Math676 Final Report

Graham Harper

1 Introduction

The goal of this semester's project was to implement the Bernardi-Raugel finite elements in `deal.II` as part of the library. The Bernardi-Raugel (BR_1) finite elements are defined as an enrichment of the Q_1^d space so that the pair (BR_1,Q_0) is an LBB stable pair for the stokes problems. This is done by defining "bubble functions" for each edge in the mesh. On the unit square $[0,1]^2$, define the four bubble functions as follows:

$$\begin{array}{ll} \hat{\psi}_1(\hat{x},\hat{y}) = \hat{y}(1-\hat{y})(1-\hat{x})\mathbf{e}_1 & \hat{\psi}_2(\hat{x},\hat{y}) = \hat{x}\hat{y}(1-\hat{y})\mathbf{e}_1 \\ \hat{\psi}_3(\hat{x},\hat{y}) = \hat{x}(1-\hat{x})(1-\hat{y})\mathbf{e}_2 & \hat{\psi}_4(\hat{x},\hat{y}) = \hat{x}\hat{y}(1-\hat{x})\mathbf{e}_2 \end{array} . \tag{1}$$

To understand how these shape functions are defined on a general mesh, let $\mathcal{T}_h$ be a quadrilateralization of the given problem domain Ω satisfying some niceness assumptions. Then for each $E \in \mathcal{T}_h$, there is an invertible bilinear transformation $\Phi : [0,1]^2 \to E$. Define the values of the shape functions on E by

$$\psi_i(x,y) = \hat{\psi}_i(\Phi^{-1}(x,y)) \qquad i = 1,2,3,4 \tag{2}$$

Recall that $\Phi^{-1}(x,y)$ is also referred to as the pullback of the coordinates (x,y). Then the Bernardi-Raugel local space is defined by

$$BR_1(E) := Q_1^2(E) \oplus \text{span}\{\psi_i : i = 1,2,3,4\} \tag{3}$$

Consider the variational formulation of the mixed Laplace equation with pure Dirichlet boundary given by

$$(K^{-1}\mathbf{u},\mathbf{v})_\Omega - (\nabla\cdot\mathbf{u},q)_\Omega - (p,\nabla\cdot\mathbf{v})_\Omega = -(p_d,\mathbf{v}\cdot\mathbf{n})_{\partial\Omega} - (f,q)_\Omega. \tag{4}$$

where the Dirichlet boundary conditions given by $p|_{\partial\Omega} = p_d$ are enforced naturally. Then the finite element discretizations $\{\mathbf{u}_h, p_h\} \in BR_1 \times Q_0$ for this equation are presented in the modified step-20 code below.

2 Code with Comments

```
namespace Step20
{
  using namespace dealii;
```

```
  template <int dim>
  class MixedLaplaceProblem
  {
  public:
    MixedLaplaceProblem (const unsigned int degree);
    void run ();

  private:
    void make_grid_and_dofs ();
    void assemble_system ();
    void solve ();
    void compute_errors () const;
    void output_results () const;

    const unsigned int   degree;

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

    BlockSparsityPattern      sparsity_pattern;
    BlockSparseMatrix<double> system_matrix;
    ConstraintMatrix          constraints;

    BlockVector<double>       solution;
    BlockVector<double>       system_rhs;
};
```

