University of Glasgow

Glasgow Computational Engineering Centre

MOFEM
mofem.eng.gla.ac.uk

9 April 2021

Linear Acoustics

Ignatios Athanasiadis, Łukasz Kaczmarczyk

WORLD CHANGING GLASGOW

MOFEM
mofem.eng.gla.ac.uk

Layout of Linear Acoustics Session

- Goal of the presentation
- Boundary value problem description
- Strong and weak form
- Code dissection
- Post processing and results

MOFEM
mofem.eng.gla.ac.uk

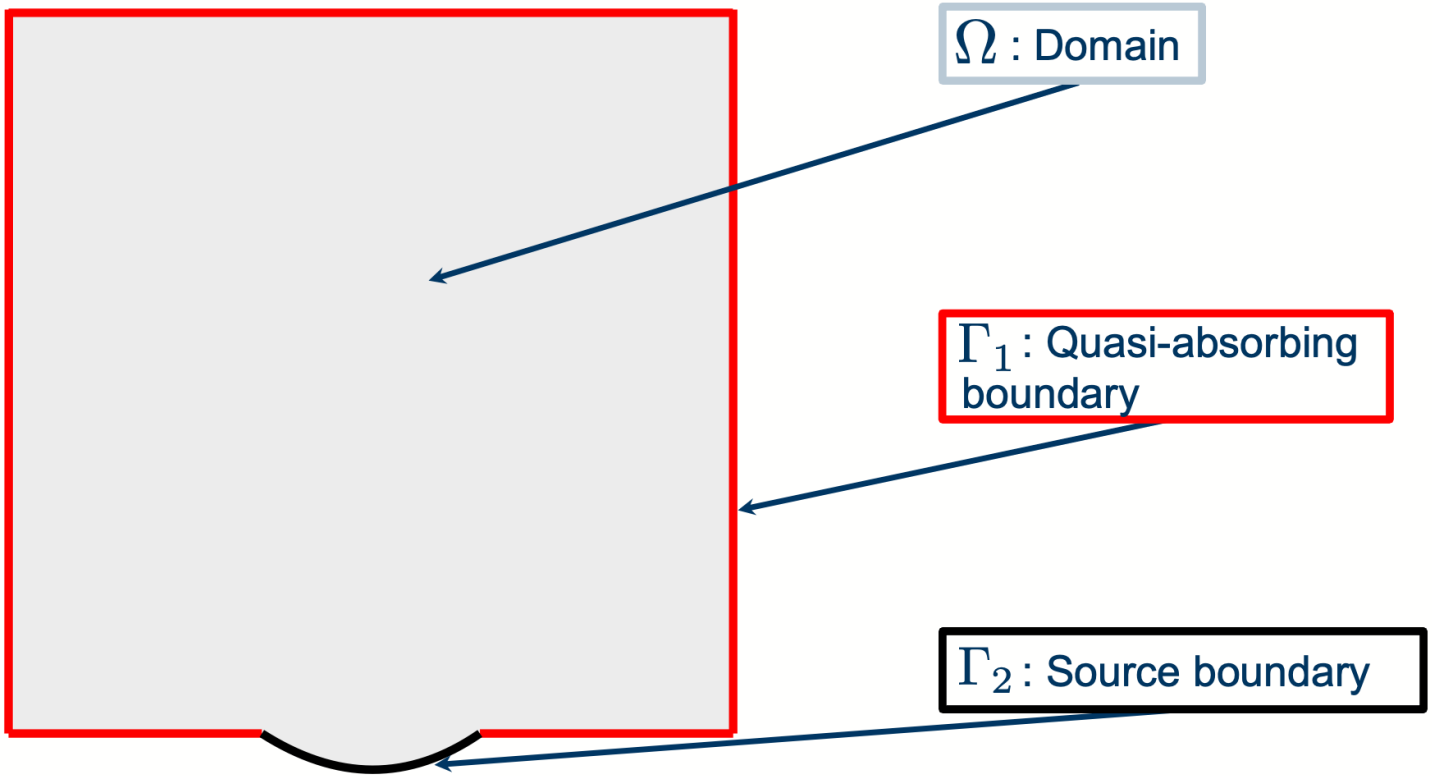
Goal of the presentation

- Solving complex field problems
- Dealing with two fields in MoFEM
- Pushing **Form Integrator** in pipelines

Tutorial clx-0 :
http://mofem.eng.gla.ac.uk/mofem/html/tutorial_hemholtz_problem.html

MOFEM
mofem.eng.gla.ac.uk

Transducer generator



Strong form

The wave equation:

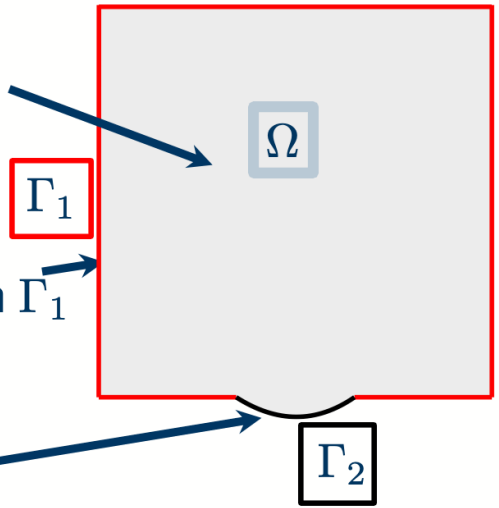
$$\frac{\partial^2 P(\mathbf{x},t)}{\partial t^2} - c^2 \nabla^2 P(\mathbf{x},t) = 0 \quad \text{in } \Omega$$

