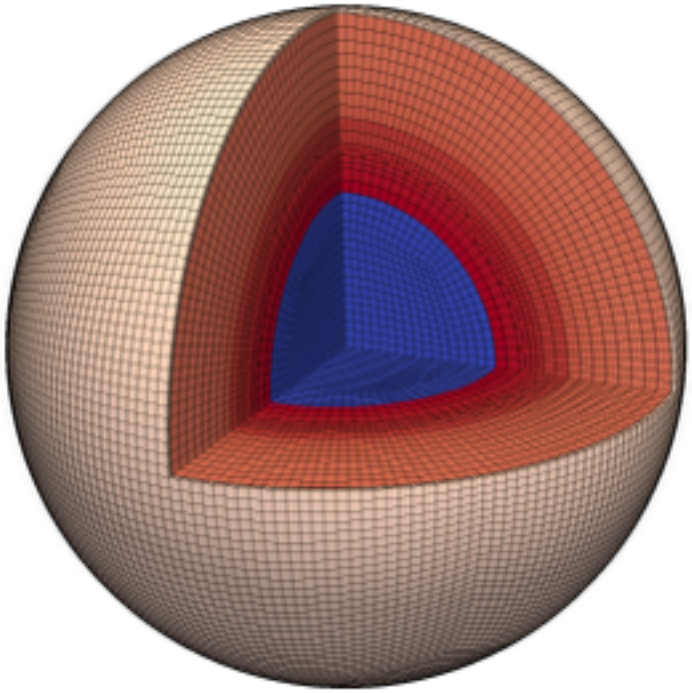
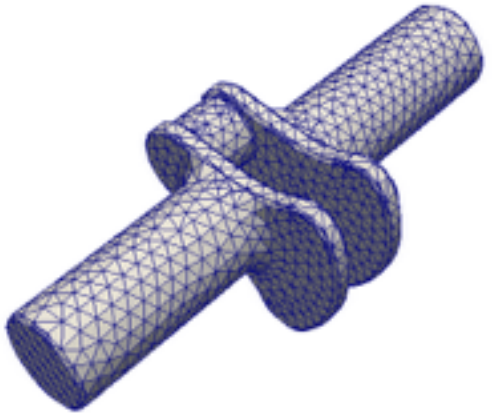
ETH zürich





Salvus: A flexible open-source package for full-waveform modelling and inversion

Michael Afanasiev, Christian Boehm, Martin van Driel, Lion Krischer, Dave May, Max Rietmann, Korbinian Sager, and Andreas Fichtner

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Platform for Advanced Scientific Computing

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

Full waveform inversion

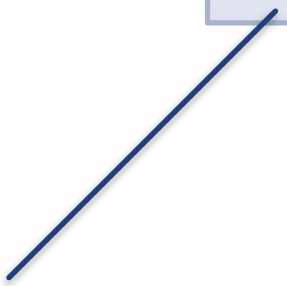
$$\partial_t^2 \mathbf{u}(\mathbf{x}) - \nabla \cdot \sigma(\mathbf{x}) - \mathbf{F} = 0$$

$$\sigma(\mathbf{x}) = \mathbf{c}(\mathbf{x}) : \nabla \mathbf{u}(\mathbf{x})$$

Full waveform inversion

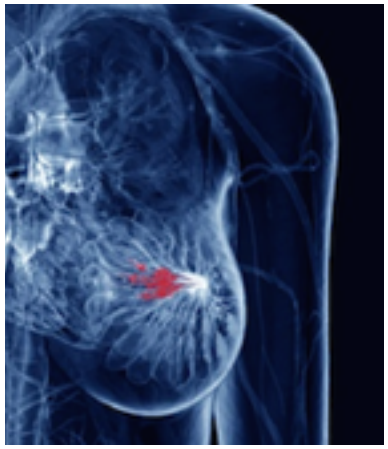
$$\partial_t^2 \mathbf{u}(\mathbf{x}) - \nabla \cdot \sigma(\mathbf{x}) - \mathbf{F} = 0$$

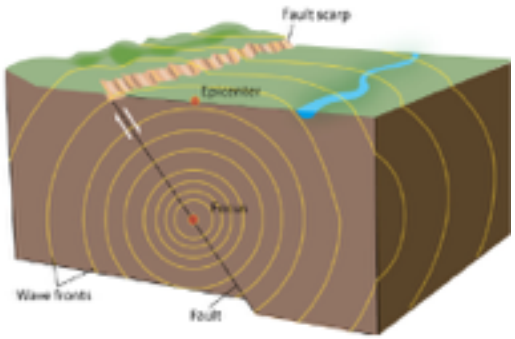
$$\sigma(\mathbf{x}) = \mathbf{c}(\mathbf{x}) : \nabla \mathbf{u}(\mathbf{x})$$

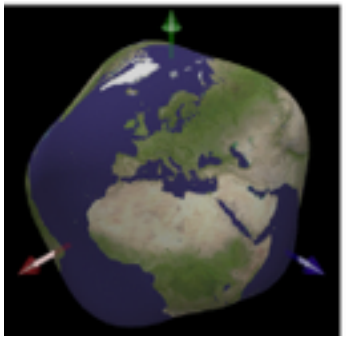


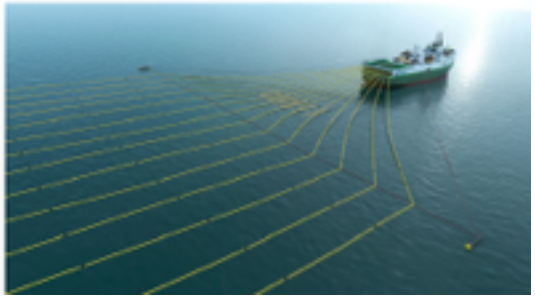
A problem of optimal control over **c**, constrained by the wave equation.

Motivation







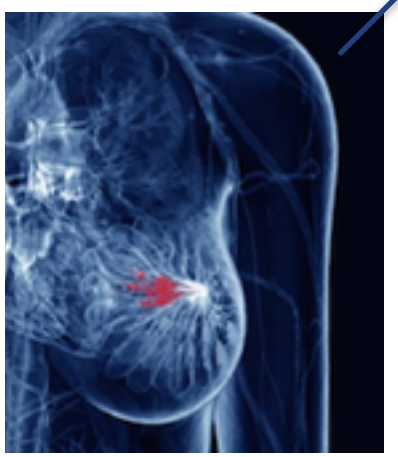


$$\sigma(\mathbf{x}) = \mathbf{c}(\mathbf{x}) : \nabla \mathbf{u}(\mathbf{x})$$

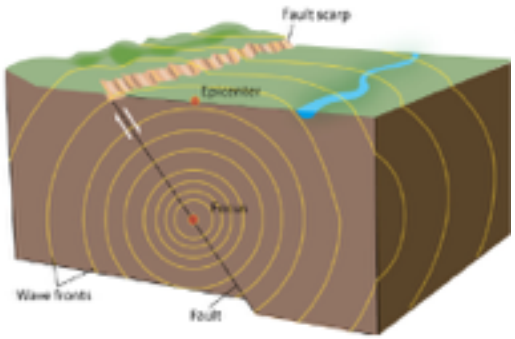
Takes on a very different character…

Motivation

Acoustic and attenuative 2D/3D propagation through tissue



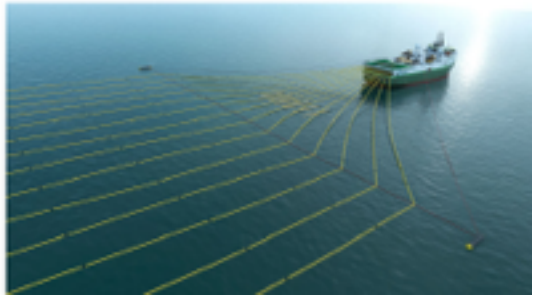
Elastic regional scale with topography



$$\sigma(\mathbf{x}) = \mathbf{c}(\mathbf{x}) : \nabla \mathbf{u}(\mathbf{x})$$

Takes on a very different character…

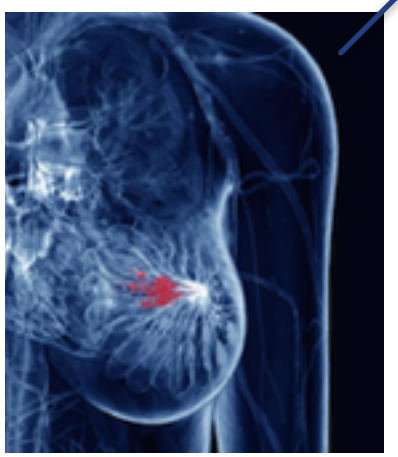
Elastic/Acoustic/ Gravity coupling, 3D, attenuative



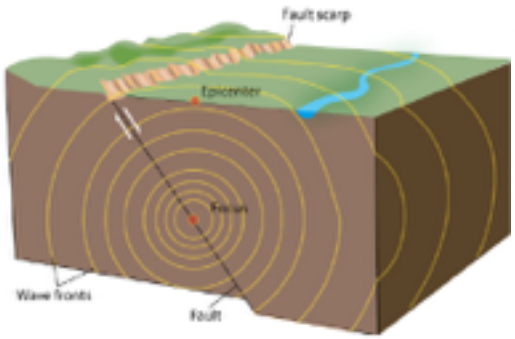
2D/3D, Elastic/Acoustic coupling, Topography, Attenuation

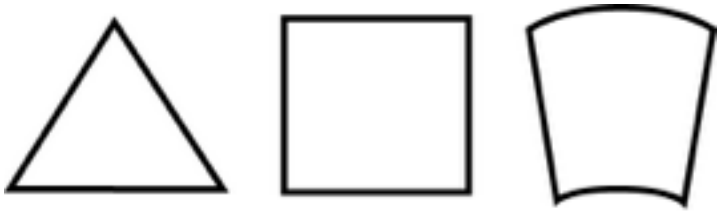
Motivation

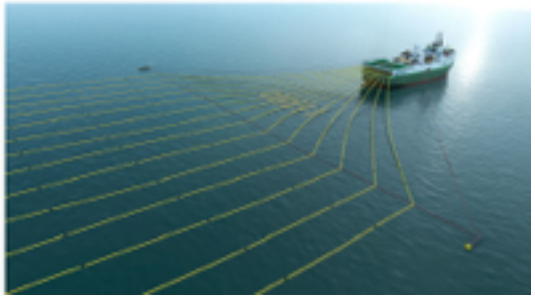
Acoustic and attenuative 2D/3D propagation through tissue