This class stores all of the information for the problem. The triangulation, fe, and dof_hanlder are all used for setup, the sparsity_pattern, system_matrix, constraints, solution, and system_rhs are all used for the linear system.

```
template <int dim>
class RightHandSide : public Function<dim>
{
public:
  RightHandSide () : Function<dim>(1) {}

  virtual double value (const Point<dim>   &p,
```

```
                        const unsigned int  component = 0) const;
};

template <int dim>
class PressureBoundaryValues : public Function<dim>
{
public:
  PressureBoundaryValues () : Function<dim>(1) {}

  virtual double value (const Point<dim>   &p,
                        const unsigned int  component = 0) const;
};


template <int dim>
class ExactSolution : public Function<dim>
{
public:
  ExactSolution () : Function<dim>(dim+1) {}

  virtual void vector_value (const Point<dim> &p,
                             Vector<double>   &value) const;
};


template <int dim>
double RightHandSide<dim>::value (const Point<dim>  &p,
                                  const unsigned int /*component*/) const
{
  //2d sinsin example
    return 2*3.14159*3.14159*std::sin(3.14159*p[0])*std::sin(3.14159*p[1]);

  //3d sinsinsin example
  //return 3*3.14159*3.14159*std::sin(3.14159*p[0])*std::sin(3.14159*p[1])*std::sin(3.13159

  //other 2d examples
  //return 0;
}
```

```
template <int dim>
double PressureBoundaryValues<dim>::value (const Point<dim>  &p,
                                           const unsigned int /*component*/) const
{
  const double alpha = 0.3;
  const double beta = 1;

  //original example
  //return -(alpha*p[0]*p[1]*p[1]/2 + beta*p[0] -alpha*p[0]*p[0]*p[0]/6);

  //2d and 3d sinsinsin examples
  return 0;

  //p(x,y)=-x example
  //return -p[0];
}

template <int dim>
void
ExactSolution<dim>::vector_value (const Point<dim> &p,
                                  Vector<double>   &values) const
{
  Assert (values.size() == dim+1,
          ExcDimensionMismatch (values.size(), dim+1));

  const double alpha = 0.3;
  const double beta = 1;

  //2d original example
  //values(0) = alpha*p[1]*p[1]/2 + beta - alpha*p[0]*p[0]/2;
  //values(1) = alpha*p[0]*p[1];
  //values(2) = -(alpha*p[0]*p[1]*p[1]/2 + beta*p[0] - alpha*p[0]*p[0]*p[0]/6);

  //2d sinsin example
  values(0) = -3.14159*std::cos(3.14159*p[0])*std::sin(3.14159*p[1]);
  values(1) = -3.14159*std::sin(3.14159*p[0])*std::cos(3.14159*p[1]);
  values(2) = std::sin(3.14159*p[0])*std::sin(3.14159*p[1]);

  //3d sinsinsin example
  //values(0) = -3.14159*std::cos(3.14159*p[0])*std::sin(3.14159*p[1])*std::sin(3.14159*p[2
  //values(1) = -3.14159*std::sin(3.14159*p[0])*std::cos(3.14159*p[1])*std::sin(3.14159*p[2
```

```
  //values(2) = -3.14159*std::sin(3.14159*p[0])*std::sin(3.14159*p[1])*std::cos(3.14159*p[2
  //values(3) = std::sin(3.14159*p[0])*std::sin(3.14159*p[1])*std::sin(3.14159*p[2]);

  //2d p(x,y)=-x example
  //values(0) = 1;
  //values(1) = 0;
  //values(2) = -p[0];
}
```

This just defines the boundary conditions for the problem. values(i) contains information about the exact pressure and velocity, the different test cases are commented out for now.

```
template <int dim>
class KInverse : public TensorFunction<2,dim>
{
public:
  KInverse () : TensorFunction<2,dim>() {}

  virtual void value_list (const std::vector<Point<dim> > &points,
                           std::vector<Tensor<2,dim> >    &values) const;
};

template <int dim>
void
KInverse<dim>::value_list (const std::vector<Point<dim> > &points,
                           std::vector<Tensor<2,dim> >    &values) const
{
  Assert (points.size() == values.size(),
          ExcDimensionMismatch (points.size(), values.size()));

  for (unsigned int p=0; p<points.size(); ++p)
    {
      values[p].clear ();

      for (unsigned int d=0; d<dim; ++d)
        values[p][d][d] = 1.;
    }
}
```

The permeability tensor $\mathbf{K}$ is taken to be the identity, and the corresponding inverse must be the identity.

```
template <int dim>
MixedLaplaceProblem<dim>::MixedLaplaceProblem (const unsigned int degree)
  :
  degree (degree),
  fe (FE_BernardiRaugel<dim>(degree+1), 1,
      FE_DGQ<dim>(degree), 1),
  dof_handler (triangulation)
{}
```