Bayliss-Turkel boundary conditions:

$$\frac{\partial P(\mathbf{x},t)}{\partial t} + c\,(\mathbf{N} \cdot \nabla P(\mathbf{x},t)) = 0 \quad \text{on } \Gamma_1$$

Dirichlet boundary conditions:

$$P(\mathbf{x},t) = \cos(\omega t) \quad \text{on } \Gamma_2$$



c : Wave speed

P: Pressure field

t : time

$\mathbf{N}$: normal vector on Γ_1

ω : frequency

$p(\mathbf{x})$: spatially varying pressure

Solution by separation of variables (standing wave):

$$P(\mathbf{x},t) = \mathrm{Re}\left(p(\mathbf{x})e^{-i\omega t}\right)$$

Alvin Bayliss and Eli Turkel. Radiation boundary conditions for wave-like equations. Communications on Pure and applied Mathematics, 33(6):707– 725, 1980

MOFEM
mofem.eng.gla.ac.uk

Weak formulation

The Helmholtz problem:

$$\left.\begin{aligned} k^2 p^{\mathrm{Re}} + \nabla^2 p^{\mathrm{Re}} &= 0 \\ k^2 p^{\mathrm{Im}} + \nabla^2 p^{\mathrm{Im}} &= 0 \end{aligned}\right\} \text{in } \Omega \qquad \left.\begin{aligned} \mathbf{N}\cdot\nabla p^{\mathrm{Re}} + k p^{\mathrm{Im}} &= 0 \\ \mathbf{N}\cdot\nabla p^{\mathrm{Im}} - k p^{\mathrm{Re}} &= 0 \end{aligned}\right\} \text{on } \Gamma_1 \qquad \left.\begin{aligned} p^{\mathrm{Re}} &= 1 \\ p^{\mathrm{Im}} &= 0 \end{aligned}\right\} \text{on } \Gamma_2$$

For $k = \omega/c$, $p(\mathbf{x}) = p^{\mathrm{Re}} + i p^{\mathrm{Im}}$ and $p^{\mathrm{Re}}, p^{\mathrm{Im}} \in H^1(\Omega)$ find p^{Re} and p^{Im} to solve the W.F.

The discrete weak form:

$$p^{\mathrm{Re}} \approx p^{h\mathrm{Re}} = \sum_{j=0}^{N-1} \phi_j \bar{p}_j^{\mathrm{Re}} \qquad p^{\mathrm{Im}} \approx p^{h\mathrm{Im}} = \sum_{j=0}^{N-1} \psi_j \bar{p}_j^{\mathrm{Im}}$$

$$\begin{bmatrix} k^2 \int_\Omega \phi_i \phi_j \mathrm{d}\Omega - \int_\Omega \nabla\phi_i \nabla\phi_j \mathrm{d}\Omega & -k \int_{\Gamma_1} \phi_i \psi_j \mathrm{d}\Gamma_1 \\ k \int_{\Gamma_1} \psi_i \phi_j \mathrm{d}\Gamma_1 & +k^2 \int_\Omega \psi_i \psi_j \mathrm{d}\Omega - \int_\Omega \nabla\psi_i \nabla\psi_j \mathrm{d}\Omega \end{bmatrix} \begin{bmatrix} \bar{p}_j^{\mathrm{Re}} \\ \bar{p}_j^{\mathrm{Im}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \text{+ Essential B.C.}$$

F. Ihlenburg, I. Babuška, Finite element solution of the Helmholtz equation with high wave number. I. the h-version of the FEM, Comput. Math. Appl. 30 (9) (1995) 9–37.

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
MoFEMErrorCode Example::runProblem() {
  MoFEMFunctionBegin;
  CHKERR readMesh();
  CHKERR setupProblem();
  CHKERR boundaryCondition();
  CHKERR assembleSystem();
  CHKERR solveSystem();
  CHKERR outputResults();
  CHKERR checkResults();
  MoFEMFunctionReturn(0);
}
```

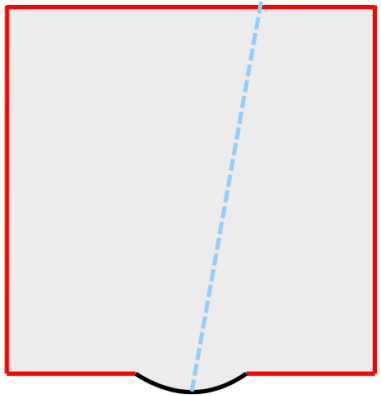
```
MoFEMErrorCode Example::setupProblem() {
  MoFEMFunctionBegin;
  // Add field
  CHKERR simpleInterface->addDomainField("P_REAL", H1,
                                         AINSWORTH_BERNSTEIN_BEZIER_BASE, 1);
  CHKERR simpleInterface->addDomainField("P_IMAG", H1,
                                         AINSWORTH_BERNSTEIN_BEZIER_BASE, 1);
  CHKERR simpleInterface->addBoundaryField("P_REAL", H1,
                                           AINSWORTH_BERNSTEIN_BEZIER_BASE, 1);
  CHKERR simpleInterface->addBoundaryField("P_IMAG", H1,
                                           AINSWORTH_BERNSTEIN_BEZIER_BASE, 1);
  int order = 6;
  CHKERR PetscOptionsGetInt(PETSC_NULL, "", "-order", &order, PETSC_NULL);
  CHKERR simpleInterface->setFieldOrder("P_REAL", order);
  CHKERR simpleInterface->setFieldOrder("P_IMAG", order);
  CHKERR simpleInterface->setUp();
  MoFEMFunctionReturn(0);
}
```

- Field names: P_REAL, P_IMAG
- Space: H^1 (scalar space)
- Base: AINSWORTH_BERSTEIN_BEZIER_BASE
- Number of coefficients per shape function: 1 (scalar field)

Implementation