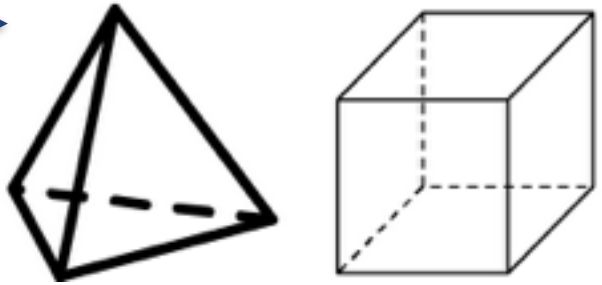


Elastic regional scale with topography







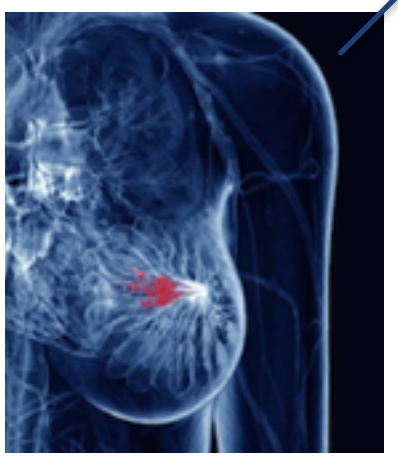


Elastic/Acoustic/ Gravity coupling, 3D, attenuative

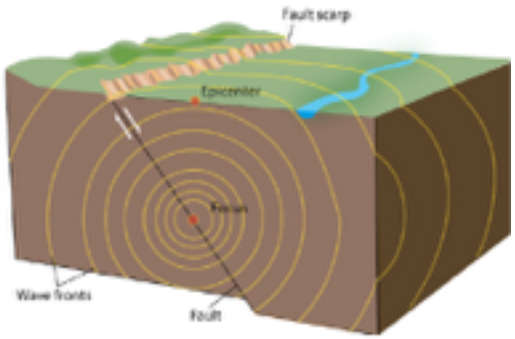
2D/3D, Elastic/Acoustic coupling, Topography, Attenuation

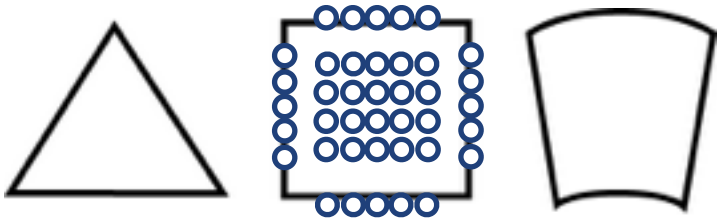
Motivation

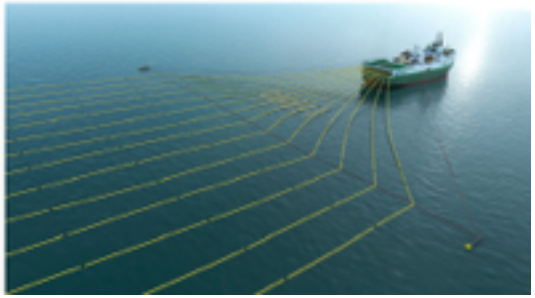
Acoustic and attenuative 2D/3D propagation through tissue

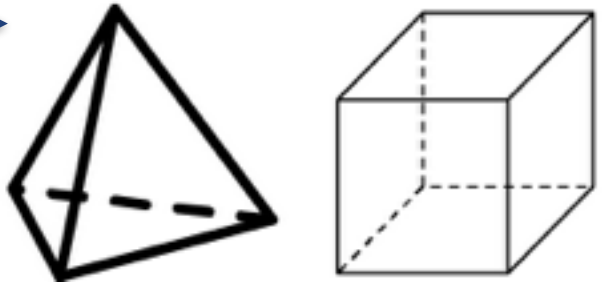


Elastic regional scale with topography









Elastic/Acoustic/ Gravity coupling, 3D, attenuative

2D/3D, Elastic/Acoustic coupling, Topography, Attenuation

Requirements

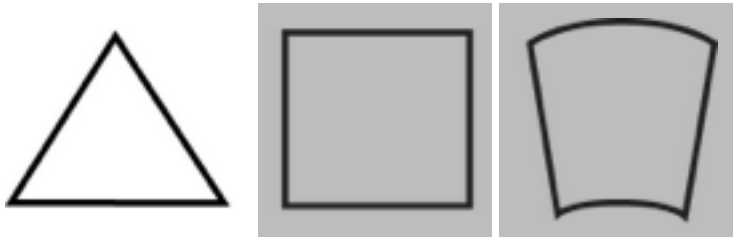
- Flexible and modular design, with support for the concurrent simulation of multiple coupled PDEs — composability
- Scalable, dimension independent spatial and temporal discretization
- Simple integration with external optimization libraries
- Correct and consistent solutions
- Speed

Flexible and modular design, with support for the concurrent simulation of multiple coupled PDEs

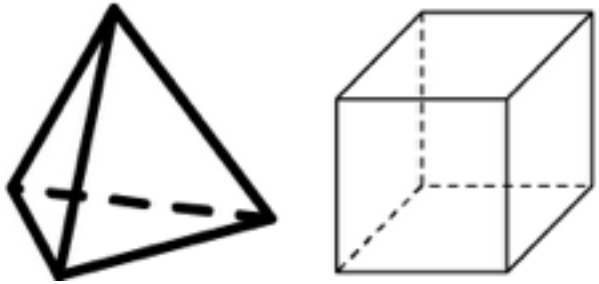
Modular design

Coupling physics…

Additional physics…



Physics 1 Physics 2 Physics …



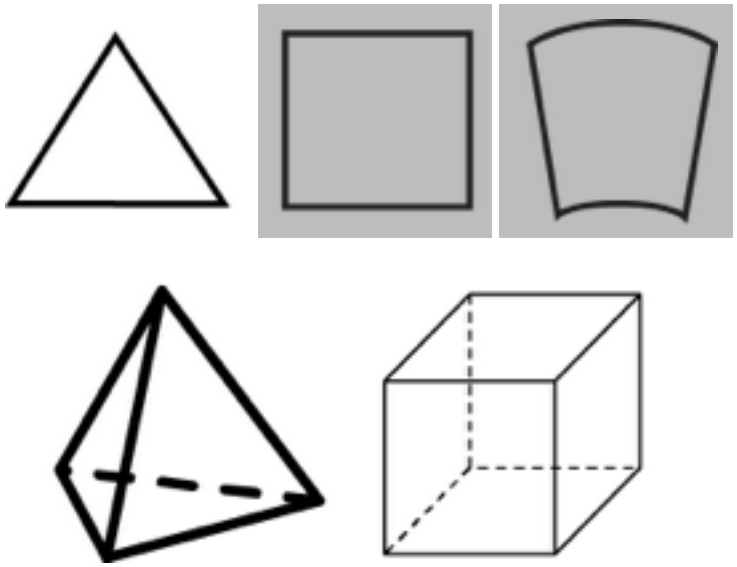
Quad Hex

FEM Basis 1 FEM Basis 2 FEM Basis …

Shape Mapping 1 Shape Mapping 2 Shape Mapping ….

Modular design (Composability)

```
Shape::computeJacobian();
Shape::interpolateParameters();
…

Element::computeSymmetricGradient();
Element::applyGradTestAndIntegrate();
…

Physics::computeStrain();
Physics::computeStress();
…

AdditionalPhysics::modifyStress();
…

CouplingPhysics::computeCouplingTerm();
…
```



…need to change any one of these independently.

Modular design: a solution with template mixins

```
template <typename Element>
MatrixXd Elastic2D<Element>::computeStiffnessTerm(const Eigen::MatrixXd &u) {

  // strain ux_x, ux_y, uy_x, uy_y.
  mStrain.leftCols<2>()  = Element::computeGradient(u.col(0));
  mStrain.rightCols<2>() = Element::computeGradient(u.col(1));

  // compute stress from strain.
  mStress = computeStress(mStrain);

  // temporary matrix to hold directional stresses.
  Matrix<double,Dynamic,2> temp_stress(Element::NumIntPnt(), 2);

  // compute stiffness.
  temp_stress.col(0) = mStress.col(0); temp_stress.col(1) = mStress.col(2);
  mStiff.col(0) = Element::applyGradTestAndIntegrate(temp_stress);
  temp_stress.col(0) = mStress.col(2); temp_stress.col(1) = mStress.col(1);
  mStiff.col(1) = Element::applyGradTestAndIntegrate(temp_stress);

  return mStiff;

}
```

Modular design: a solution with template mixins

```
template <typename Element>
MatrixXd Elastic2D<Element>::computeStiffnessTerm(const Eigen::MatrixXd &u) {

  // strain ux_x, ux_y, uy_x, uy_y.
  mStrain.leftCols<2>()  = Element::computeGradient(u.col(0));
  mStrain.rightCols<2>() = Element::computeGradient(u.col(1));

  // compute stress from strain.
  mStress = computeStress(mStrain);

  // temporary matrix to hold directional stresses.
  Matrix<double,Dynamic,2> temp_stress(Element::NumIntPnt(), 2);

  // compute stiffness.
  temp_stress.col(0) = mStress.col(0); temp_stress.col(1) = mStress.col(2);
  mStiff.col(0) = Element::applyGradTestAndIntegrate(temp_stress);
  temp_stress.col(0) = mStress.col(2); temp_stress.col(1) = mStress.col(1);
  mStiff.col(1) = Element::applyGradTestAndIntegrate(temp_stress);

  return mStiff;

}
```

Modular design: a solution with template mixins

```
template <typename Physics>
MatrixXd Attenuation<Physics>::computeStiffnessTerm(const Eigen::MatrixXd &u)
{

  strain = Physics::computeGradient(u);

  /* Modify strain by subtracting from memory variable equations… */

  stress = Physics::computeStress(strain);

  stiff = Physics::applyGradTestAndIntegrate(stress);

  return stiff;

}
```

Modular design: a solution with template mixins

Building up elements…

QuadP1
QuadP2
TensorBasis
Acoustic
Elastic2D
Elastic3D
Cpl2Acoustic
Cpl2Elastic
Attenuation
…

Modular design: a solution with template mixins

Building up elements…

```
QuadP1
QuadP2
TensorBasis
Acoustic
Elastic2D
Elastic3D
Cpl2Acoustic
Cpl2Elastic
Attenuation
```

…

Adding attenuation to surface elements….

```
ElementAdapter<
  Attenuation<
    Elastic2D<
      Quad<
        QuadP2>>>> AtnElasticQuadP1;
```

Modular design: a solution with template mixins

Building up elements…

```
QuadP1
QuadP2
TensorBasis
Acoustic
Elastic2D
Elastic3D
Cpl2Acoustic
Cpl2Elastic
Attenuation
…
```

Adding attenuation to surface elements….

```
ElementAdapter<
  Attenuation<
    Elastic2D<
      Quad<
        QuadP2>>>> AtnElasticQuadP1;
```

Adding coupling to CMB….

```
ElementAdapter<
  CoupleToAcoustic<
    Elastic2D<
      Quad<
        QuadP1>>>> AtnElasticQuadP1;
```

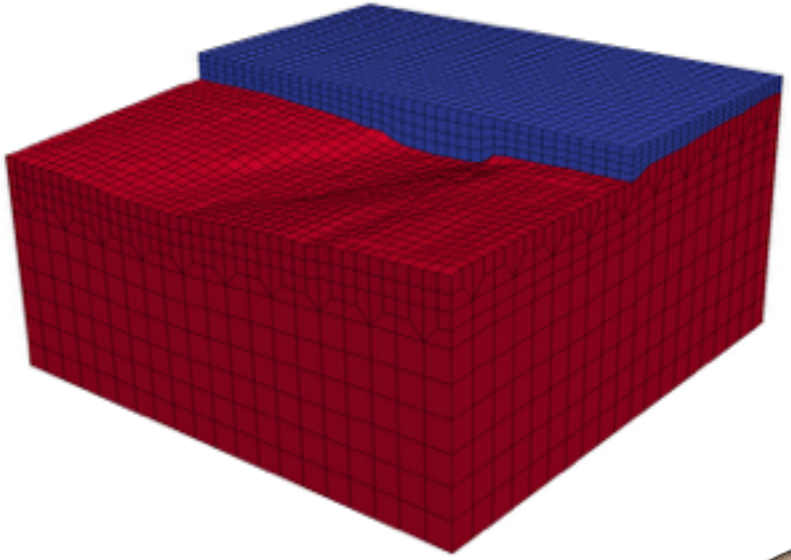
Modular design: a solution with template mixins