This merely initializes the finite element system using the given degree and finite element pair.

```
  template <int dim>
  void MixedLaplaceProblem<dim>::make_grid_and_dofs ()
  {
    GridGenerator::hyper_cube (triangulation, -1, 1);
    triangulation.refine_global (7);

    dof_handler.distribute_dofs (fe);

    DoFRenumbering::component_wise (dof_handler);
    constraints.close();

    std::vector<types::global_dof_index> dofs_per_component (dim+1);
    DoFTools::count_dofs_per_component (dof_handler, dofs_per_component);
    const unsigned int n_u = dofs_per_component[0],
                       n_p = dofs_per_component[dim];

    std::cout << "Number of active cells: "
              << triangulation.n_active_cells()
              << std::endl
              << "Total number of cells: "
              << triangulation.n_cells()
              << std::endl
              << "Number of degrees of freedom: "
              << dof_handler.n_dofs()
```

```
              << " (" << n_u << '+' << n_p << ')'
              << std::endl;

    BlockDynamicSparsityPattern dsp(2, 2);
    dsp.block(0, 0).reinit (n_u, n_u);
    dsp.block(1, 0).reinit (n_p, n_u);
    dsp.block(0, 1).reinit (n_u, n_p);
    dsp.block(1, 1).reinit (n_p, n_p);
    dsp.collect_sizes ();
    DoFTools::make_sparsity_pattern (dof_handler, dsp);

    sparsity_pattern.copy_from(dsp);
    system_matrix.reinit (sparsity_pattern);

    solution.reinit (2);
    solution.block(0).reinit (n_u);
    solution.block(1).reinit (n_p);
    solution.collect_sizes ();

    system_rhs.reinit (2);
    system_rhs.block(0).reinit (n_u);
    system_rhs.block(1).reinit (n_p);
    system_rhs.collect_sizes ();
}
```

This chunk of code sets up the global linear system and divides it into the block structure for $\mathbf{u}_h$ and p_h. Correspondingly, it does the same for the right-hand side.

```
  template <int dim>
  void MixedLaplaceProblem<dim>::assemble_system ()
  {
    QGauss<dim>   quadrature_formula(3);
    QGauss<dim-1> face_quadrature_formula(3);

    FEValues<dim> fe_values (fe, quadrature_formula,
                             update_values    | update_gradients |
                             update_quadrature_points  | update_JxW_values);
    FEFaceValues<dim> fe_face_values (fe, face_quadrature_formula,
                                      update_values    | update_normal_vectors |
                                      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();
const unsigned int   n_face_q_points = face_quadrature_formula.size();

FullMatrix<double>   local_matrix (dofs_per_cell, dofs_per_cell);
Vector<double>       local_rhs (dofs_per_cell);

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

const RightHandSide<dim>          right_hand_side;
const PressureBoundaryValues<dim> pressure_boundary_values;
const KInverse<dim>               k_inverse;

std::vector<double> rhs_values (n_q_points);
std::vector<double> boundary_values (n_face_q_points);
std::vector<Tensor<2,dim> > k_inverse_values (n_q_points);

const FEValuesExtractors::Vector velocities (0);
const FEValuesExtractors::Scalar pressure (dim);

typename DoFHandler<dim>::active_cell_iterator
cell = dof_handler.begin_active(),
endc = dof_handler.end();
for (; cell!=endc; ++cell)
  {
    fe_values.reinit (cell);
    local_matrix = 0;
    local_rhs = 0;

    right_hand_side.value_list (fe_values.get_quadrature_points(),
                                rhs_values);
    k_inverse.value_list (fe_values.get_quadrature_points(),
                          k_inverse_values);

    for (unsigned int q=0; q<n_q_points; ++q)
      for (unsigned int i=0; i<dofs_per_cell; ++i)
        {
          const Tensor<1,dim> phi_i_u     = fe_values[velocities].value (i, q);
          const double        div_phi_i_u = fe_values[velocities].divergence (i, q);
          const double        phi_i_p     = fe_values[pressure].value (i, q);
```

```
        for (unsigned int j=0; j<dofs_per_cell; ++j)
          {
            const Tensor<1,dim> phi_j_u     = fe_values[velocities].value (j, q);
            const double        div_phi_j_u = fe_values[velocities].divergence (j, q)
            const double        phi_j_p     = fe_values[pressure].value (j, q);

            local_matrix(i,j) += (phi_i_u * k_inverse_values[q] * phi_j_u
                                  - div_phi_i_u * phi_j_p
                                  - phi_i_p * div_phi_j_u)
                                 * fe_values.JxW(q);
          }

        local_rhs(i) += -phi_i_p *
                        rhs_values[q] *
                        fe_values.JxW(q);
      }

  for (unsigned int face_n=0;
       face_n<GeometryInfo<dim>::faces_per_cell;
       ++face_n)
    if (cell->at_boundary(face_n))
      {
        fe_face_values.reinit (cell, face_n);

        pressure_boundary_values
        .value_list (fe_face_values.get_quadrature_points(),
                     boundary_values);

        for (unsigned int q=0; q<n_face_q_points; ++q)
          for (unsigned int i=0; i<dofs_per_cell; ++i)
            local_rhs(i) += -(fe_face_values[velocities].value (i, q) *
                              fe_face_values.normal_vector(q) *
                              boundary_values[q] *
                              fe_face_values.JxW(q));
      }


  cell->get_dof_indices (local_dof_indices);

constraints.distribute_local_to_global (local_matrix,
```

```
                                                local_rhs,
                                                local_dof_indices,
                                                system_matrix,
                                                system_rhs);
      }
}
```

