PDE-Constrained Optimisation with Mesh-Independent Convergence in Dolfin-Adjoint

Simon W. Funke
M. Nordaas, P. E. Farrell, J. Young

Simula Research Laboratory

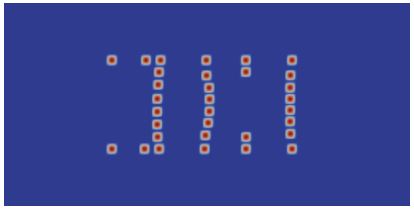
17th March 2015

Overview

1. Introduction
2. A high-level framework for PDE-constrained optimisation
3. How to achieve mesh-independent convergence
4. Conclusions

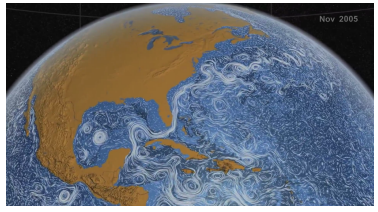
Some applications of PDE-constrained optimisation

Optimal control



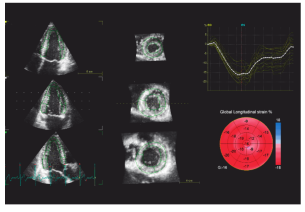
Optimal configuration of tidal stream turbines

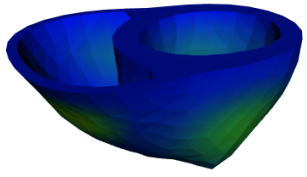
Data assimilation



Assimilating measurements into a global ocean model (ECCO2 project)

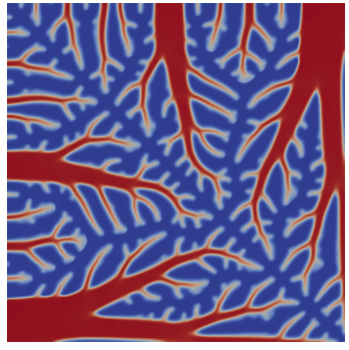
Parameter estimation





Determining unkown parameters in a cardiac model (G. Balaban)

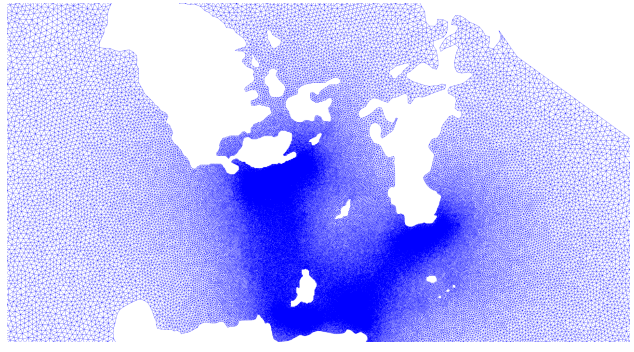
Shape/topology optimisation



Optimal design of a heat sink

Challenges in PDE-constrained optimisation

- Evaluation of objective and constraint functions require the solution of PDEs
  $\rightarrow$ efficient PDE and optimisation solvers
- Infinite or high-dimensional controls
  $\rightarrow$ derivative based optimisation and adjoint model
- Approximation of an infinit-dimensional problem
  $\rightarrow$ robust solvers against mesh-refinement



The robust and efficient solution of PDE-constrained optimisation problems requires the interplay of many components

A high-level framework for PDE-constrained optimisation

Add dolfin-adjoint and FEniCs logo
An introduction by example

The resulting code is compact and close to mathmatics

Optimal control of the Poisson equation

$$\min_{u,m} \frac{1}{2}\|u - u_d\|_{L^2}^2 + \frac{\alpha}{2}\|f\|_{L^2}^2$$

$$\text{subject to} -\nabla^2 u = f \quad \text{on } \Omega,$$
$$u = 0 \quad \text{on } \partial\Omega.$$

Discrete problem

$$\min_{m} \frac{1}{2}\|u(f) - u_d\|_{L^2}^2 + \frac{\alpha}{2}\|f\|_{L^2}^2$$

where $u(f)$ solves: Given f, find $u \in V_0$ such that

$$\int_\Omega \nabla u \cdot \nabla v - fv \mathrm{d}x = 0$$

```
from dolfin import *
from dolfin_adjoint import *

mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
v = TestFunction(V)
u = Function(V)

F = inner(grad(u), grad(v))*dx - f*v*dx
bc = DirichletBC(V, 0.0, "on_boundary")
solve(F == 0, u, bc)

J = Functional(0.5*(u-u_d)**2*dx + 0.5*alpha*f**2*dx)
m = Control(f)
Jhat = ReducedFunctional(J, m)
minimize(Jhat, method="Newton-CG")
```