All in one element loop…

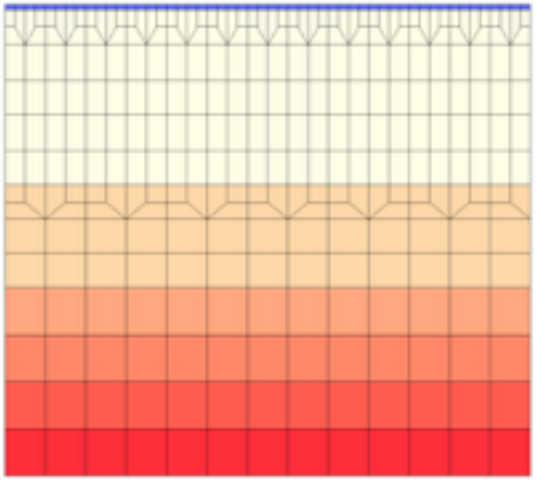
```
// Get values from distributed array.
mesh->pullElementalFields();

// Compute element integrals.
for (auto &elm: elements) {

  // Get relevant values.
  u = mesh->getFields(elm->closure());

  // Compute stiffness.
  ku = elm->computeStiffnessTerm(u);

  // Compute surface integral.
  s = elm->computeSurfaceIntegral(u);

  // Compute source term.
  f = elm->computeSourceTerm(time);

  // Compute acceleration.
  a = f - ku + s

  // Assemble.
  mesh->pushFields(elm->closure());

}

// Push values to distributed array.
mesh->pushElementalFields();
```

Flexible spatial and temporal discretization

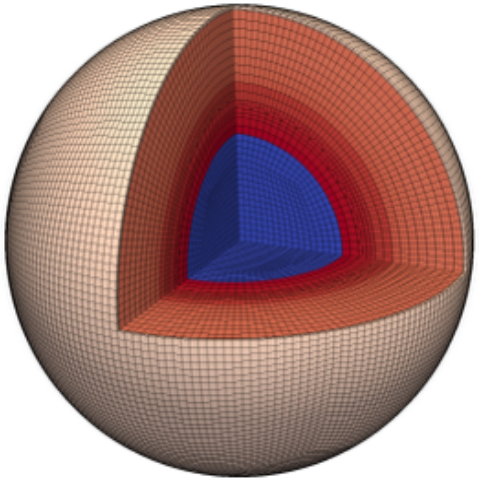
Flexible spatial discretization: Builtin python-based meshing tools



Exploration scale
(with topography)



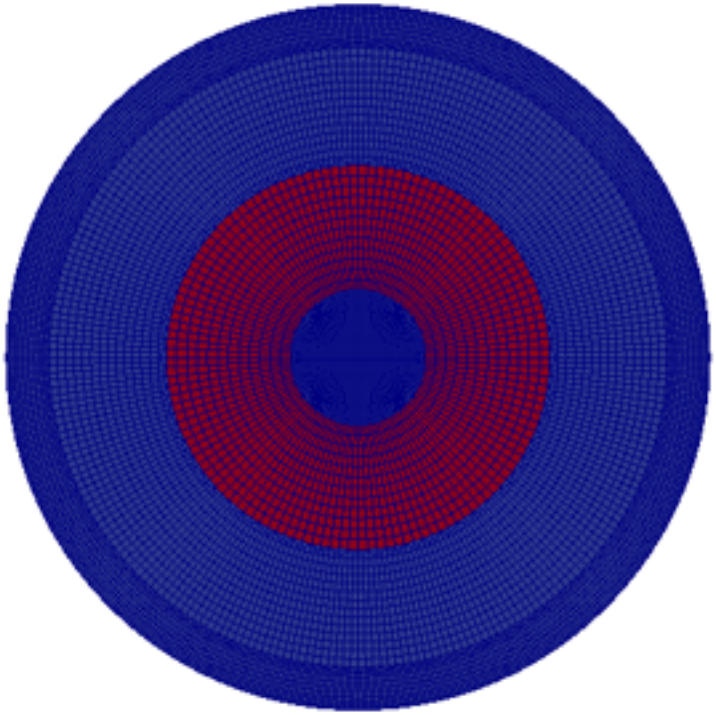
2D applications



Planets (Mars, Europa, …)

Flexible spatial discretization: PETSc DMPLEX

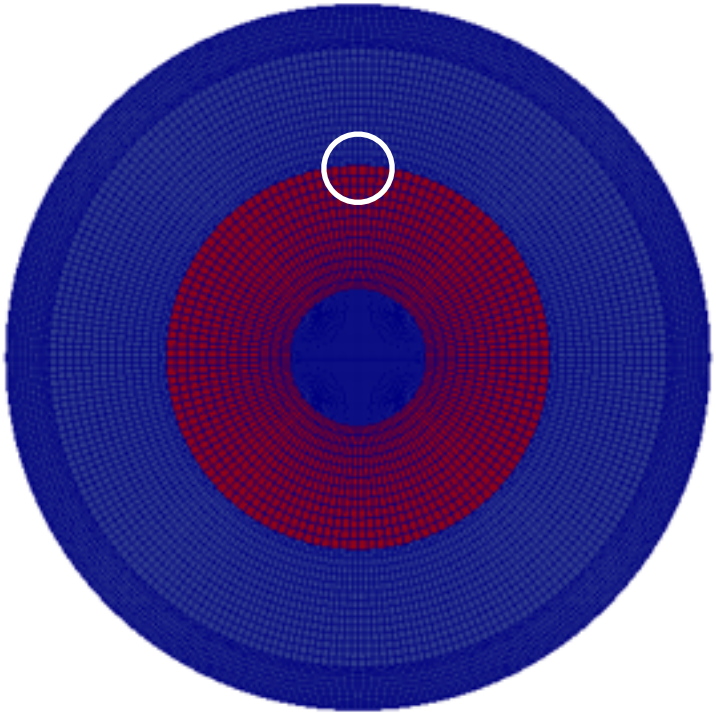
An example with coupling



Red: fluid
Blue: solid

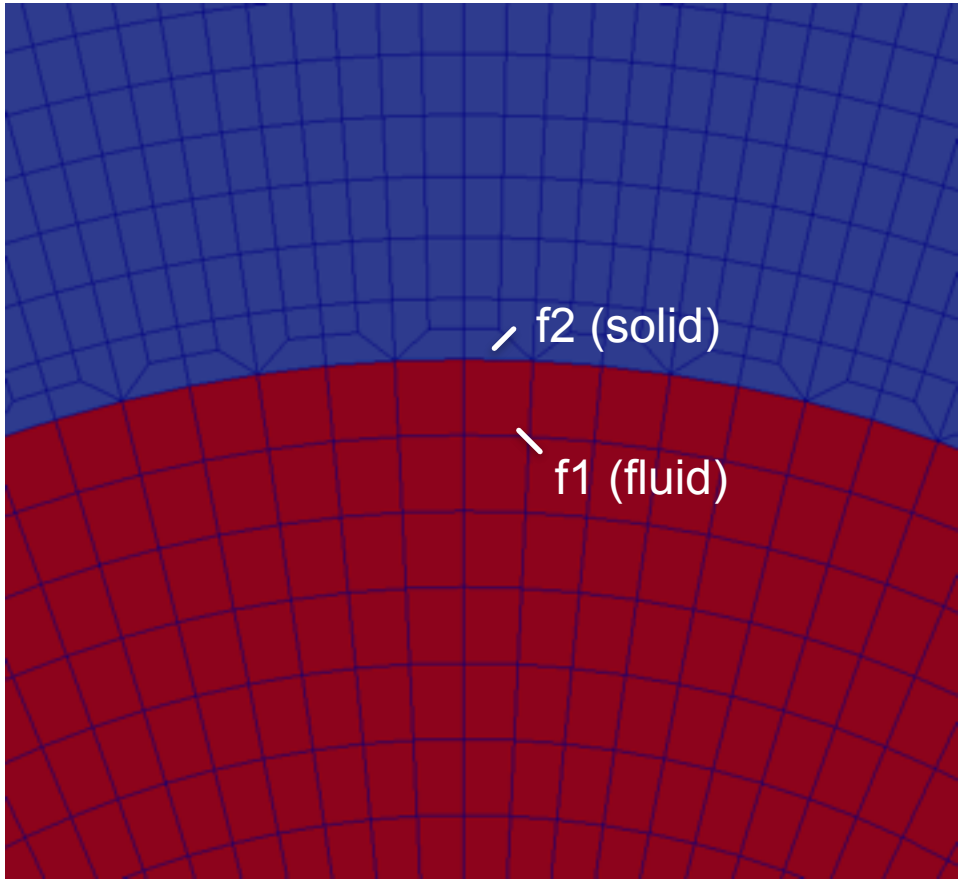
Flexible spatial discretization: PETSc DMPLEX