This is where the majority of work is done. The local and global matrices and vectors are computed and assembled according to the local-to-global distribution. The piece of code most unique to this problem is `(phi_i_u * k_inverse_values[q] * phi_j_u - div_phi_i_u * phi_j_p - phi_i_p * div_phi_j_u) * fe_values.JxW(q)`, since this is the left-hand side bilinear functional for this problem. It evaluates the integral for $(K^{-1}\mathbf{u}, \mathbf{v})$ and the other two terms using the quadrature.

```
  template <int dim>
  void MixedLaplaceProblem<dim>::solve ()
  {
    std::cout << "Solving linear system... ";
    Timer timer;

    SparseDirectUMFPACK  A_direct;
    A_direct.initialize(system_matrix);

    A_direct.vmult (solution, system_rhs);
    constraints.distribute (solution);

    timer.stop ();
    std::cout << "done ("
            << timer.cpu_time()
            << "s)"
            << std::endl;
}
```

The solver used for this problem is a direct solver. This was to combat any issues that may arise from using a preconditioned solver that depended on special structures of the system.

```
  template <int dim>
```

```
  void MixedLaplaceProblem<dim>::compute_errors () const
  {
    const ComponentSelectFunction<dim>
    pressure_mask (dim, dim+1);
    const ComponentSelectFunction<dim>
    velocity_mask(std::make_pair(0, dim), dim+1);

    ExactSolution<dim> exact_solution;
    Vector<double> cellwise_errors (triangulation.n_active_cells());

    QTrapez<1>     q_trapez;
    QIterated<dim> quadrature (q_trapez, 4);

    VectorTools::integrate_difference (dof_handler, solution, exact_solution,
                                       cellwise_errors, quadrature,
                                       VectorTools::L2_norm,
                                       &pressure_mask);
    const double p_l2_error = VectorTools::compute_global_error(triangulation,
                                                                cellwise_errors,
                                                                VectorTools::L2_norm);

    VectorTools::integrate_difference (dof_handler, solution, exact_solution,
                                       cellwise_errors, quadrature,
                                       VectorTools::L2_norm,
                                       &velocity_mask);
    const double u_l2_error = VectorTools::compute_global_error(triangulation,
                                                                cellwise_errors,
                                                                VectorTools::L2_norm);

    std::cout << "Errors: ||e_p||_L2 = " << p_l2_error
              << ",   ||e_u||_L2 = " << u_l2_error
              << std::endl;
}
```

This is where the L^2 errors are computed. It's important to note that these were computed in the results section using a 2nd order quadrature, even though this displays a 4th order quadrature. No noticable difference was observed between the two quadratures.

```
  template <int dim>
  void MixedLaplaceProblem<dim>::output_results () const
```

```
  {
    std::vector<std::string> solution_names(dim, "u");
    solution_names.push_back ("p");
    std::vector<DataComponentInterpretation::DataComponentInterpretation>
    interpretation (dim,
                    DataComponentInterpretation::component_is_part_of_vector);
    interpretation.push_back (DataComponentInterpretation::component_is_scalar);

    DataOut<dim> data_out;
    data_out.add_data_vector (dof_handler, solution, solution_names, interpretation);

    data_out.build_patches (degree);

    std::ofstream output ("solution.vtu");
    data_out.write_vtu (output);
}
```

The solution is then written to a vtu file for visualization in Paraview. This is the piece that generates the nice figures you see in the results section.