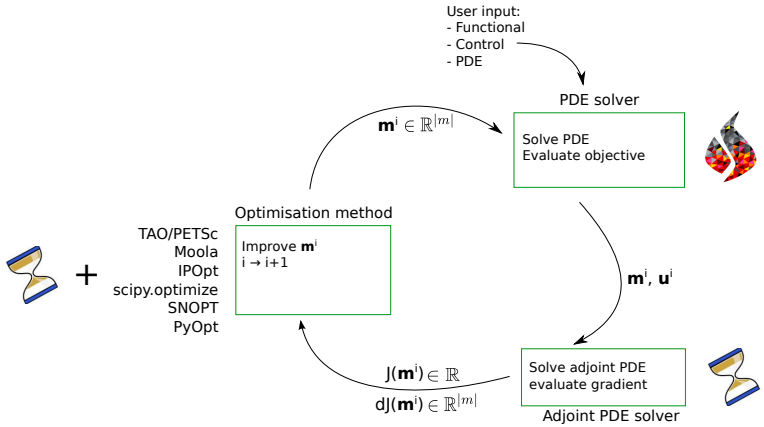
FEniCS combines generality and efficiency through code generation

```
from dolfin import *
from dolfin_adjoint import *

mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
v = TestFunction(V)
u = Function(V)

F = inner(grad(u), grad(v))*dx - f*v*dx
bc = DirichletBC(V, 0.0, "on_boundary")
solve(F == 0, u, bc)

J = Functional(0.5*(u-u_d)**2*dx
             + 0.5*alpha*f**2*dx)
m = Control(f)
Jhat = ReducedFunctional(J, m)
minimize(Jhat, method="Newton-CG")
```