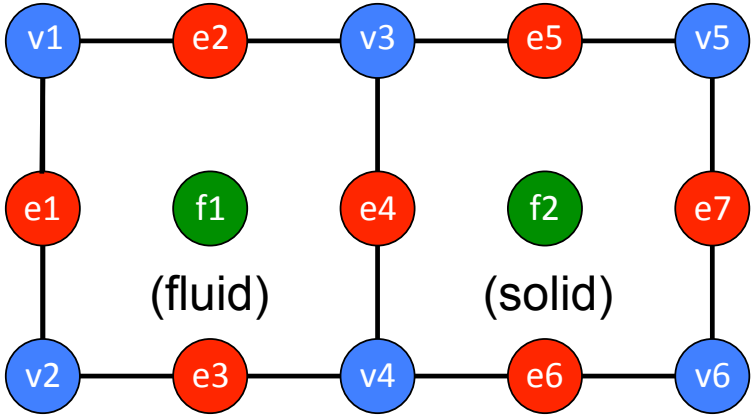
An example with coupling



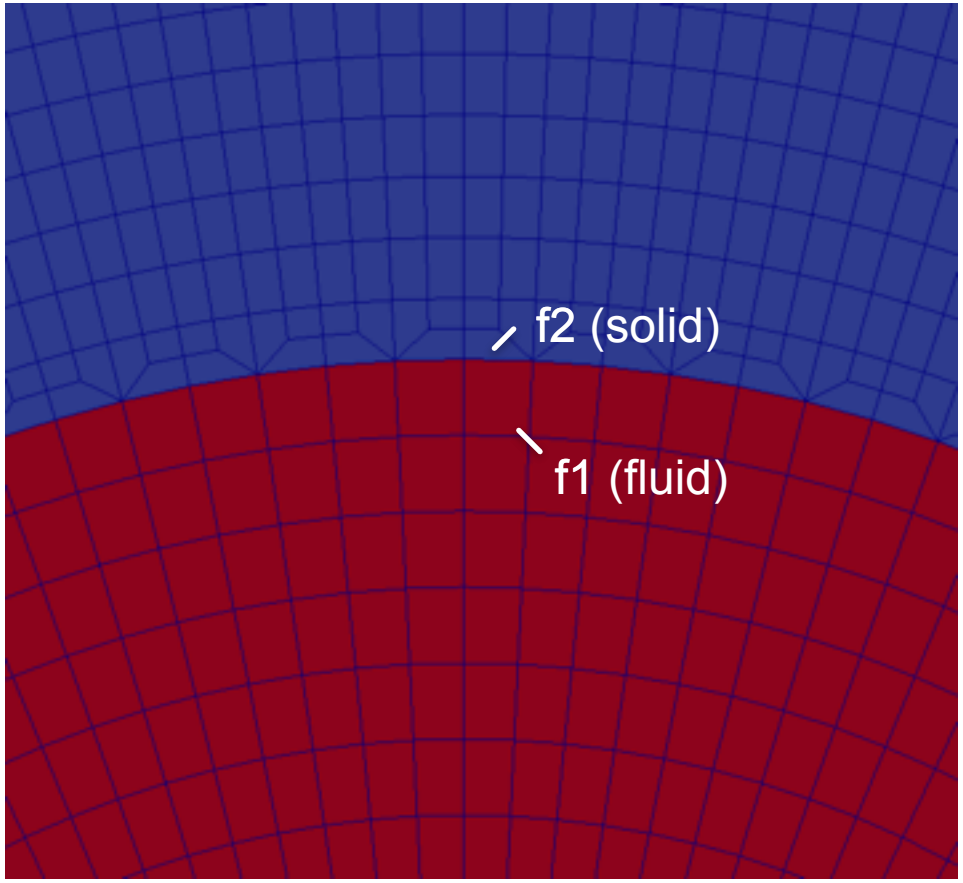
Red: fluid
Blue: solid

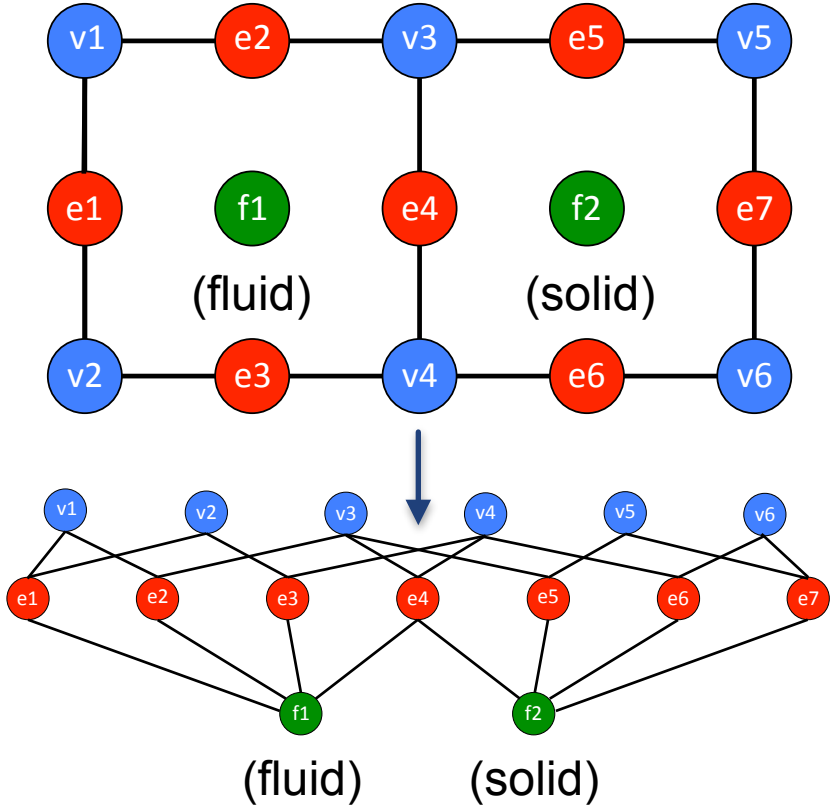
Flexible spatial discretization: PETSc DMPLEX



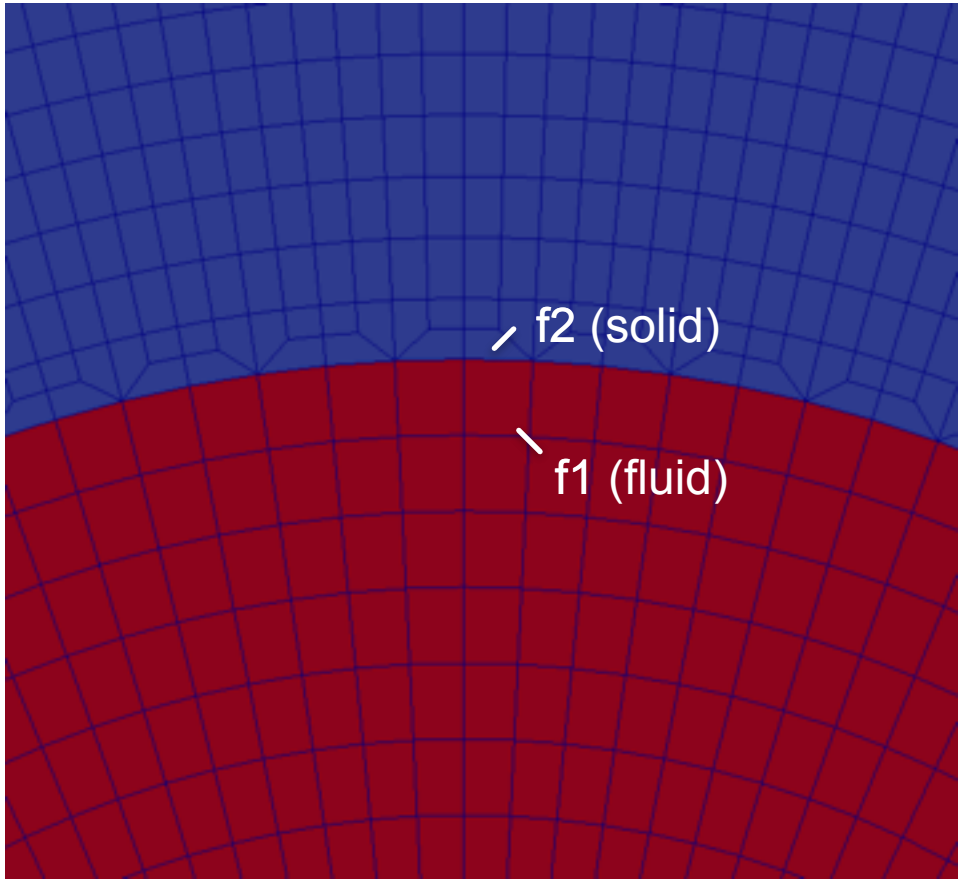


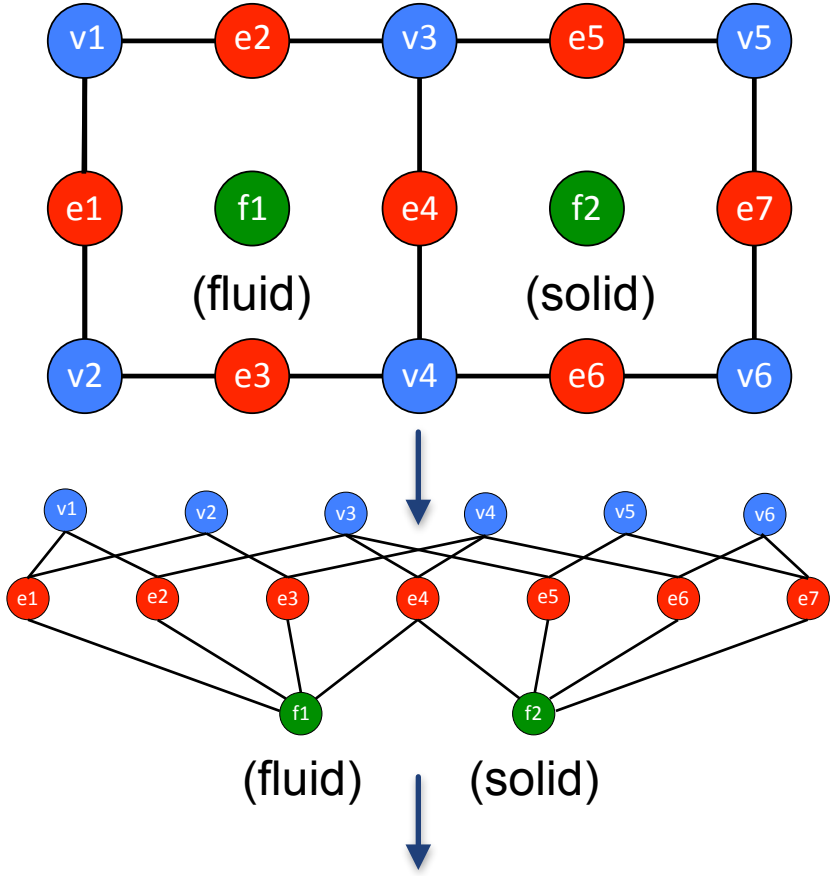
Flexible spatial discretization: PETSc DMPLEX





Flexible spatial discretization: PETSc DMPLEX



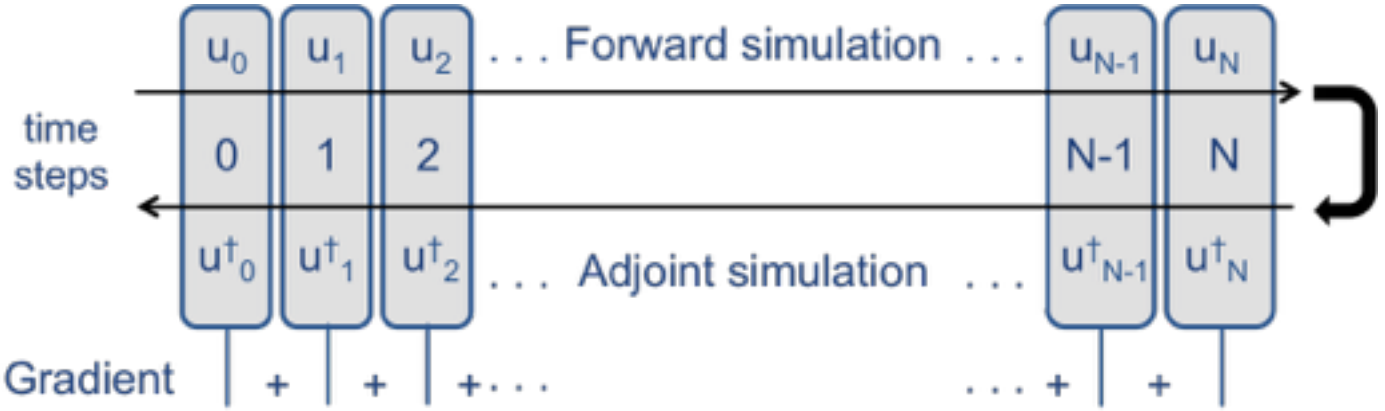



```
element_vector.push_back(
  ElasticCplAcousticQuadP1(options));
```

Integration with external optimization routines (salvus_opt)

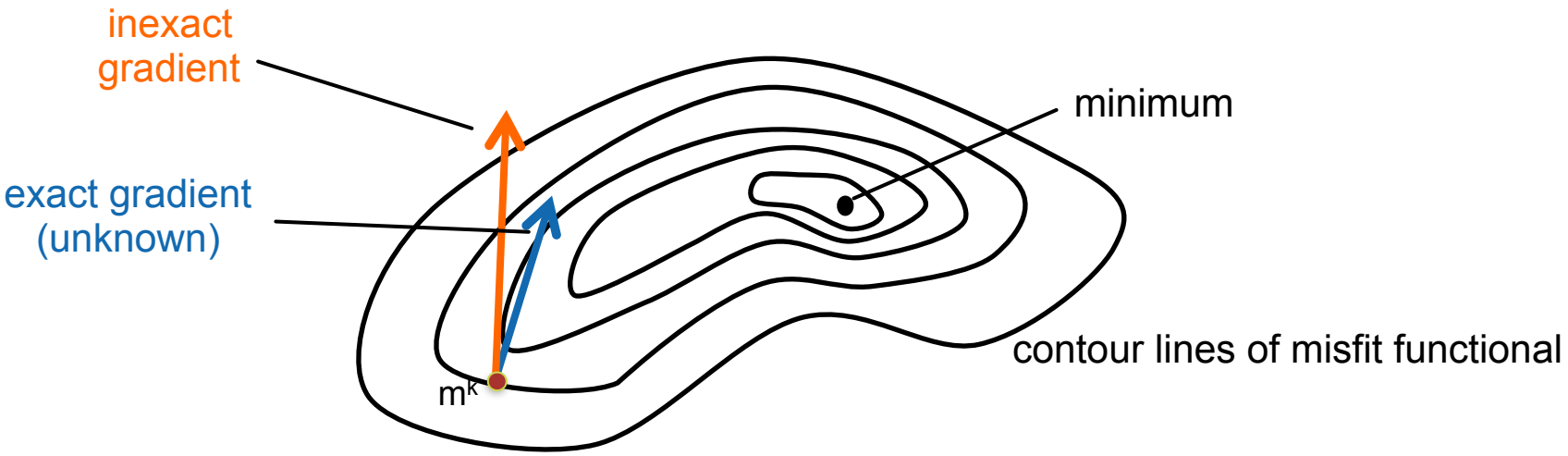
Wavefield Compression

Time-domain full-waveform inversion requires massive storage capabilities to store the forward wavefield.

