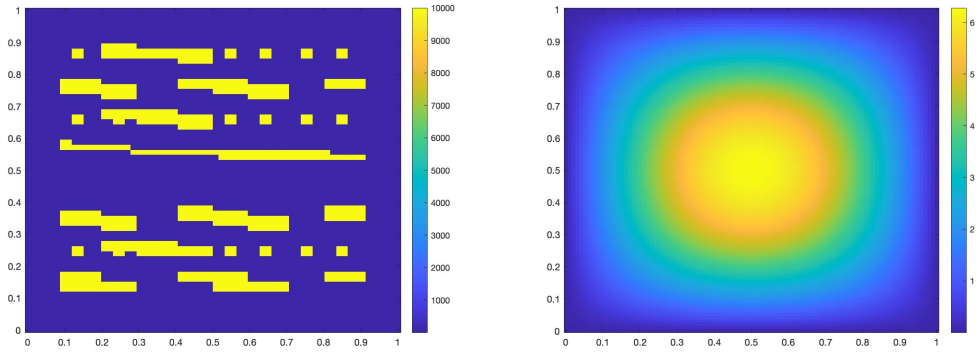


Figure 3: Left: κ. Right: source term f.

In our first example, the option compute_snapshot_flag is set to be false. Both the global_refine_time and local_refine_time is 4. We show some basis functions to illustrate our algorithm. We pick a local cell randomly. The eigenvector corresponding to the smallest eigenvalue for every spectral problem is a constant vector with every entry being the same. So after multiplying the eigenvector with the partition of unity, the final basis function is a scale of the partition of unity.

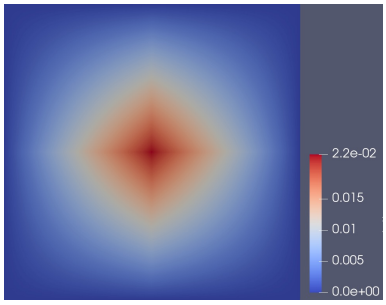


Figure 4: Basis function corresponding to the smallest eigenvalue.

The following four plots show the basis functions corresponding to the second, third, fourth, and fifth eigenvalue. We can see that in the high-value regions, the basis functions tend to be flat. The reason behind this phenomena is that the flow move much faster in these regions.

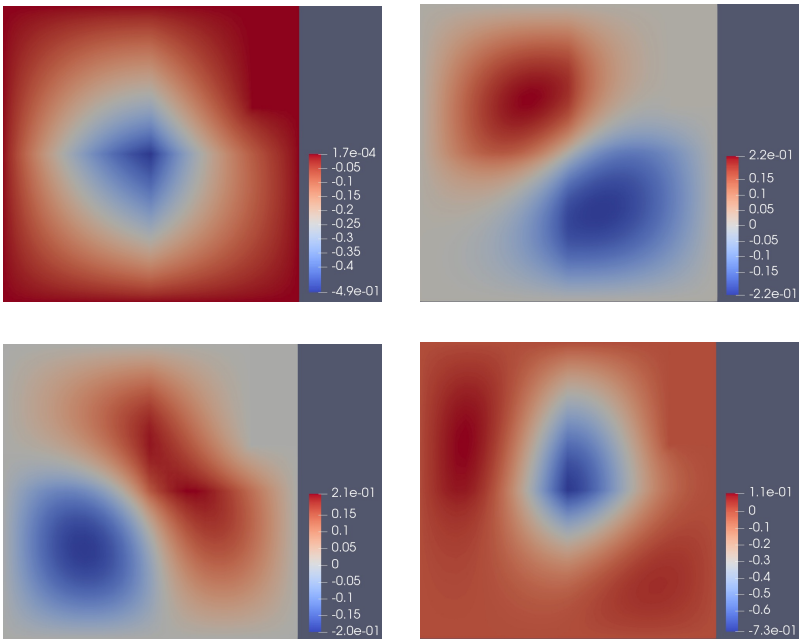


Figure 5: Basis functions.

The solution obtained using the fine grid basis functions and the coarse grid basis functions are presented in Figure 6. The n_of_loc_basis is set to be 5. These two plots resemble each other. To be more precise, we compute the errors. The relative L^2 error equals 0.0154% and the relative energy error is 1.151%. Such small errors indicate that our GMsFEM can obtain the same accuracy as the fine grid basis functions.

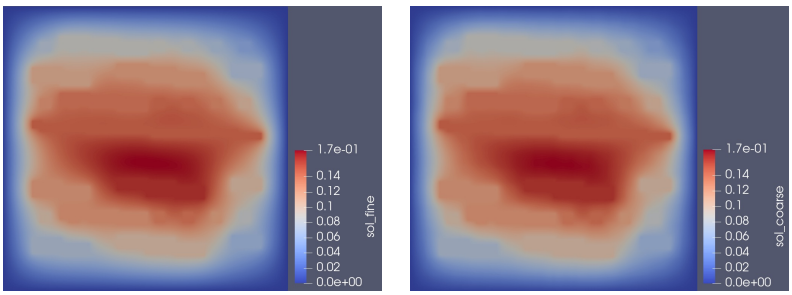


Figure 6: Left: fine solution. Right: coarse solution.