```cpp
MoFEMErrorCode Example::runProblem() {
  MoFEMFunctionBegin;
  CHKERR readMesh();
  CHKERR setupProblem();
  CHKERR boundaryCondition();
  CHKERR assembleSystem();
  CHKERR solveSystem();
  CHKERR outputResults();
  CHKERR checkResults();
  MoFEMFunctionReturn(0);
}
```

```cpp
MoFEMErrorCode Example::boundaryCondition() {
  MoFEMFunctionBegin;

  auto get_ents_on_mesh_skin = [&]() {
    Range boundary_entities;
    for (_IT_CUBITMESHSETS_BY_SET_TYPE_FOR_LOOP_(mField, BLOCKSET, it)) {
      std::string entity_name = it->getName();
      if (entity_name.compare(0, 2, "BC") == 0) {
        CHKERR it->getMeshsetIdEntitiesByDimension(mField.get_moab(), 1,
                                                   boundary_entities, true);
      }
    }
    // Add vertices to boundary entities
    Range boundary_vertices;
    CHKERR mField.get_moab().get_connectivity(boundary_entities,
                                              boundary_vertices, true);
    boundary_entities.merge(boundary_vertices);

    return boundary_entities;
  };
```



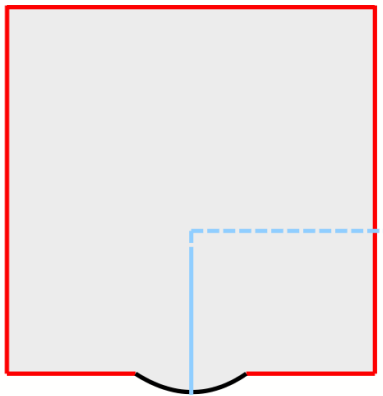
Function continuation

Implementation

Function continuation

```
MoFEMErrorCode Example::runProblem() {
  MoFEMFunctionBegin;
  CHKERR readMesh();
  CHKERR setupProblem();
  CHKERR boundaryCondition();
  CHKERR assembleSystem();
  CHKERR solveSystem();
  CHKERR outputResults();
  CHKERR checkResults();
  MoFEMFunctionReturn(0);
}
```

```
auto mark_boundary_dofs = [&](Range &&skin_edges) {
    auto problem_manager = mField.getInterface<ProblemsManager>();
    auto marker_ptr = boost::make_shared<std::vector<unsigned char>>();
    problem_manager->markDofs(simpleInterface->getProblemName(), ROW,
                              skin_edges, *marker_ptr);
    return marker_ptr;
  };

  auto remove_dofs_from_problem = [&](Range &&skin_edges) {
    MoFEMFunctionBegin;
    auto problem_manager = mField.getInterface<ProblemsManager>();
    CHKERR problem_manager->removeDofsOnEntities(
        simpleInterface->getProblemName(), "P_IMAG", skin_edges, 0, 1);
    MoFEMFunctionReturn(0);
  };

  // Get global local vector of marked DOFs. Is global, since is set for all
  // DOFs on processor. Is local since only DOFs on processor are in the
  // vector. To access DOFs use local indices.
  boundaryMarker = mark_boundary_dofs(get_ents_on_mesh_skin());
  CHKERR remove_dofs_from_problem(get_ents_on_mesh_skin());

  MoFEMFunctionReturn(0);
}
```

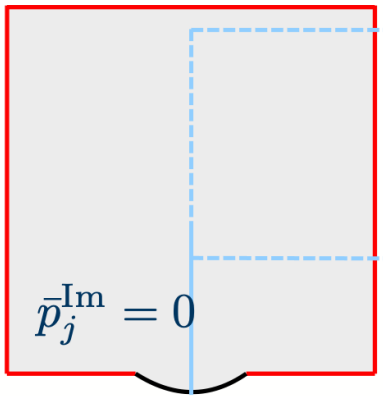


Implementation

Function continuation

```
MoFEMErrorCode Example::runProblem() {
  MoFEMFunctionBegin;
  CHKERR readMesh();
  CHKERR setupProblem();
  CHKERR boundaryCondition();
  CHKERR assembleSystem();
  CHKERR solveSystem();
  CHKERR outputResults();
  CHKERR checkResults();
  MoFEMFunctionReturn(0);
}
```