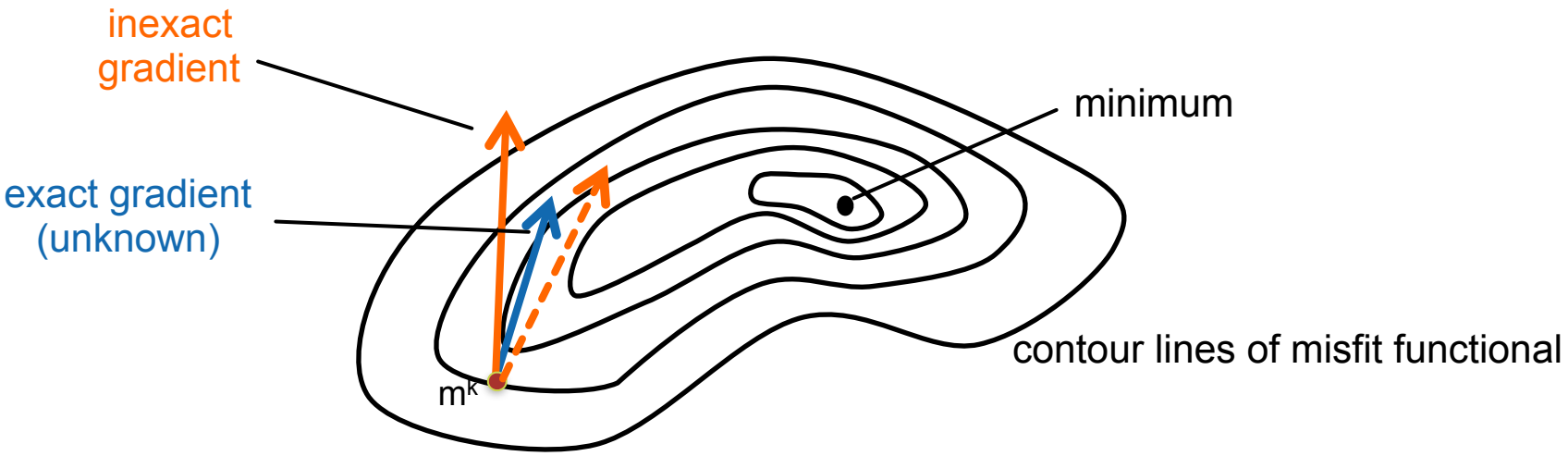


Goal:
Find a good tradeoff between memory requirements and computational overhead by using customized compression methods.

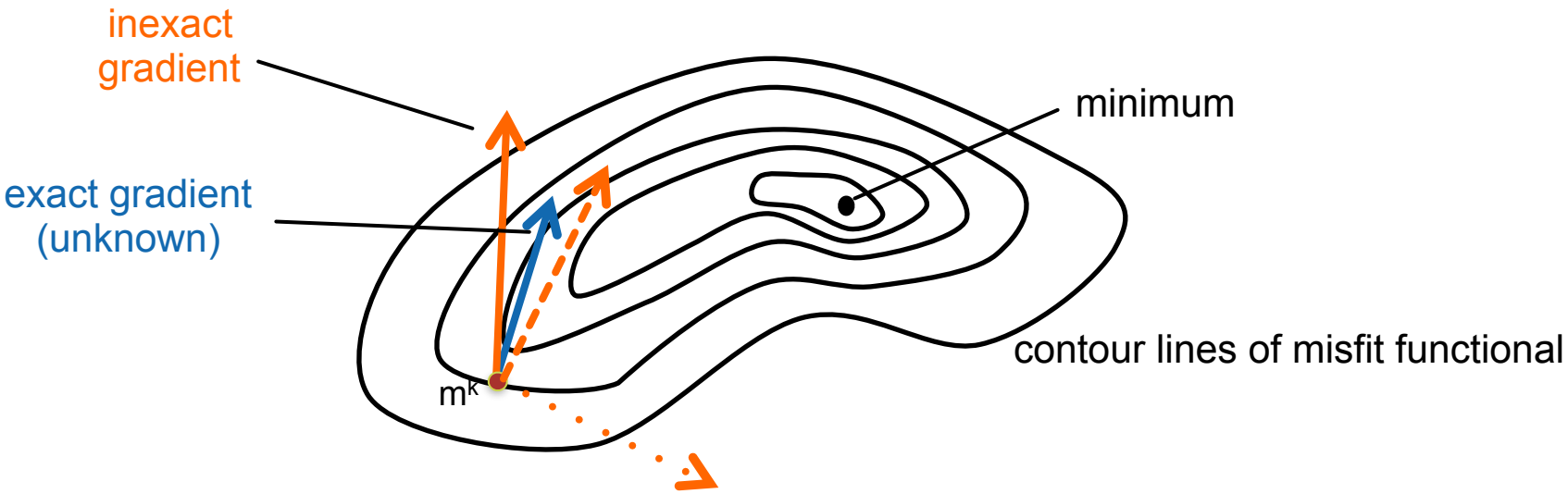
Line-Search with Inexact Gradients



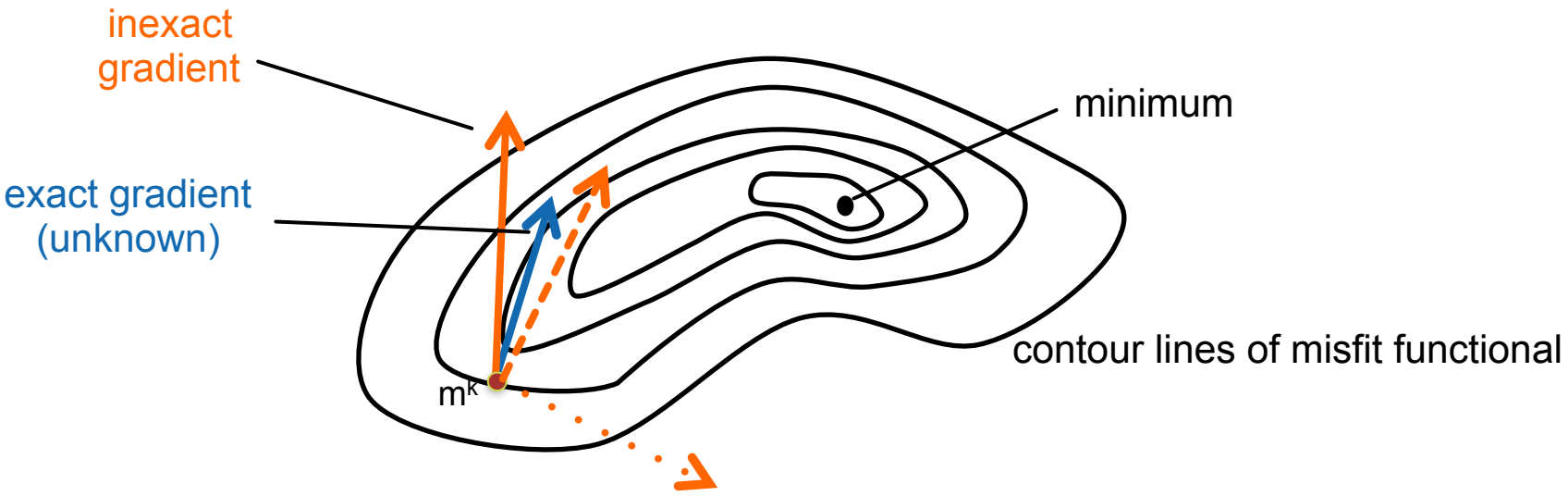
Line-Search with Inexact Gradients



Line-Search with Inexact Gradients



Line-Search with Inexact Gradients



Compression thresholds can be chosen adaptively based on

- the norm of the inexact gradient,
- the ratio of actual and predicted misfit reduction.

Convergence can be ensured if the relative error is smaller than 50%.

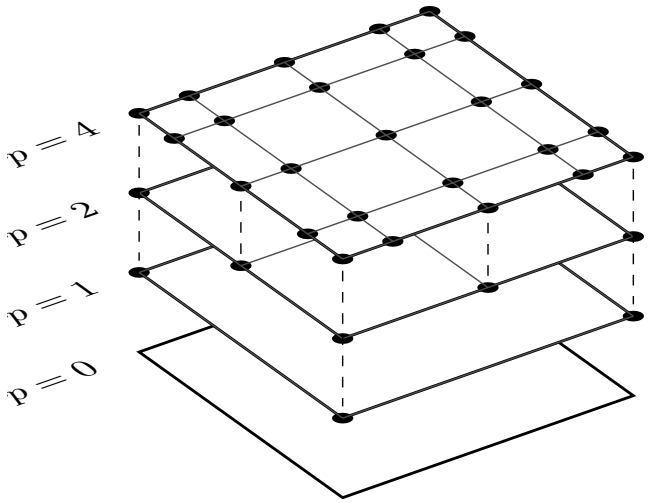
Wavefield Compression

- Substantial reduction of memory requirements and I/O operations at negligible extra costs.
  - 200 TB per event