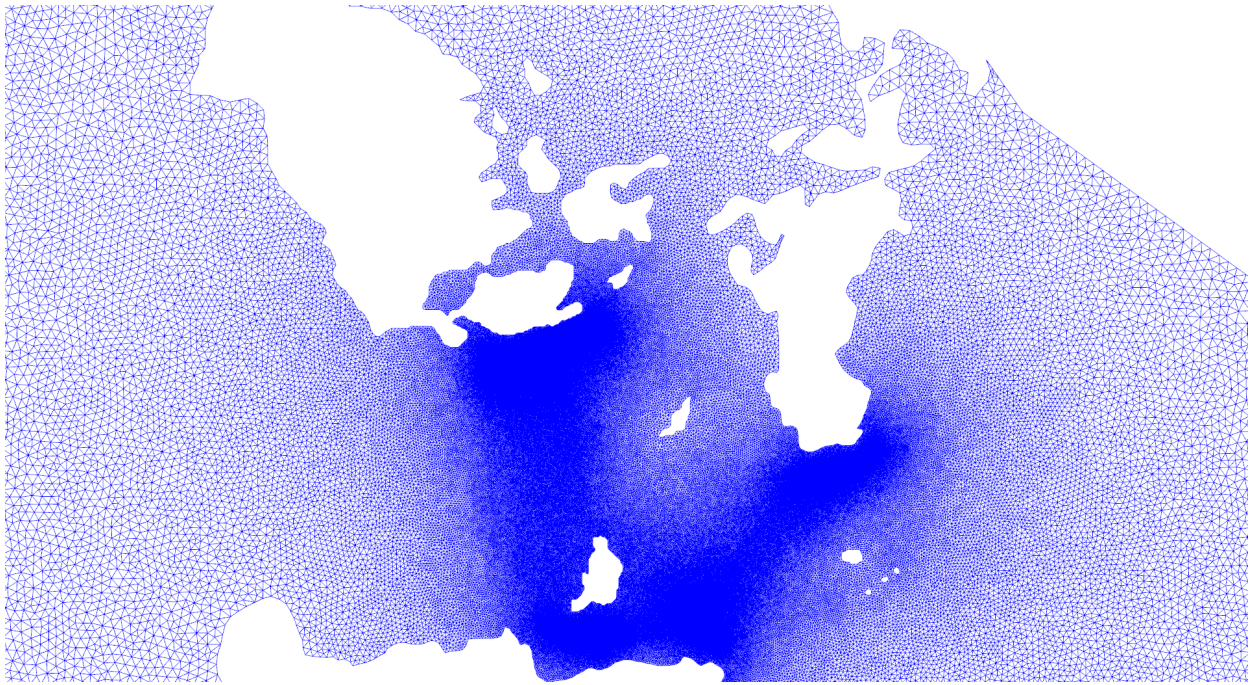
Add a figure on FEniCS

dolfin-adjoint automatically derives the adjoint and interfaces to various optimisation libraries

```
from dolfin import *
from dolfin_adjoint import *

mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
v = TestFunction(V)
u = Function(V)

F = inner(grad(u), grad(v))*dx - f*v*dx
bc = DirichletBC(V, 0.0, "on_boundary")
solve(F == 0, u, bc)

J = Functional(0.5*(u-u_d)**2*dx
               + 0.5*alpha*f**2*dx)
m = Control(f)
Jhat = ReducedFunctional(J, m)
minimize(Jhat, method="Newton-CG")
```



Achieving mesh-independent convergence



Discrete optimisations yields mesh-dependent convergence

Optimal control of the heat equation with successive mesh refinement

Limited-memory BFGS (PETSc/TAO)

Refinement level	0	1	2	3	4	5	6
BFGS iterations	11	39	58	79	80	128	132

Newton-CG (TAO)

Refinement level	0	1	2	3	4	5	6
Newton iterations	1	1	1	1	1	1	1
CG iterations	13	28	38	55	65	95	131

Similar behaviour with scipy.optimize, PyOpt, SNOPT, ...

Infinite dimensional optimisation problem

- Consider the reduced optimisation problem (with no additional constraints)

$$\min_{m \in \mathcal{M}} \hat{J}(m)$$

- After discretization

$$\min_{m \in \mathcal{M}^h} \hat{J}(m_h)$$

  for a finite dimensional $\mathcal{M}^h \in \mathcal{M}$

- Select a basis $\phi_1, ..., \phi_n$ of $\mathcal{M}^h$. We can write $m = \sum_i^n m_i \phi_i$. Then the optimisation problem can be considered in $\mathbf{m} = (m_1, ..., m_n)^T \in \mathbb{R}^n$ but with an inner product to a weighted Euclidean product

$$\langle m_1, m_2 \rangle_{\mathcal{M}} = \mathbf{m}_1^T T \mathbf{m}_2$$

  with positive definite matrix $T \in \mathbb{R}^{n \times n}$.

The optimisation algorithm should use the weighted Euclidian product, but most implementations assume a standard Euclidean product.

Gradient computation

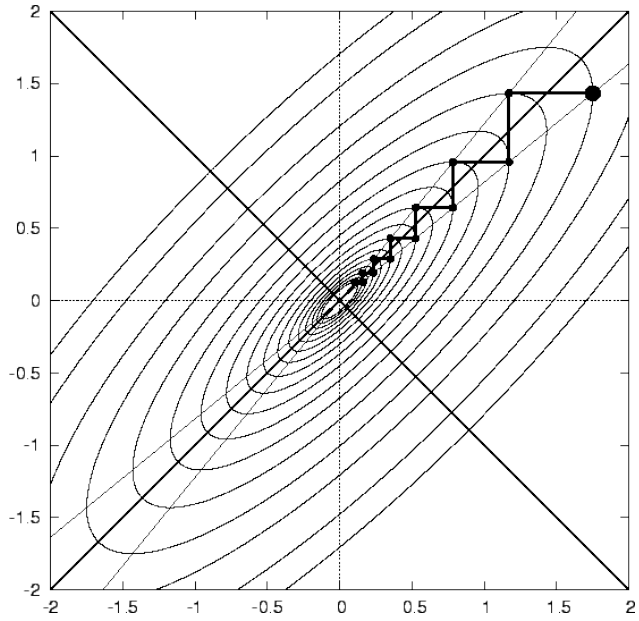
Let $\hat{J} : \mathbb{R}^n \to \mathbb{R}$. Denote the derivative of $\hat{J}$ by $\mathrm{d}\hat{J}$.
The gradient $\nabla\hat{J}(m)$ is defined as the vector for which

$$\left\langle \nabla\hat{J}, m' \right\rangle = D\hat{J}(m)m' \quad \forall m' \tag{1}$$