```
auto mark_boundary_dofs = [&](Range &&skin_edges) {
    auto problem_manager = mField.getInterface<ProblemsManager>();
    auto marker_ptr = boost::make_shared<std::vector<unsigned char>>();
    problem_manager->markDofs(simpleInterface->getProblemName(), ROW,
                              skin_edges, *marker_ptr);
    return marker_ptr;
  };

  auto remove_dofs_from_problem = [&](Range &&skin_edges) {
    MoFEMFunctionBegin;
    auto problem_manager = mField.getInterface<ProblemsManager>();
    CHKERR problem_manager->removeDofsOnEntities(
        simpleInterface->getProblemName(), "P_IMAG", skin_edges, 0, 1);
    MoFEMFunctionReturn(0);
  };

  // Get global local vector of marked DOFs. Is global, since is set for all
  // DOFs on processor. Is local since only DOFs on processor are in the
  // vector. To access DOFs use local indices.
  boundaryMarker = mark_boundary_dofs(get_ents_on_mesh_skin());
  CHKERR remove_dofs_from_problem(get_ents_on_mesh_skin());

  MoFEMFunctionReturn(0);
}
```



MOFEM
mofem.eng.gla.ac.uk

Implementation

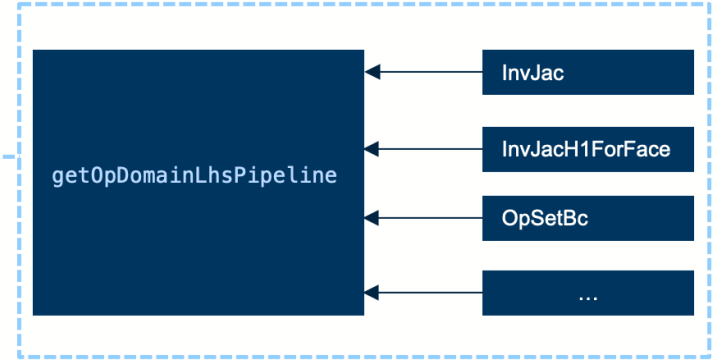
```
MoFEMErrorCode Example::runProblem() {
  MoFEMFunctionBegin;
  CHKERR readMesh();
  CHKERR setupProblem();
  CHKERR boundaryCondition();
  CHKERR assembleSystem();
  CHKERR solveSystem();
  CHKERR outputResults();
  CHKERR checkResults();
  MoFEMFunctionReturn(0);
}
```

```
MoFEMErrorCode Example::assembleSystem() {
  MoFEMFunctionBegin;
  PipelineManager *pipeline_mng = mField.getInterface<PipelineManager>();

  double k = 90;
  CHKERR PetscOptionsGetScalar(PETSC_NULL, "", "-k", &k, PETSC_NULL);

  auto beta = [](const double, const double, const double) { return -1; };
  auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
  auto kp = [k](const double, const double, const double) { return k; };
  auto km = [k](const double, const double, const double) { return -k; };
  auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

  auto set_domain = [&]() {
      <. . .>
  };

  auto set_boundary = [&]() {
       <. . .>
  };

  CHKERR set_domain();
  CHKERR set_boundary();

  MoFEMFunctionReturn(0);
}
```

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```



MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

$$J^{-1} = \frac{\partial \xi_i}{\partial X_j}$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

$$(\nabla\phi)_j = \frac{\partial\phi}{\partial\xi_i}\frac{\partial\xi_i}{\partial X_j}$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

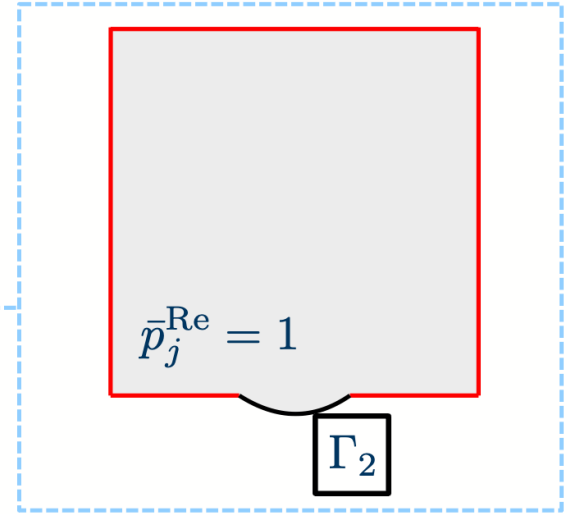
```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

$$(\nabla\phi)_j = \frac{\partial\phi}{\partial\xi_i}\frac{\partial\xi_i}{\partial X_j}$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```



MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainGradGrad =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpGradGrad<1, 1, 2>;
```

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainGradGrad =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpGradGrad<1, 1, 2>;
```

$$-\int_{\Omega} \nabla\phi_i \nabla\phi_j \mathrm{d}\Omega$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainGradGrad =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpGradGrad<1, 1, 2>;
```

$$-\int_{\Omega} \nabla\phi_i \nabla\phi_j \mathrm{d}\Omega$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainGradGrad =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpGradGrad<1, 1, 2>;
```

$$-\int_{\Omega} \nabla\psi_i \nabla\psi_j \mathrm{d}\Omega$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainMass =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainMass =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

$$k^2 \int_\Omega \phi_i \phi_j \, \mathrm{d}\Omega$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainMass =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

$$k^2 \int_\Omega \phi_i \phi_j \, \mathrm{d}\Omega$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainMass =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