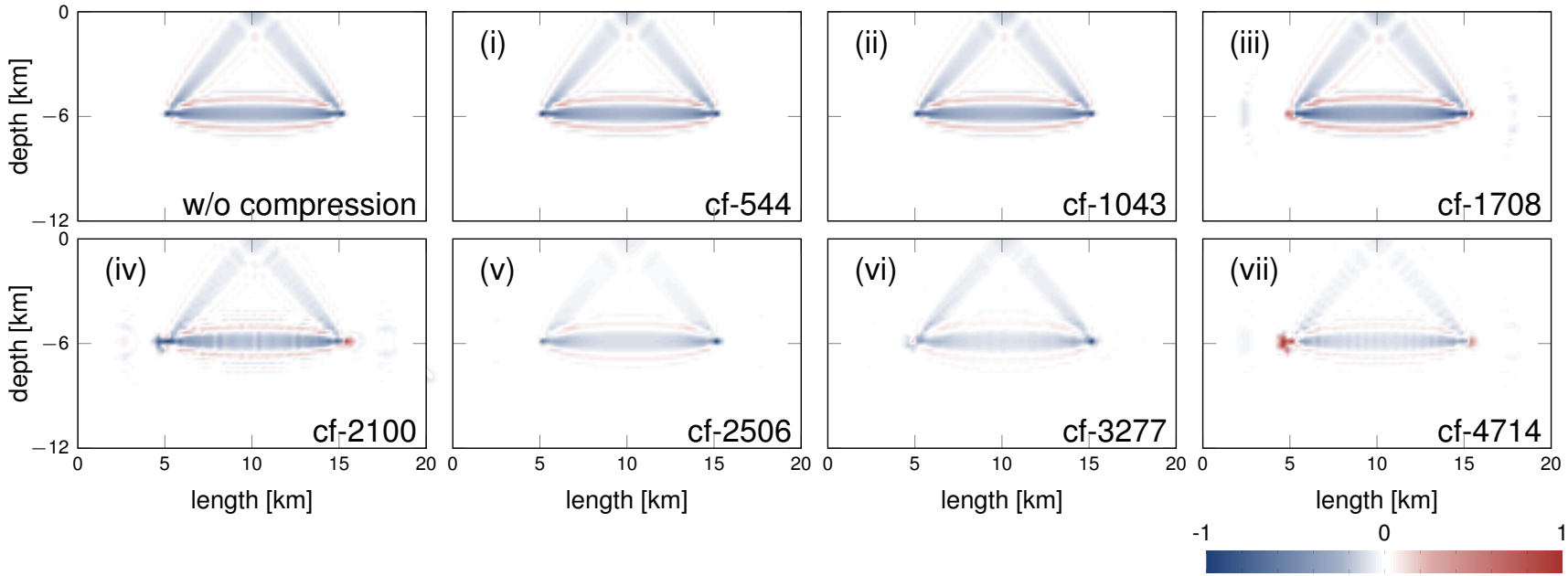
- Using approximate gradients does not significantly slow down the rate of convergence to solve the inverse problem.

- The error in the inexact gradients can be controlled such that the lossy compression does not significantly affect the inverted results.

- Prediction-correction on hierarchical grids
- Requantization
- Re-interpolation with sliding-window cubic splines



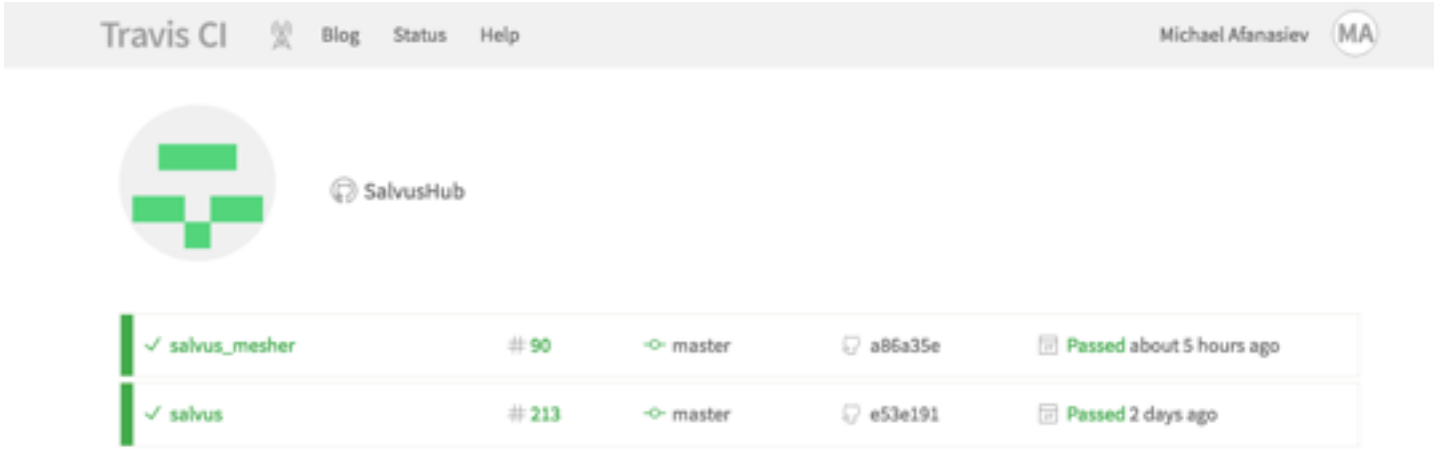
Integration with optimization libraries



- Consistent discrete adjoint equation
- Built-in interface to wavefield compression to reduce memory requirements
- Extensions for Hessian-vector products (currently under development)

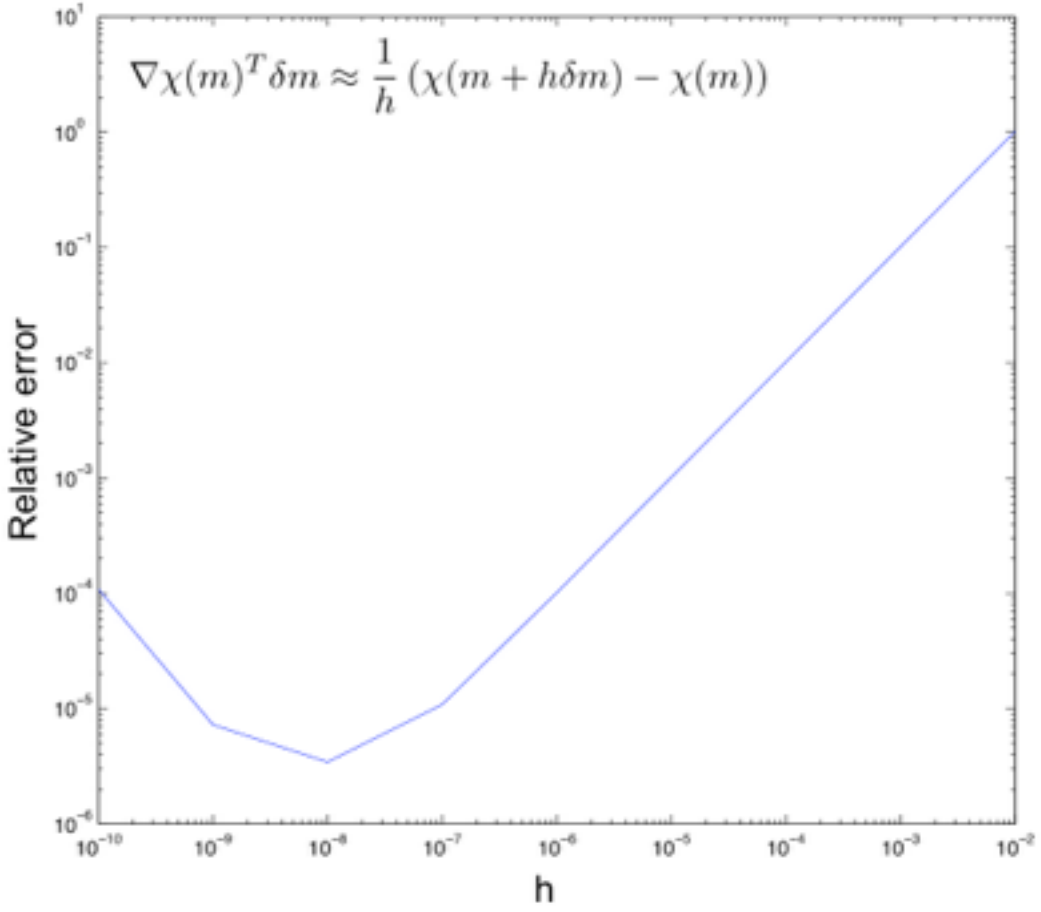
Comprehensive testing suite

Comprehensive Test Suite



- Every contribution is run against a comprehensive test suite, driven by Catch™
  - Analytical integrations on element volumes, faces, edges
  - Analytical time-dependent solutions
  - Proper interpolation of sources/receivers
- Eases collaboration from the level of student to domain specialist

Comprehensive Test Suite



Testing integration with optimization libraries

Speed

Speed

Templates resolved at compile time

```
ElementAdapter<Attenuation<Elastic2D<Quad<QuadP1>>>> AtnElasticQuadP1;
```

	SPECFEM3D CARTESIAN	SALVUS
Stiffness Calculation (microseconds)	1.7	15

Speed

Templates resolved at compile time

```
ElementAdapter<Attenuation<Elastic2D<Quad<QuadP1>>>> AtnElasticQuadP1;
```

	SPECFEM3D CARTESIAN	SALVUS (precomputed Jacobian)
Stiffness Calculation (microseconds)	1.7	5

Speed

Templates resolved at compile time

```
ElementAdapter<Attenuation<Elastic2D<Quad<QuadP1>>>> AtnElasticQuadP1;
```

	SPECFEM3D CARTESIAN	SALVUS (Deville optimized strides)
Stiffness Calculation (microseconds)	1.7	2.2

Speed

Templates resolved at compile time

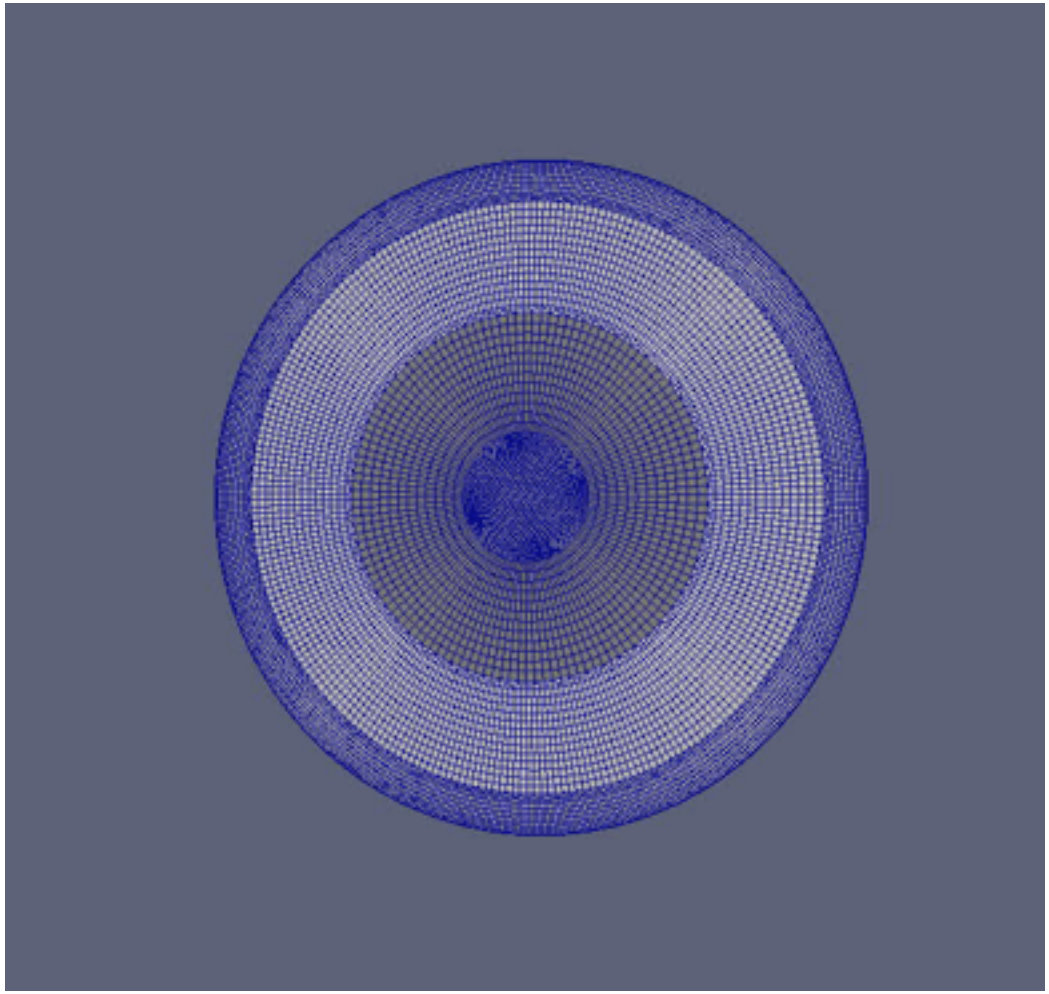
```
ElementAdapter<Attenuation<Elastic2D<Quad<QuadP1>>>> AtnElasticQuadP1;
```