$\to \nabla\hat{J}(m)$ *depends on the inner product!*
It is important to analyse the optimisation algorithm for the infinite dimensional problem and to apply it properly to the discretised problem

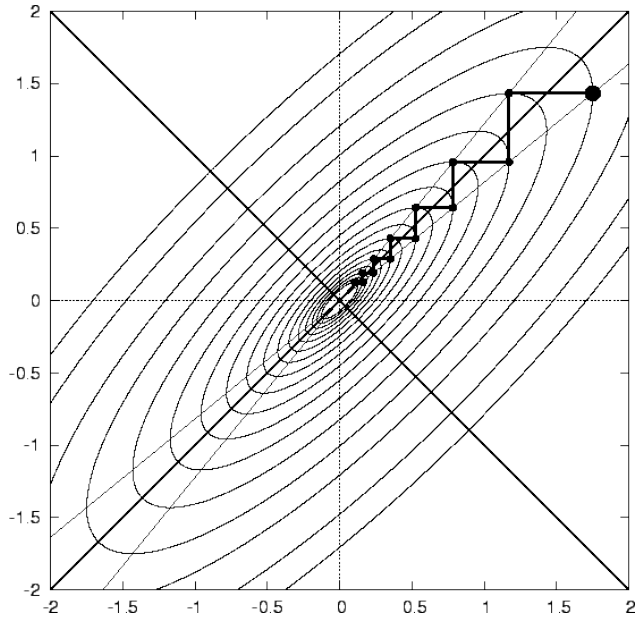
The textbook steepest descent algorithm

$i \leftarrow 0$
while $|dJ(m^i)| < \mathrm{tol}$:
$m^{i+1} \leftarrow m^i - \gamma dJ(m^i)$
$i \leftarrow i + 1$



The textbook steepest descent algorithm

$i \leftarrow 0$
while $|dJ(m^i)| < \mathrm{tol}$:
$m^{i+1} \leftarrow m^i - \gamma dJ(m^i)$
$i \leftarrow i + 1$

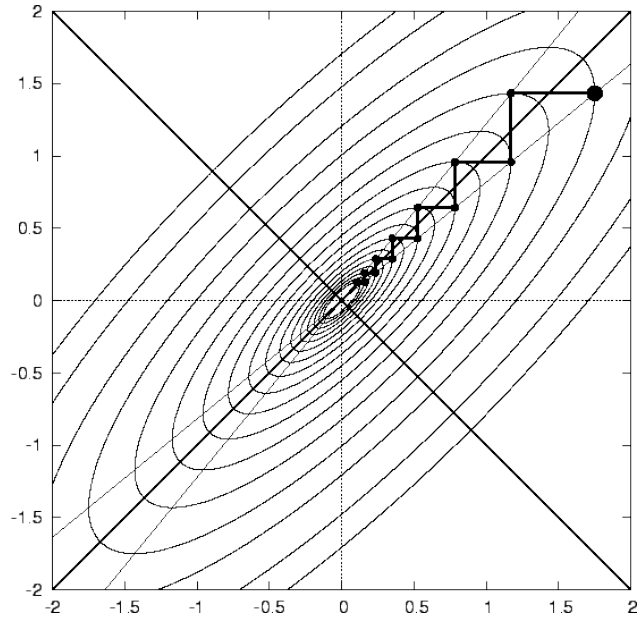


But:

- $m^i \in W$
- $J : W \to \mathbb{R}$
- $dJ(m^i) : W \to \mathbb{R} \in W^*$

Steepest descent algorithm in Hilbert space setting

$i \leftarrow 0$
while $\|dJ(m^i)\|_{W^*} < \mathrm{tol}$:
$m^{i+1} \leftarrow m^i - \gamma \nabla dJ(m^i)$
$i \leftarrow i + 1$