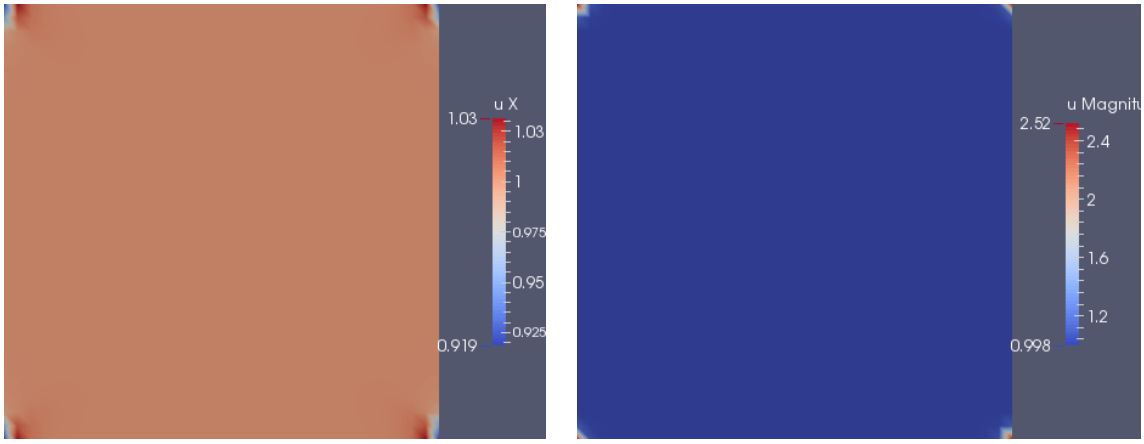
```
  template <int dim>
  void MixedLaplaceProblem<dim>::run ()
  {
    make_grid_and_dofs();
    assemble_system ();
    solve ();
    compute_errors ();
    output_results ();
  }
}

int main ()
{
  try
    {
      using namespace dealii;
      using namespace Step20;

      MixedLaplaceProblem<2> mixed_laplace_problem(0);
//      MixedLaplaceProblem<3> mixed_laplace_problem(0);
```

```
      mixed_laplace_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;
}
```

This is the meat of the code that runs everything. MixedLaplaceEquation::run() runs all of the previous code together, and run() is called by main().

3 Results

Several test cases were explored for this problem. They show the convergence of the element for the mixed Laplace equation, compare zero boundary condition to nonzero boundary condition, and show the generalization of the element to 3D.



u_h $\|u_h\|$

Figure 1: Numerical solution for example 1, $n = 2^5$

Example 1 (Nonzero boundary condition)
$\Omega = [-1, 1]^2$ with exact solution $p(x, y) = -x$.

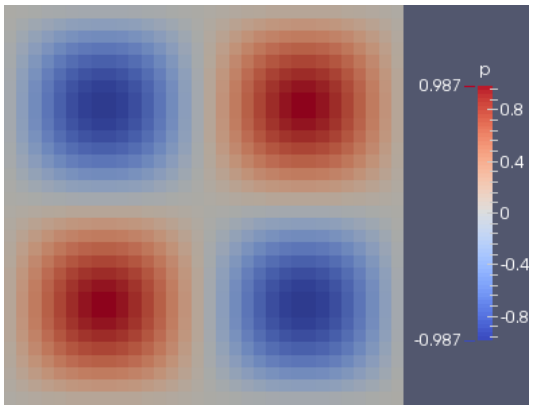
n	$\|p - p_h\|_{L^2}$	$\|u - u_h\|_{L^2}$	conv. rate (u)
2^1	7.0747E-1	1.2403E0	-
2^2	3.5383E-1	7.4202E-1	0.74
2^3	1.7682E-1	3.7579E-1	0.98
2^4	8.8394E-2	1.8780E-1	1.00
2^5	4.4195E-2	9.3953E-2	1.00

The convergence rate in L^2 norm for velocity is 1. It is easy to observe the convergence in pressure is also first-order. The singularities in the corners of the domain are worrisome, but they do not detract from the overall convergence. This motivates testing a zero boundary condition problem.