	SPECFEM3D CARTESIAN	SALVUS (Deville optimized strides)
Stiffness Calculation (microseconds)	1.7	2.2

Haven’t really tried yet…

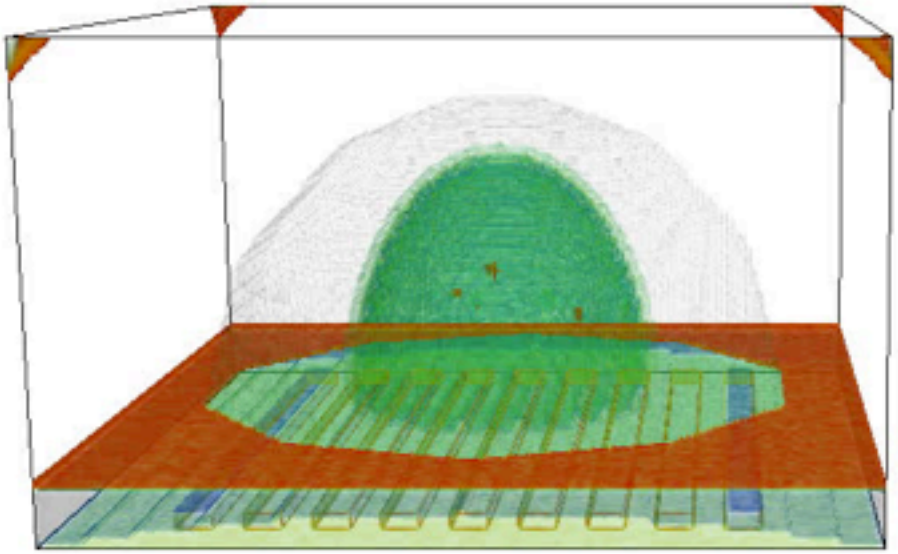
Applications

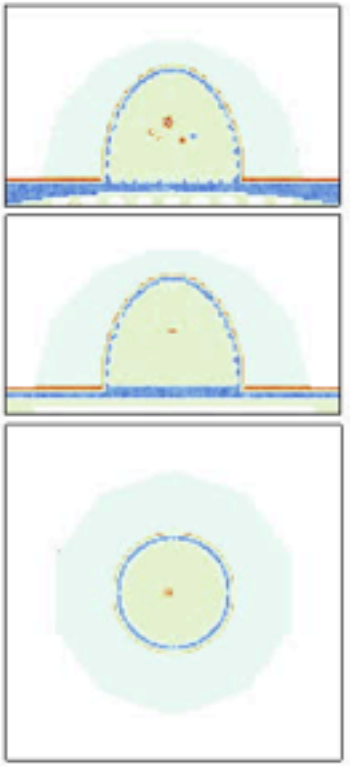
Applications



Applications

Applications





Outlook

Accelerators?

```
#include <stdio.h>
#include <iostream>
#include <cuda.h>

class QuadP1 {
public:

  __device__ __host__ void jacobian() {
    printf("%s\n", "Computing QuadP1 Jacobian.");
  };

};

class TriangleP1 {
public:
  __device __host__ void jacobian() {
    printf("%s\n", "Computing TriangeP1 Jacobian.");
  };
};

template <typename ConcreteElement>
class Quad: public ConcreteElement {

public:
  __device__ __host__ void gradient() {
    printf("%s\n", "Computing quad gradient.");
    ConcreteElement::jacobian();
  };

};

template <typename ConcreteElement>
class Tri: public ConcreteElement
public:
  __device__ __host__ void gradient() {
    printf("%s\n", "Computing triangle gradient.");
    ConcreteElement::jacobian();
  };
};
```

Accelerators?

```
#include <stdio.h>
#include <iostream>
#include <cuda.h>

class QuadP1 {
public:

  __device__ __host__ void jacobian() {
    printf("%s\n", "Computing QuadP1 Jacobian.");
  };

};

class TriangleP1 {
public:
  __device__ __host__ void jacobian() {
    printf("%s\n", "Computing TriangeP1 Jacobian.");
  };
};

template <typename ConcreteElement>
class Quad: public ConcreteElement {

public:
  __device__ __host__ void gradient() {
    printf("%s\n", "Computing quad gradient.");
    ConcreteElement::jacobian();
  };

};

template <typename ConcreteElement>
class Tri: public ConcreteElement
public:
  __device__ __host__ void gradient() {
    printf("%s\n", "Computing triangle gradient.");
    ConcreteElement::jacobian();
  };
};
```

Graphics cards? Knights Landing?

```
// Get values from distributed array.
mesh->pullElementalFields();

// Compute element integrals.
for (auto &elm: elements) {

  // Get relevant values.
  u = mesh->getFields(elm->closure());

  // Compute stiffness.
  ku = elm->computeStiffnessTerm(u);

  // Compute surface integral.
  s = elm->computeSurfaceIntegral(u);

  // Compute source term.
  f = elm->computeSourceTerm(time);

  // Compute acceleration.
  a = f - ku + s

  // Assemble.
  mesh->pushFields(elm->closure());

}

// Push values to distributed array.
mesh->pushElementalFields();
```

Outlook

- Frequency domain (billions of dofs…)
- Bridge the gap between research and production codes
- Nonconforming meshes
- Applications of FWI to new and interesting domains
- Additional physics (GPR, electromagnetic, …)

Very warm thanks to:

- Kuangdai Leng (University of Oxford)
- Dan Chown (Achates power)
- Matt Knepley (Rice university)
- Tarje Nissen-Meyer (University of Oxford)

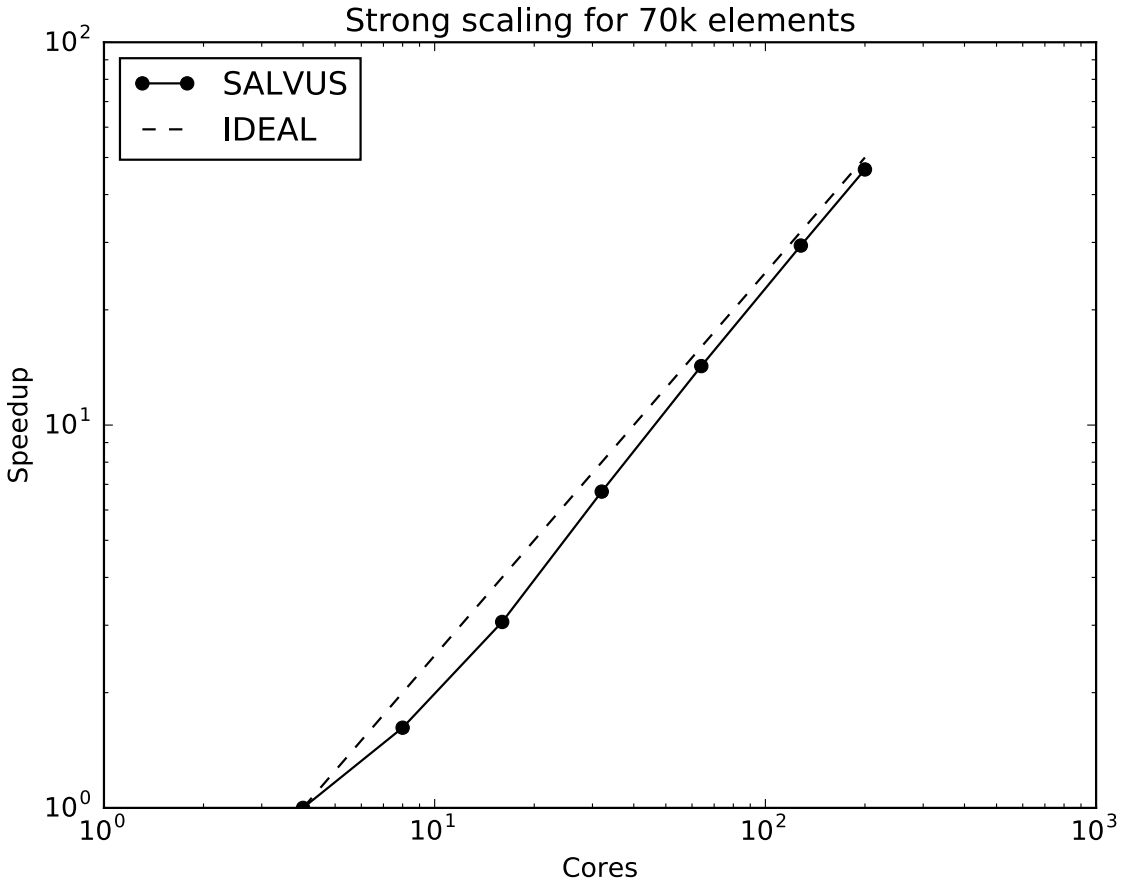
Features and Methods

- trust-region and line search methods
- reusing pre-computed information whenever possible
  -> interpolation instead of backtracking
  -> book-keeping of Earth models
- built-in regularization and smoothing
- constraint handling using projection methods and homogenization
  -> constraints on m are cheap, “feasible-point methods”
- handling of inexact derivatives and change of objective function