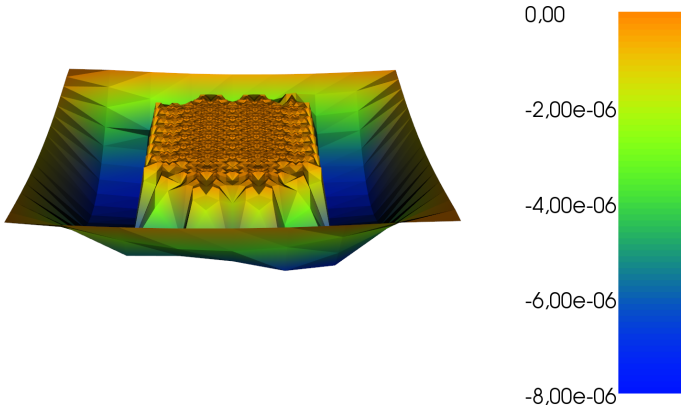


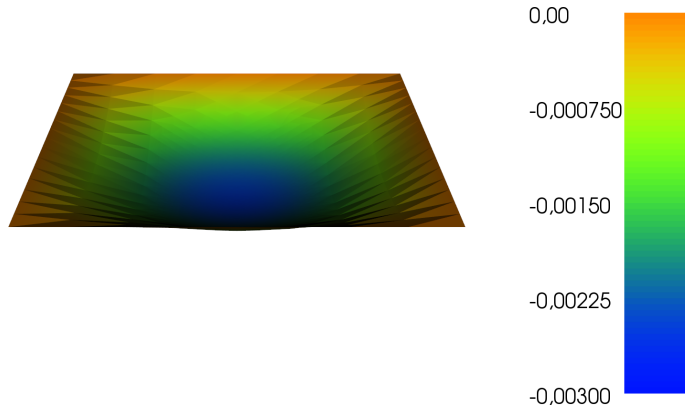
- $m^i \in W$
- $J : W \to \mathbb{R}$
- $dJ(m^i) : W \to \mathbb{R} \in W^*$
- $\nabla J(m^i) \in W$

With Riesz isomoprohism $R : W^* \to W$ the gradient is defined as $\nabla J(m) = R(dJ(m))$, i.e.

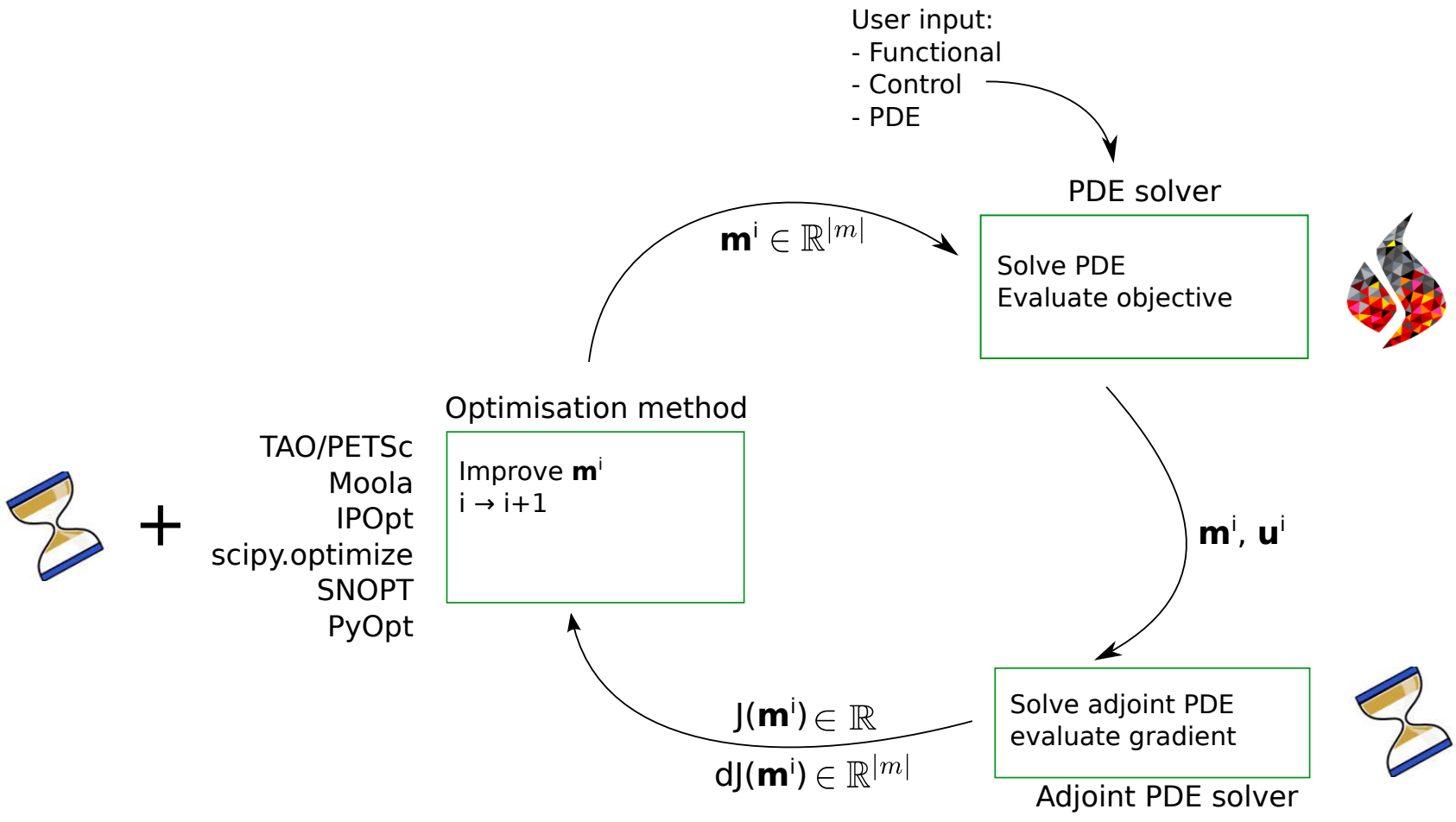
$$\langle dJ(m), \delta m \rangle_{W^*,W} = (\nabla f(m), \delta m)_W$$