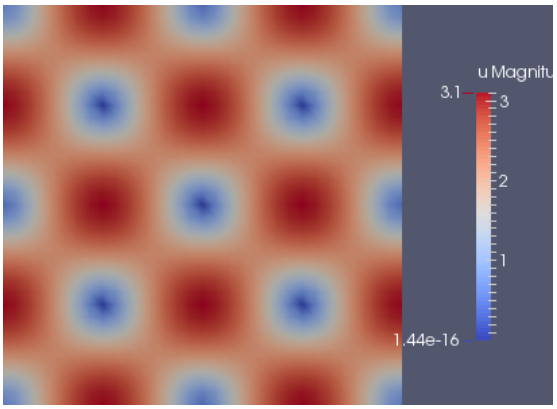
Example 2 (Zero boundary condition)
Exact solution $p(x, y) = \sin(\pi x) \sin(\pi y)$

n	$\|p - p_h\|_{L^2}$	$\|u - u_h\|_{L^2}$	conv. rate (u)
2^1	9.0167E-1	1.6681E0	-
2^2	6.8533E-1	2.1575E0	-0.37
2^3	3.7923E-1	9.5820E-1	1.17
2^4	1.9467E-1	3.5538E-1	1.43
2^5	9.7967E-2	1.2285E-1	1.53
2^6	4.9062E-2	4.2237E-2	1.54
2^7	2.4541E-2	1.4653E-2	1.53

The convergence rate in L^2 norm for velocity is somewhere near 1.5 this time. I am