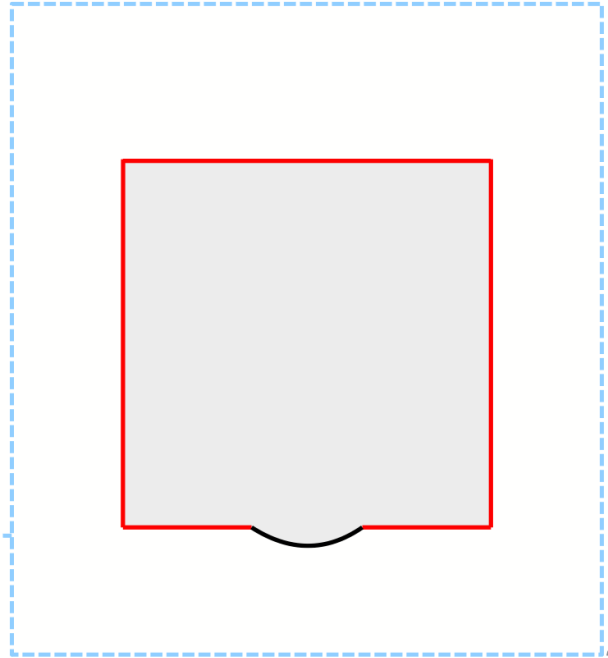
$$k^2 \int_\Omega \psi_i \psi_j \mathrm{d}\Omega$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

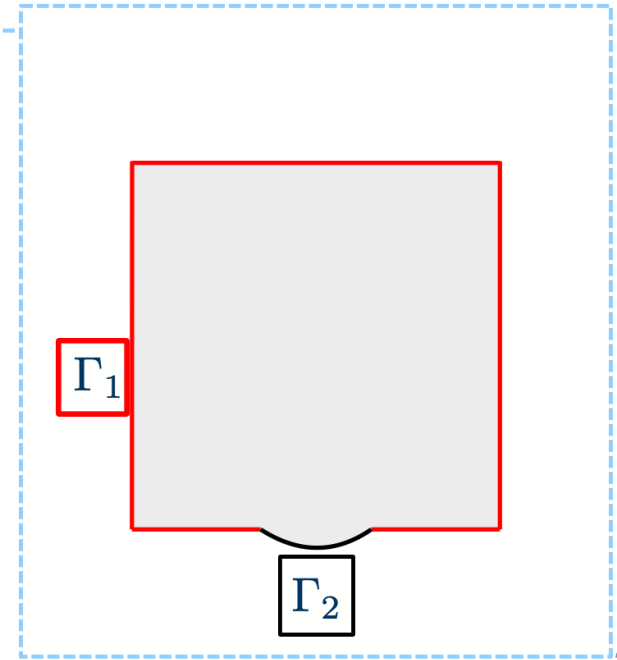
```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_domain = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpCalculateInvJacForFace(invJac));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetInvJacH1ForFace(invJac));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainGradGrad("P_IMAG", "P_IMAG", beta));

  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_REAL", "P_REAL", k2));
  pipeline_mng->getOpDomainLhsPipeline().push_back(
      new OpDomainMass("P_IMAG", "P_IMAG", k2));

  pipeline_mng->getOpDomainLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpDomainMass =
FormsIntegrators<DomainEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```



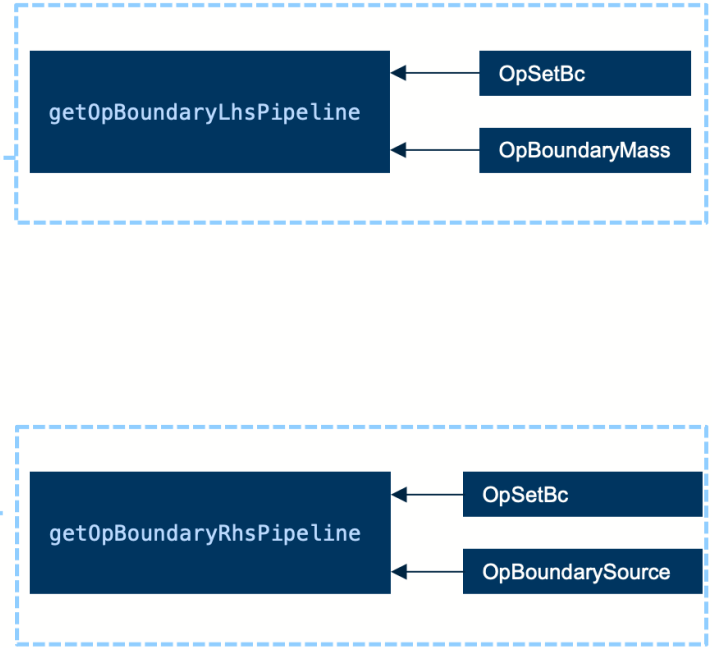
Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```



Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```



MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpBoundaryMass =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpBoundaryMass =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

$$k \int_{\Gamma_1} \psi_i \phi_j \mathrm{d}\Gamma_1$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpBoundaryMass =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

$$k \int_{\Gamma_1} \psi_i \phi_j \, \mathrm{d}\Gamma_1$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
   MoFEMFunctionBegin;
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpSetBc("P_REAL", true, boundaryMarker));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpBoundaryMass("P_IMAG", "P_REAL", kp));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpBoundaryMass("P_REAL", "P_IMAG", km));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpSetBc("P_REAL", false, boundaryMarker));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpBoundaryMass("P_REAL", "P_REAL", beta));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

   pipeline_mng->getOpBoundaryRhsPipeline().push_back(
       new OpSetBc("P_REAL", false, boundaryMarker));
   pipeline_mng->getOpBoundaryRhsPipeline().push_back(
       new OpBoundarySource("P_REAL", beta));
   pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

   CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
   CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
   MoFEMFunctionReturn(0);
};
```

```
using OpBoundaryMass =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

$$-k\int_{\Gamma_1}\phi_i\psi_j\,\mathrm{d}\Gamma_1$$

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpBoundaryMass =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::BiLinearForm<GAUSS>::OpMass<1, 1>;
```