We want to test our method using different number of local basis functions. The error data is shown in the table below. We can see that the error is quite large when $n = 1$. As we add more basis from each local cell, the error drops down.

n	L^2 error	Energy error
1	27.23%	51.58%
2	3.66%	17.61%
3	0.26%	4.92%
4	0.11%	3.19%
5	0.0154%	1.151%

We want to test our method with respect to different source terms. The source term we use is

$$f = 10 * exp[-B * ((x - 0.5)^2 + (y - 0.5)^2)].$$

The function with $B = 1$ and $B = 500$ is shown in the left and right plot in Figure 7, respectively. The function becomes more and more singular as B increases.

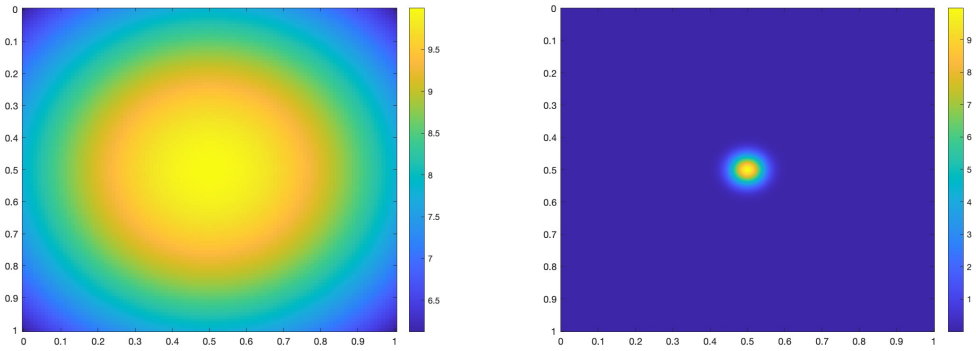


Figure 7: Left: B = 1. Right: B = 500

We have global_refine_times = 4, local_refine_times = 4 and n_of_loc_basis = 5. We show the error data below.

B	L^2 error	Energy error
1	0.0148%	1.13%
10	0.0158%	1.17%
50	0.0166%	1.18%
100	0.0161%	1.09%
200	0.0158%	1.04%
500	0.0182%	1.37%

From the table, we find that when the source term becomes more singular, the error does not change too much and our method obtain high accuracy. We show the fine and coarse solution when $B = 500$ in Figure 8. The solution profiles are similar to each other.

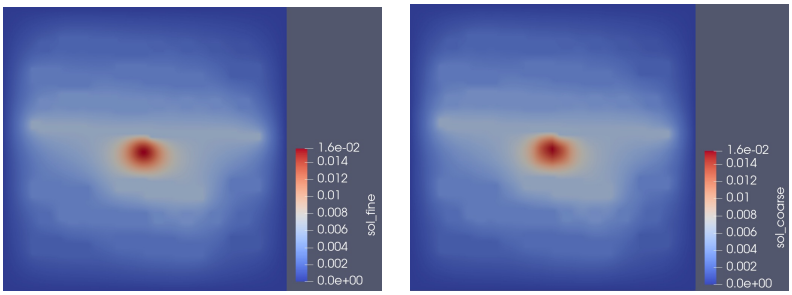


Figure 8: Left: fine solution. Right: coarse solution.

To be complete, we show a case when compute_snapshot_flag = true. We have global_refine_times = 4, local_refine_times = 3 and n_of_loc_basis = 5. The source term in Figure 3 is used. The fine and coarse solution are shown in Figure 9.

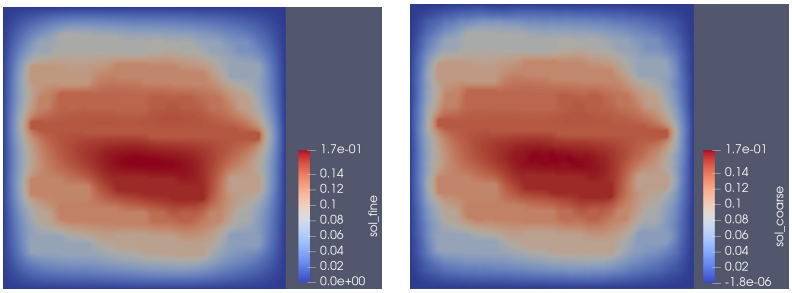


Figure 9: Left: fine solution. Right: coarse solution.

The relative L^2 error is 0.0516% and the relative energy error is 2.11%. We can see that the scheme where we solve the spectral problem in the snapshot space also works well.

5 Conclusion

We find that GMsFEM can obtain the same accuracy as the fine grid basis functions. Indeed, the preparation of basis functions takes time. But the preparation is a one-time cost. After the basis functions are constructed, the media properties are extracted and we can reuse it to solve many equations.

6 Future Work

I am working on the parallel computing for local cell problems. Once it is done, we will solve multiple local cell problems in different threads at the same time, which accelerate the computation a lot. One more possible work is to make the code compatible for 3D cases. As the dimension increases, the computation saving in our method is more significant than 2D cases.

7 Acknowledgement

The author wants to express his sincere gratitude to Dr. Matthias Maier for his hard work and patience in this course throughout the semester.

References

[1] Admin. Eigen c++ matrix library tutorial – saving and loading data in and from csv files, Jul 2020.

[2] 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(3):231–246, 2022.

[3] Y. Efendiev, J. Galvis, and T. Y. Hou. Generalized multiscale finite element methods (gmsfem). *Journal of Computational Physics*, 251:116–135, 2013.

[4] Y. Efendiev and T. Y. Hou. *Multiscale finite element methods: theory and applications*, volume 4. Springer Science & Business Media, 2009.