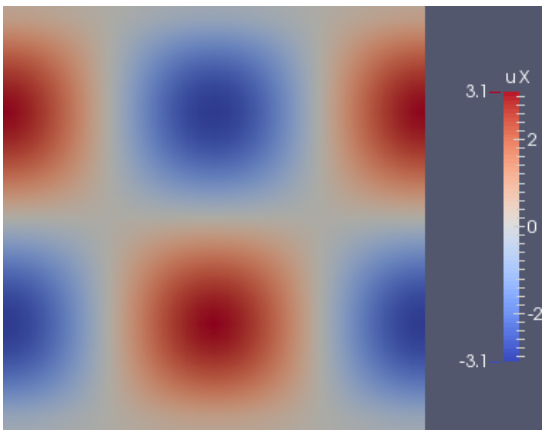


p_h

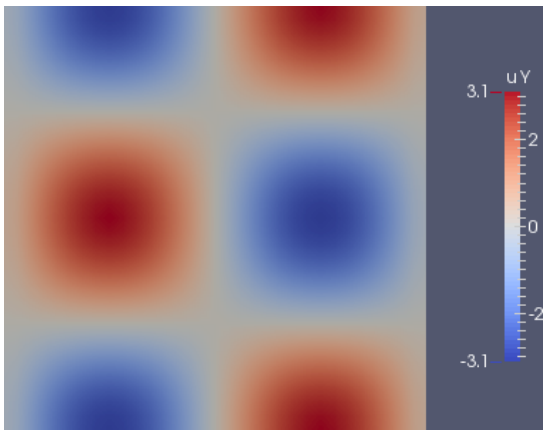


$\|u_h\|$

Figure 2: Numerical solution for example 2, $n = 2^5$



x-velocity



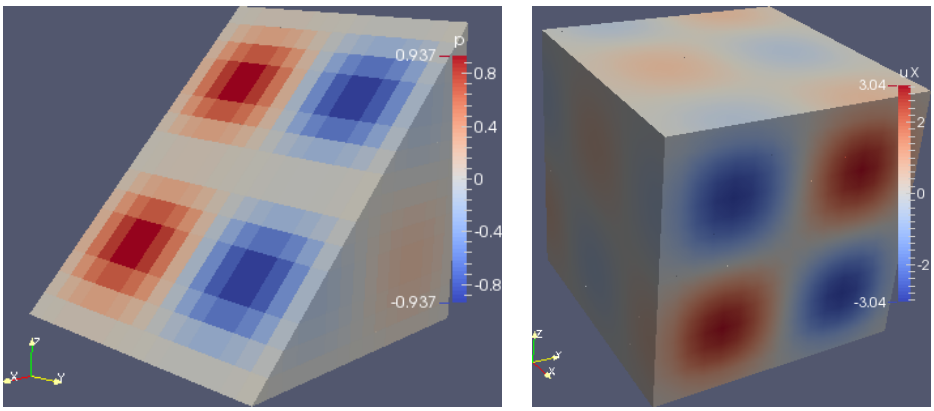
y-velocity

Figure 3: Numerical solution for example 2, $n = 2^5$

unable to come up with a better explanation than "maybe this element isn't optimal for the mixed Laplace equation" to explain this phenomenon. This could be an mildly interesting subject of future investigations, although considering how silly this element combination is, this element pair is likely not worth investigating analytically. It is again easy to observe the convergence in pressure is first-order. This numerical solution seems to lack the strange boundary behavior of the previous example, so it's worth checking this for 3D as well.

Example 3 (3D example)
Exact solution $p(x, y) = \sin(\pi x)\sin(\pi y)\sin(\pi z)$

n	$\|p - p_h\|_{L^2}$	$\|u - u_h\|_{L^2}$	conv. rate (u)
2^1	1.0251E0	2.6923E0	-
2^2	7.8726E-1	3.3393E0	-0.31
2^3	4.5899E-1	1.4300E0	1.22
2^4	2.3806E-1	5.1156E-1	1.48
2^5	-	-	-

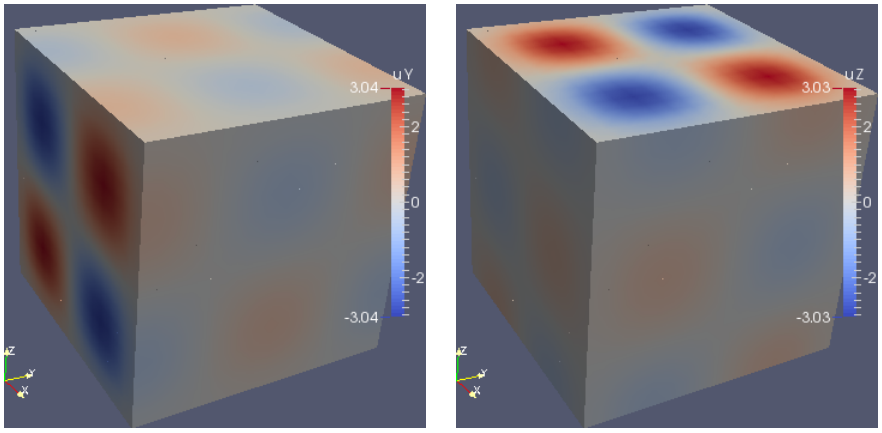


p_h x-velocity

Figure 4: Numerical solution for example 3, $n = 2^4$

The convergence rate in L^2 norm for velocity is again near 1.5. This isn't too surprising, given the previous result, but it is nice to see that the results look like they match the exact solution pretty well. The mesh size was not reduced to 2^5 divisions in each direction as it increased the degrees of freedom far beyond what the direct solver was capable of solving. The plot of p_h is a cutaway of the unit cube to show the oscillations of the exact solution on the interior. The convergence in pressure is first-order, and the numerical solution has no artifacts near the corners.

In conclusion, this seems like the implementation of Bernardi-Raugel finite elements in `deal.II` is very promising. In order to do more appropriate testing, they should be



y-velocity z-velocity

Figure 5: Numerical solution for example 3, $n = 2^4$

implemented in the step-22 program (which requires hard-coding the boundary conditions specifically for that element). This is one target of my upcoming work this summer, and another target is to implement these elements for linear elasticity and perform some analysis to see if a locking-free method can be extracted from them. Both of these will confirm the implementation is correct and ready to be published in the library officially.