$$-k\int_{\Gamma_1}\phi_i\psi_j\,\mathrm{d}\Gamma_1$$

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
   MoFEMFunctionBegin;
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpSetBc("P_REAL", true, boundaryMarker));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpBoundaryMass("P_IMAG", "P_REAL", kp));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpBoundaryMass("P_REAL", "P_IMAG", km));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpSetBc("P_REAL", false, boundaryMarker));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(
       new OpBoundaryMass("P_REAL", "P_REAL", beta));
   pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

   pipeline_mng->getOpBoundaryRhsPipeline().push_back(
       new OpSetBc("P_REAL", false, boundaryMarker));
   pipeline_mng->getOpBoundaryRhsPipeline().push_back(
       new OpBoundarySource("P_REAL", beta));
   pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

   CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
   CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
   MoFEMFunctionReturn(0);
};
```

```
using OpBoundarySource =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::LinearForm<GAUSS>::OpSource<1, 1>;
```

MOFEM
mofem.eng.gla.ac.uk

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

```
using OpBoundarySource =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::LinearForm<GAUSS>::OpSource<1, 1>;
```

$$-\int_{\Gamma_1} \psi_i \mathrm{d}\Gamma_1$$

Implementation

```
auto beta = [](const double, const double, const double) { return -1; };
auto k2 = [k](const double, const double, const double) { return pow(k, 2); };
auto kp = [k](const double, const double, const double) { return k; };
auto km = [k](const double, const double, const double) { return -k; };
auto integration_rule = [](int, int, int p_data) { return 2 * p_data; };

auto set_boundary = [&]() {
  MoFEMFunctionBegin;
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", true, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_IMAG", "P_REAL", kp));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_IMAG", km));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(
      new OpBoundaryMass("P_REAL", "P_REAL", beta));
  pipeline_mng->getOpBoundaryLhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpSetBc("P_REAL", false, boundaryMarker));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(
      new OpBoundarySource("P_REAL", beta));
  pipeline_mng->getOpBoundaryRhsPipeline().push_back(new OpUnSetBc("P_REAL"));