Gradient and derivative operator are, and should be treated as different objects





FEniCS/dolfin-adjoint allows to automated most steps of the optimisation



In this talk we use the optimal heating problem as a model problem

$$\min_{u,m} \frac{1}{2}\|u - d\|^2_{L^2(\Omega)} + \frac{\alpha}{2}\|m\|^2_{L^2(\Omega)}$$

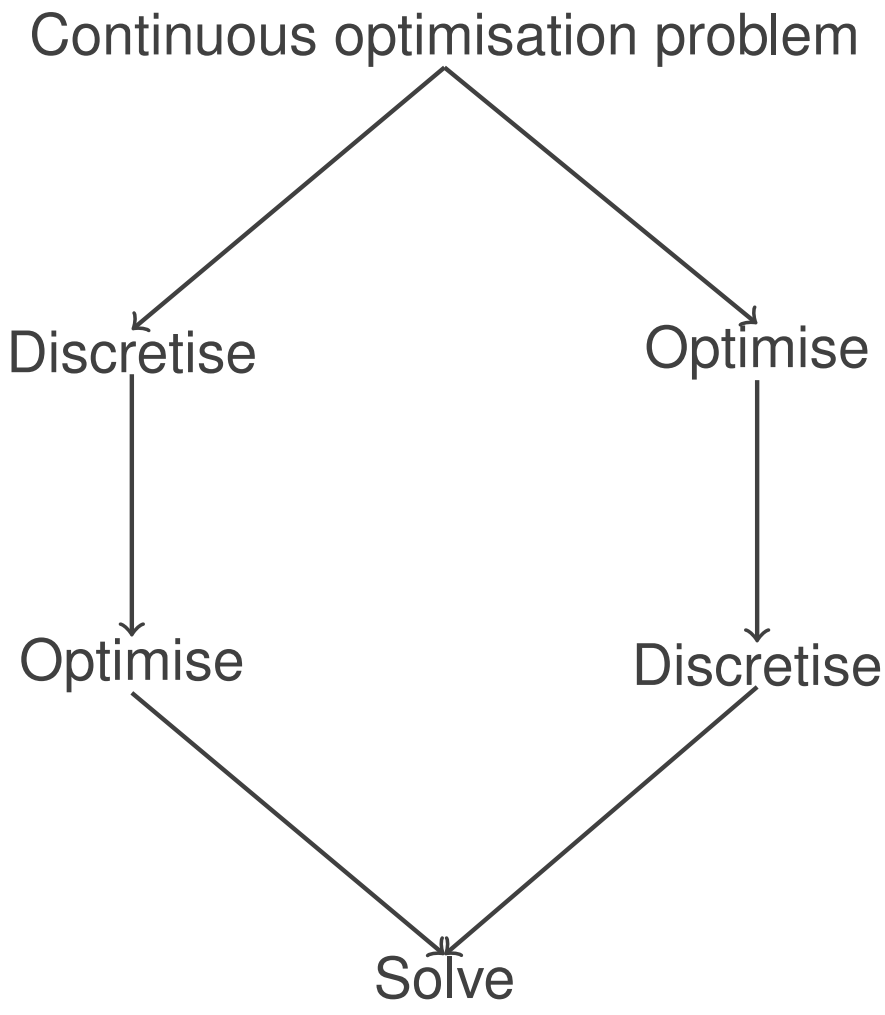
where

- $m \in L^2(\Omega)$ is the heating power
- $u \in H^1_0(\Omega)$ is the weak solution to:

