Utilizing deal.II to simulate the EMAC formulation for the Navier-Stokes Equations

Sean Ingimarson

Clemson University

Utilizing deal.II to simulate the EMAC formulation for the Navier-Stokes Equations

December 7, 2020

EMAC formulation of Navier-Stokes

EMAC-Reg, or Energy, Momentum, and Angular Momentum Conserving Regularization.

$$u_t + \underbrace{2D(u)u + (\nabla \cdot u)u}_{\text{new nonlinear term}} + \nabla p - \nu \Delta u = f,$$

$$\nabla \cdot u = 0.$$

- Mathematically equivalent to NSE
- Replace nonlinear term $u \cdot \nabla u$
- Quantity conserving formulations perform better, more physically relevant

EMAC-Reg formulation of Navier-Stokes

EMAC-Reg, or Energy, Momentum, and Angular Momentum Conserving Regularization.

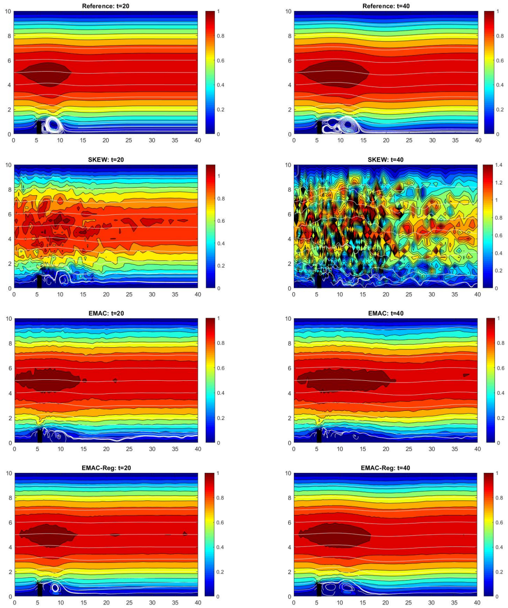
$$
\begin{aligned}
u_t + 2D(w)w + (\nabla \cdot w)w + \nabla p - \nu \Delta u &= f, \\
-\alpha^2 \Delta w + w &= u, \\
\nabla \cdot w &= 0.
\end{aligned}
$$

- Introduce spatial filter for better performance on coarser meshes
- Didn't have time to code it, also difficult

Plan for implementation

- Learn step-57 (Steady NSE on box)
- Implement time loop using DiscreteTime class
- Switch formulation from NSE to EMAC (nonlinear term is $D(u)u + (\nabla \cdot u)u$), then EMAC-Reg
- Benchmark Problem for convergence analysis

Results to expect (Step Problem)



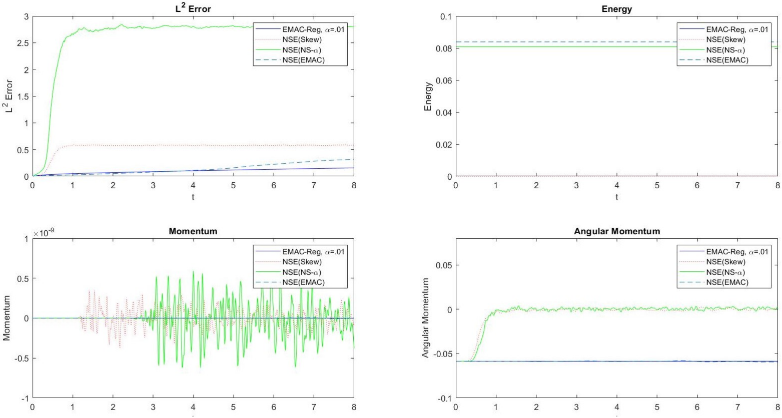
Results to expect (convergence analysis)

$$w = \begin{bmatrix} -\cos(\pi x)\sin(\pi y) \\ \sin(\pi x)\cos(\pi y) \end{bmatrix} e^{-2\pi^2 \nu t},$$
$$u = (1 + 2\pi^2\alpha^2)w,$$
$$p = -w \cdot \nabla w,$$

h	$\|\|w - w_h\|\|_{\infty,0}$	Rate	$\|\|w - w_h\|\|_{\infty,1}$	Rate
1/2	3.68383e-02	-	6.40968e-01	-
1/4	8.01551e-03	2.20034	2.15719e-01	1.57110
1/8	7.40875e-04	3.43549	4.88487e-02	2.14276
1/16	8.07978e-05	3.19684	1.13421e-02	2.10663
1/32	9.78736e-06	3.04532	2.77000e-03	2.03374
1/64	1.21936e-06	3.00480	6.89712e-04	2.00582
1/128	1.64691e-07	2.88829	1.72432e-04	1.99996

Table 1: Convergence results for both u and w for EMAC-Reg (should be similar to EMAC)

Results to expect (Computed Quantities)