  CHKERR pipeline_mng->setDomainRhsIntegrationRule(integration_rule);
  CHKERR pipeline_mng->setBoundaryLhsIntegrationRule(integration_rule);
  MoFEMFunctionReturn(0);
};
```

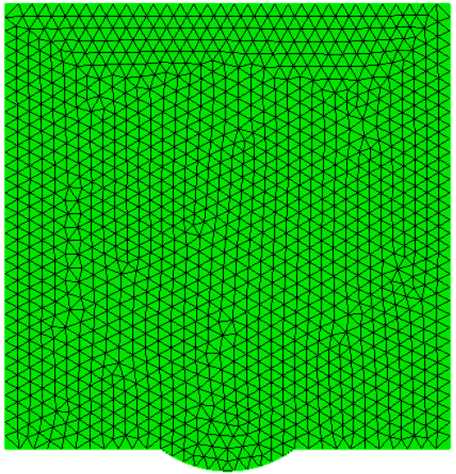
```
using OpBoundarySource =
FormsIntegrators<EdgeEleOp>::Assembly<
    PETSC>::LinearForm<GAUSS>::OpSource<1, 1>;
```

$$-\int_{\Gamma_1} \psi_i \mathrm{d}\Gamma_1$$

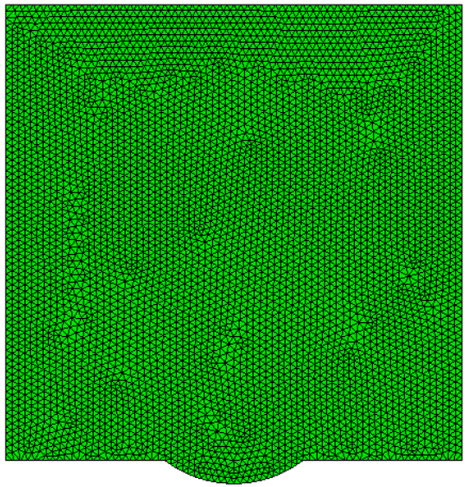
MOFEM
mofem.eng.gla.ac.uk

Meshes used

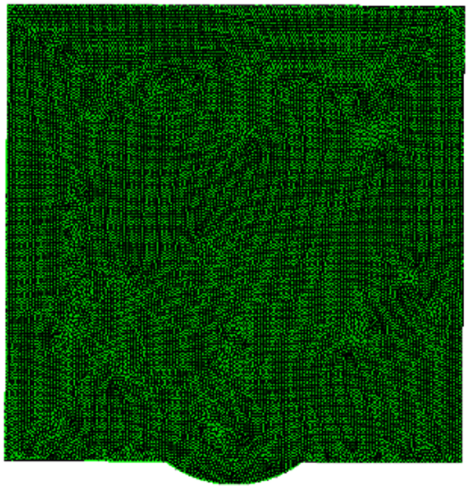
Coarse



Moderate



Fine

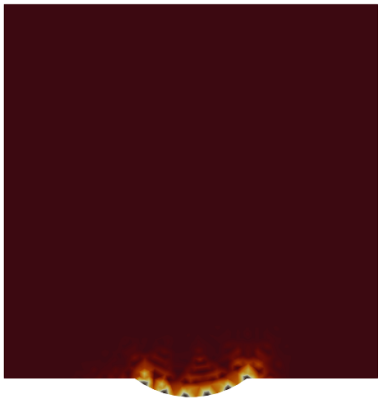


MOFEM
mofem.eng.gla.ac.uk

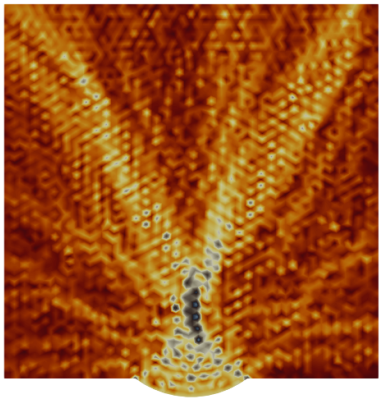
Influence of order choice

For all results: k = 180, coarse mesh

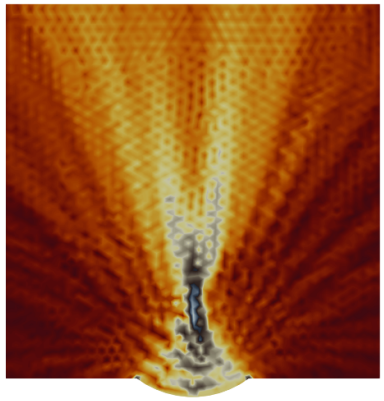
order = 1



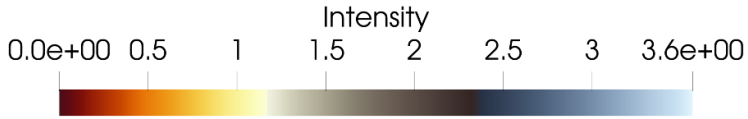
order = 2



order = 3



order = 4

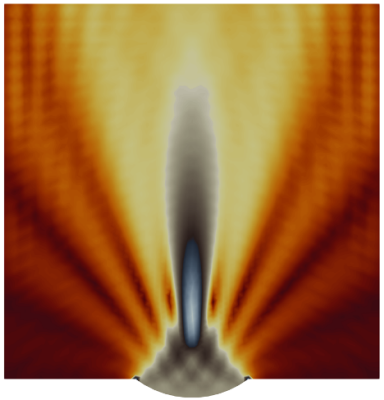


MOFEM
mofem.eng.gla.ac.uk

Influence of *k*

For all results: order = 8, coarse mesh

k = 90

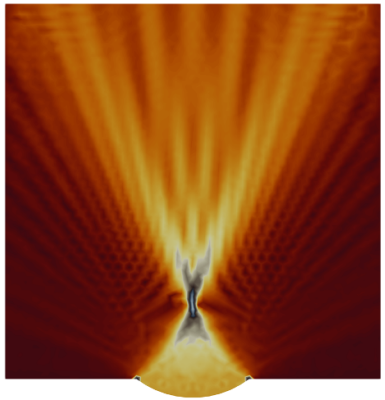


Intensity k = 90
0.0e+001 2.5e+00