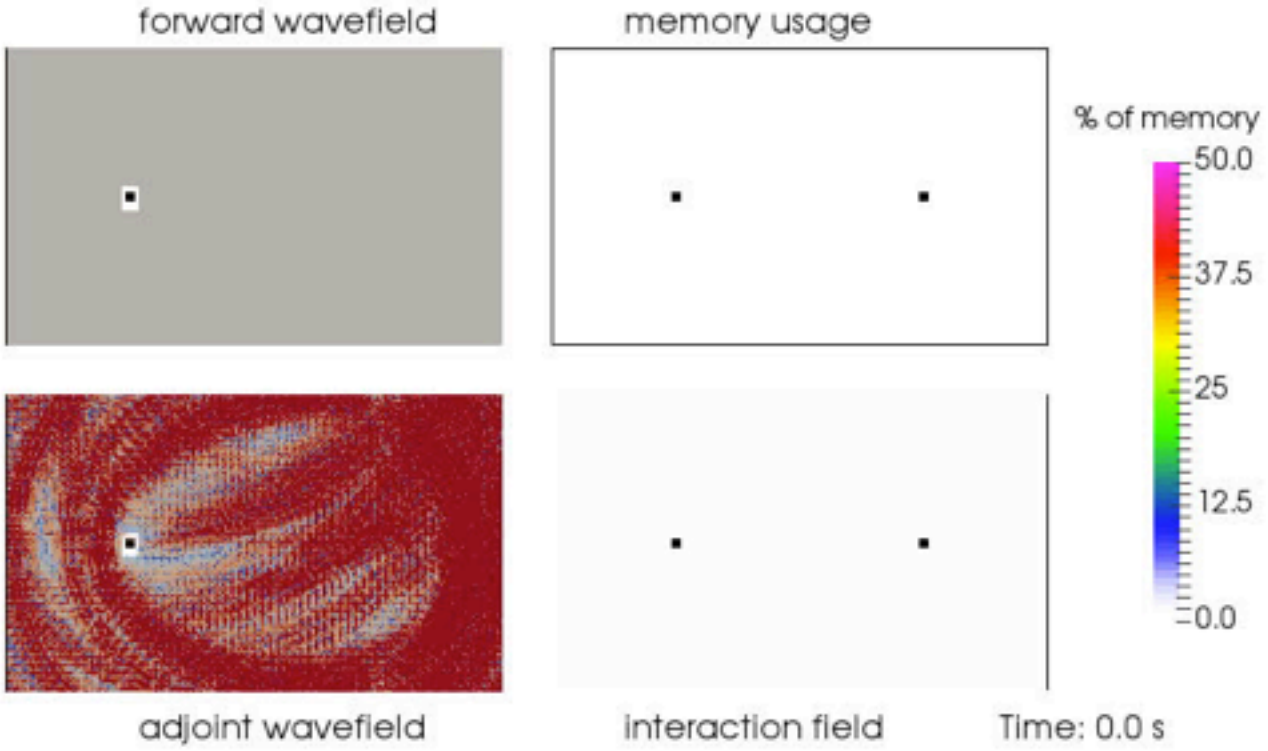
Speed: Scaling

Maintain the scaling characteristics of PETSc

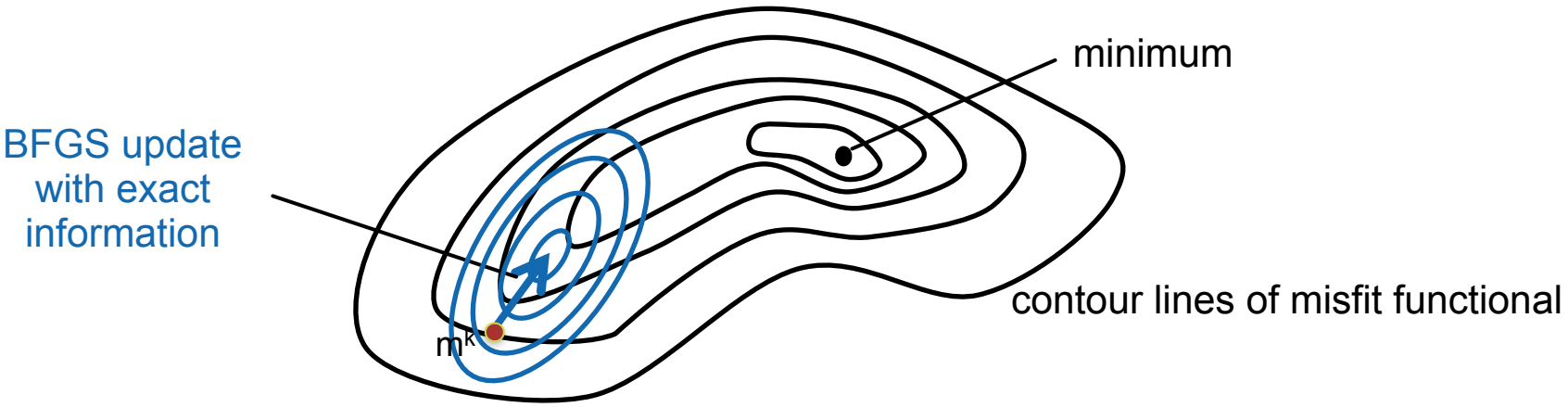
Parallelization via PETSc is essentially free



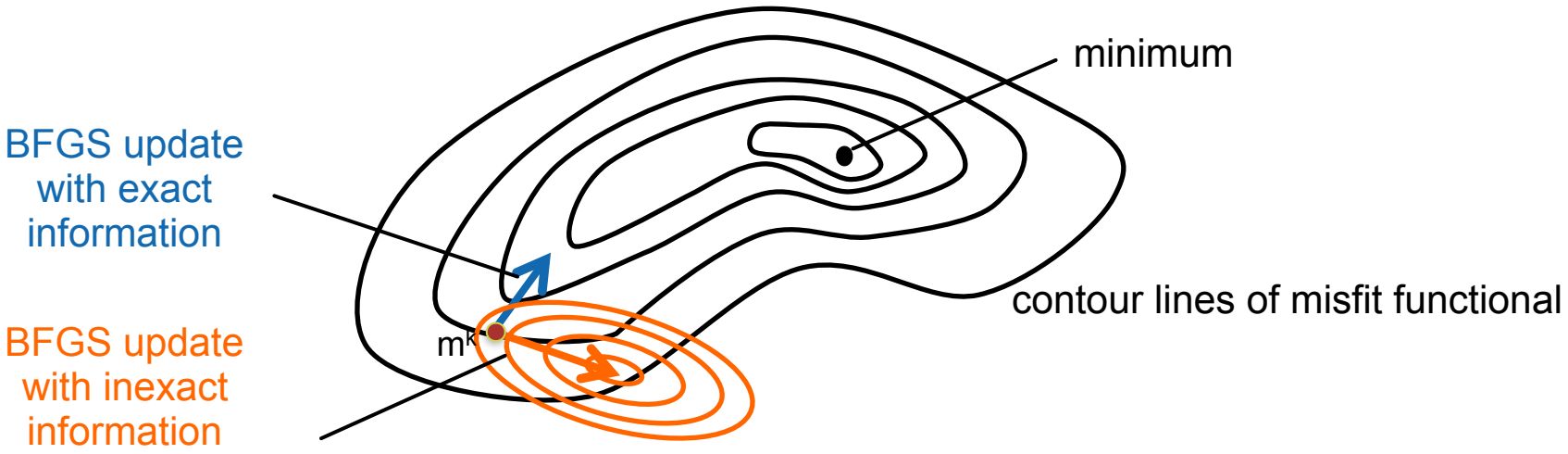
Wavefield Compression



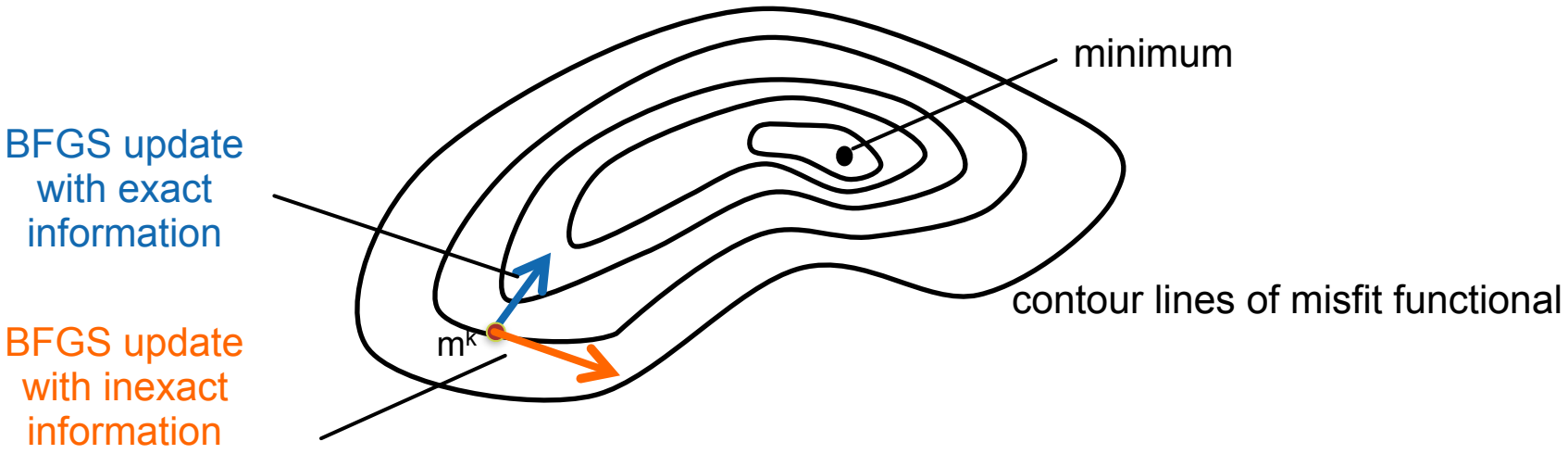
Line-Search with Inexact Gradients



Line-Search with Inexact Gradients



Line-Search with Inexact Gradients



Classical workaround for Newton-type methods:

$$(H_k + \alpha I)s^k = -\tilde{g}^k$$

Coupling to other physics…

```
#include <Model/ExodusModel.h>
#include <Physics/AcousticElastic2D.h>
#include <Utilities/Options.h>
#include <Mesh/Mesh.h>

using namespace Eigen;

template <typename BasePhysics>
std::vector<std::string> AcousticToElastic2D<BasePhysics>::PullElementalFields() const {
  return {"ux", "uy", "v"};
}

template <typename BasePhysics>
void AcousticToElastic2D<BasePhysics>::setBoundaryConditions(Mesh *mesh) {
  for (auto e: mesh->CouplingFields(BasePhysics::ElmNum())) {
    mEdg.push_back(std::get<0>(e));
    mNbr.push_back(mesh->GetNeighbouringElement(mEdg.back(), BasePhysics::ElmNum()));
    mNbrCtr.push_back(mesh->getElementCoordinateClosure(mNbr.back()).colwise().mean());
  }
  BasePhysics::setBoundaryConditions(mesh);
}

template <typename BasePhysics>
Eigen::MatrixXd AcousticToElastic2D<BasePhysics>::computeSurfaceIntegral(const Eigen::Ref<const
Eigen::MatrixXd> &u) {

  // col0->ux, col1->uy, col2->potential.
  Eigen::MatrixXd rval = Eigen::MatrixXd::Zero(BasePhysics::NumIntPnt(), 2);
  for (int i = 0; i < mEdg.size(); i++) {
    rval.col(0) += mRho_0[i] * BasePhysics::applyTestAndIntegrateEdge(u.col(2), mEdg[i]);
    rval.col(1) += mRho_0[i] * BasePhysics::applyTestAndIntegrateEdge(u.col(2), mEdg[i]);
  }

  return -1 * rval;

}
```

Modular design: a solution with template mixins

```
template <typename Element>
MatrixXd Elastic2D<Element>::computeStress(const Eigen::Ref<const Eigen::MatrixXd>
&strain) {

  mc11 = Element::ParAtIntPts("C11");
  mc12 = Element::ParAtIntPts("C12");
  mc22 = Element::ParAtIntPts("C22");
  mc33 = Element::ParAtIntPts("C33");

  Matrix<double,Dynamic,3> stress(Element::NumIntPnt(), 3);
  VectorXd uxy_plus_uyx = strain.col(1) + strain.col(2);

  stress.col(0) =
    mc11 * strain.col(0) + mc12 * strain.col(3) + mc13 * uxy_plus_uyx);

  stress.col(1) =
   mc12 * strain.col(0) + mc22 * strain.col(3) + mc23 * uxy_plus_uyx);

  stress.col(2) =
    mc13 * strain.col(0) + mc23 * strain.col(3) + mc33 * uxy_plus_uyx);

  return stress;

}
```