General idea of step 57

- Nonlinear term $(u \cdot \nabla u)$ in NSE is problematic
- Spatially discretize so we get

$$u \cdot \nabla u \approx u_k \cdot \nabla u_{k+1} + u_{k+1} \cdot \nabla u_k - u_k \cdot \nabla u_k$$

- Convergence is quadratic, expect convergence in 2-3 iterations
- Do the same exact thing for EMAC formulation

$$2D(u)u \approx 2D(u_k)u_{k+1} + 2D(u_{k+1})u_k - 2D(u_k)u_k$$
$$(\nabla \cdot u)u \approx (\nabla \cdot u_k)u_{k+1} + (\nabla \cdot u_{k+1})u_k - (\nabla \cdot u_k)u_k$$

Time implementation

NSE assembly (Identical to step-57)

```
            {
              if (assemble_matrix)
                {
                  for (unsigned int j = 0; j < dofs_per_cell; ++j)
                    {
                      local_matrix(i, j) +=
                        (viscosity * scalar_product(grad_phi_u[j], grad_phi_u[i])
                         + present_velocity_gradients[q] * phi_u[j] * phi_u[i]
                         + grad_phi_u[j] * present_velocity_values[q] * phi_u[i]
                         - div_phi_u[i] * phi_p[j]
                         - phi_p[i] * div_phi_u[j]
                         + gamma * div_phi_u[j] * div_phi_u[i]

                         + phi_p[i] * phi_p[j]
                         // Time dependent component
                         + (1.0 / dt) * phi_u[j] * phi_u[i]
                         )
                         * fe_values.JxW(q);
                    }
                }

              double present_velocity_divergence =
                trace(present_velocity_gradients[q]);
              local_rhs(i) +=
                (- viscosity * scalar_product(present_velocity_gradients[q],grad_phi_u[i])
                 - present_velocity_gradients[q] * present_velocity_values[q] * phi_u[i]
                 + present_pressure_values[q] * div_phi_u[i]
                 + present_velocity_divergence * phi_p[i]
                 - gamma * present_velocity_divergence * div_phi_u[i]
                 // time dependent terms
                 + (1.0 / dt) * old_solution_values[q] * phi_u[i]
                 - (1.0 / dt) * present_velocity_values[q] * phi_u[i]
                 )
                 * fe_values.JxW(q);
            }
        }

      cell->get_dof_indices(local_dof_indices);
```

EMAC assembly

step-57.cc @ step-57 - Qt Creator

StationaryNavierStokes<dim>::assemble(const bool, const bool, const double) -> void

```cpp
                {
                  local_matrix(i, j) +=
                    (viscosity * scalar_product(grad_phi_u[j], grad_phi_u[i])
                     //--------------------------------------------------------
                     // EMAC terms
                     + 2 * present_velocity_symgrad[q] * phi_u[j] * phi_u[i]          // (2D(u_k)u_{k+1},v)
                     + 2 * symgrad_phi_u[j] * present_velocity_values[q] * phi_u[i]   // (2D(u_{k+1})u_k,v)

                     + div_phi_u[j] * present_velocity_values[q] * phi_u[i]  // ((div u_{k+1})u_k,v)
                     + present_velocity_divergence * phi_u[j] * phi_u[i]     // ((div u_k)u_{k+1},v)
                     //--------------------------------------------------------
                     - div_phi_u[i] * phi_p[j]
                     - phi_p[i] * div_phi_u[j]
                     + gamma * div_phi_u[j] * div_phi_u[i]

                     + phi_p[i] * phi_p[j]
                     // Time dependent component
                     + (1.0 / dt) * phi_u[j] * phi_u[i]
                     )
                     * fe_values.JxW(q);
                }
            }

          local_rhs(i) +=
            (- viscosity * scalar_product(present_velocity_gradients[q],grad_phi_u[i])
             //--------------------------------------------------------------------
             // EMAC terms
             - 2 * present_velocity_symgrad[q] * present_velocity_values[q] * phi_u[i]      // (2D(u_k)u_k,v)
             - present_velocity_divergence * present_velocity_values[q] * phi_u[i]          // ((div u_k)u_k,v)
             //----------------------------------------------------------------------
             + present_pressure_values[q] * div_phi_u[i]
             + present_velocity_divergence * phi_p[i]
             - gamma * present_velocity_divergence * div_phi_u[i]
             // time dependent terms
             + (1.0 / dt) * old_solution_values[q] * phi_u[i]
             - (1.0 / dt) * present_velocity_values[q] * phi_u[i]
             )
             * fe_values.JxW(q);
        }
```

Future work

- Implement EMAC-Reg, test in 3D to see coarse mesh effectiveness.
- Implement MPI, solving on one core in 3D is incredibly inefficient.
- Calculate energy, momentum, and angular momentum for appropriate problem in deal.II.
- Implement benchmark problem from slide 6.