k = 180

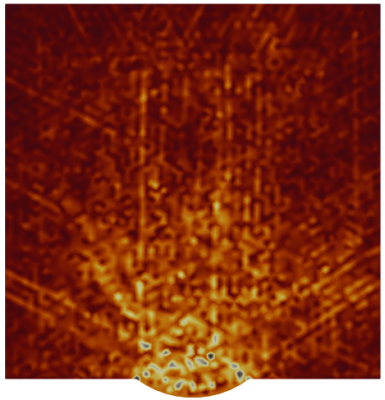
Intensity k = 180
0.0e+00 23.5e+00

k = 360



Intensit k =360
0.0e+00 4.8e+00

k = 720



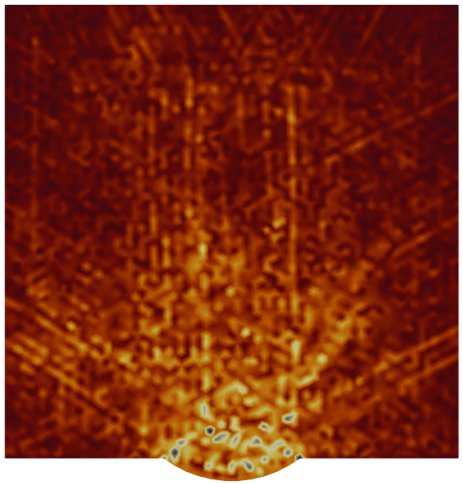
Intensity k =720
0.0e+00 6.8e+00

MOFEM
mofem.eng.gla.ac.uk

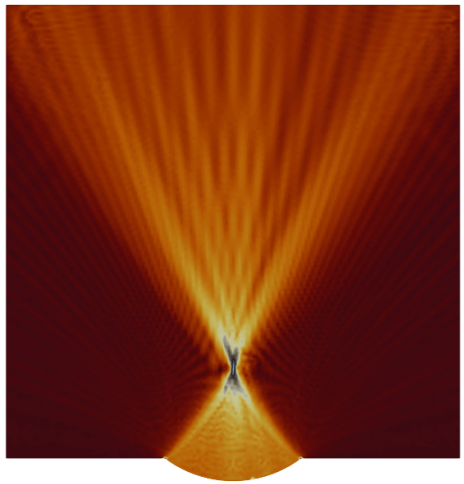
Mesh density influence

For all results: k= 720, order = 8

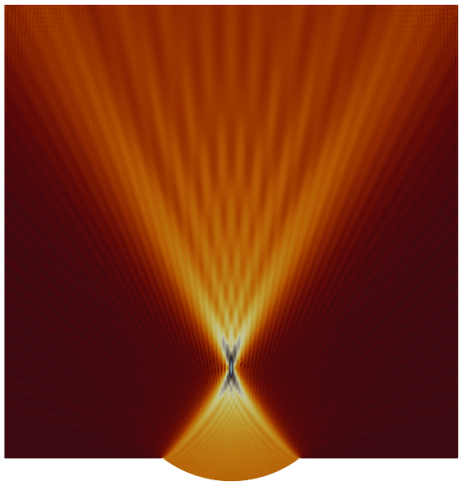
coarse mesh

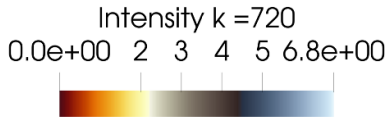


medium mesh



fine mesh



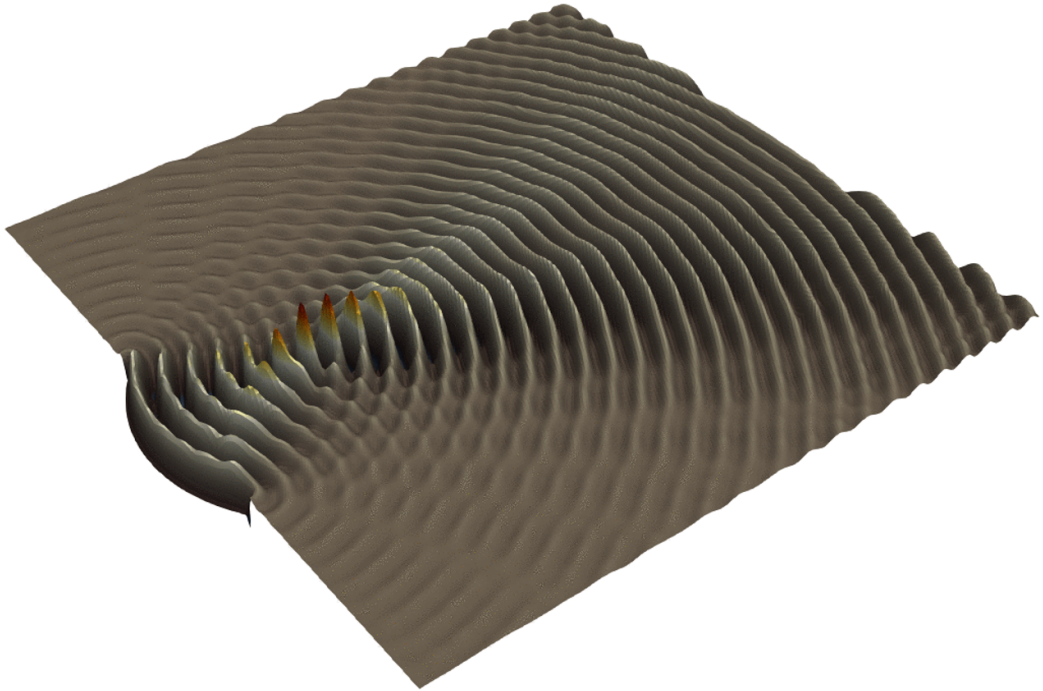


MOFEM
mofem.eng.gla.ac.uk

Wave propagation in post-processing

For: k = 180, order = 8, fine mesh

$$P(\mathbf{x},t) = p^{\mathrm{Re}}\cos(\omega t) + p^{\mathrm{Im}}\sin(\omega t)$$



MOFEM
mofem.eng.gla.ac.uk

Summary

- Solution of the Helmholtz problem
- Example of how to treat two fields in MoFEM
- Pushing **Form Integrator** in pipelines
- Influence of *h*- and *p*- refinement


University of Glasgow
GCC
Glasgow Computational Engineering Centre
MOFEM
mofem.eng.gla.ac.uk
Thank you for your attention!
WORLD CHANGING GLASGOW