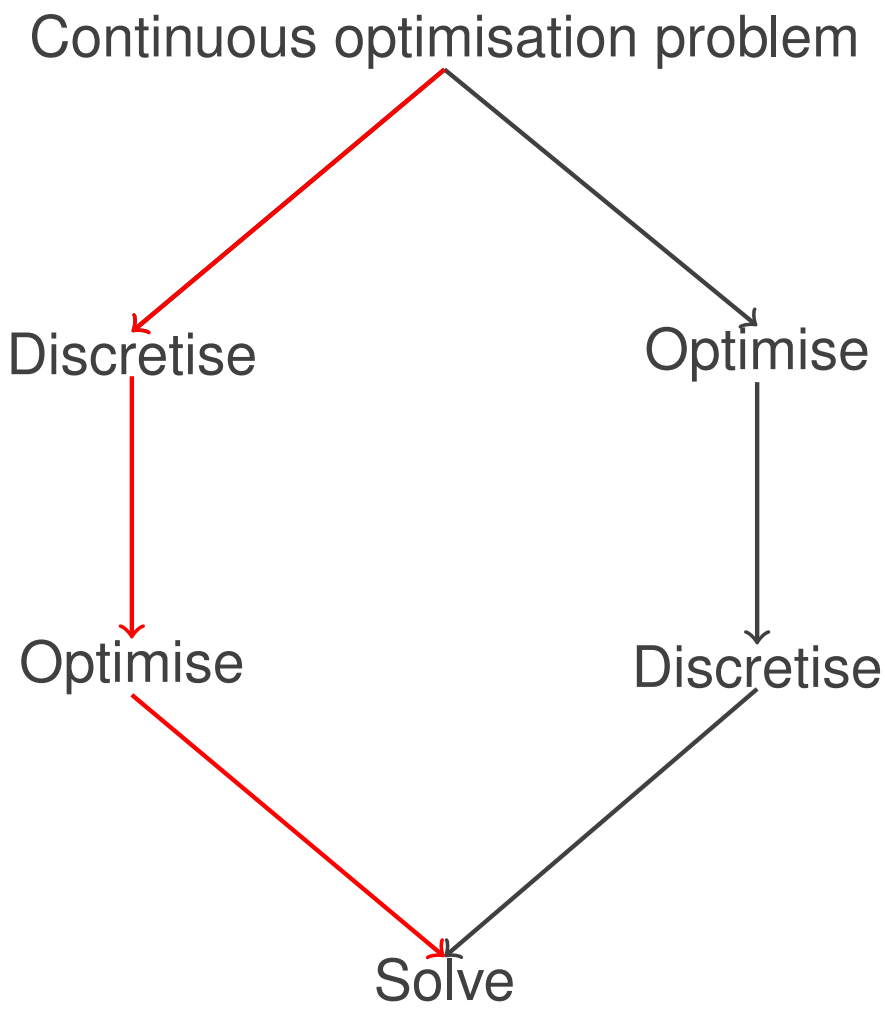
$$-\Delta u = m$$

- $d \in L^2(\Omega)$ is the desired temperature profile
- $\alpha \in \mathbb{R}$ is a regularisation parameter

There are two strategies for solving the optimisation problem



There are two strategies for solving the optimisation problem



Step 1: Discretise the model problem

Continuous problem:

$$\min_{(u,m)\in L^2(\Omega)\times H_0^1(\Omega)} \frac{1}{2}\|u-d\|^2_{L^2(\Omega)} + \frac{\alpha}{2}\|m\|^2_{L^2(\Omega)}$$

$$\text{subject to:}$$

$$-\Delta u = m \text{ in } \Omega$$

Discretised problem:

$$\min_{(\mathbf{u},\mathbf{m})\in\mathbb{R}^{|u|+|m|}} \frac{1}{2}(\mathbf{u}-\mathbf{d})^T M_u (\mathbf{u}-\mathbf{d}) + \frac{\alpha}{2}\mathbf{m}^T M_m \mathbf{m}$$

$$\text{subject to:}$$

$$D\mathbf{u} = M\mathbf{m}$$

where M are mass matrices and D is the discrete Laplace operator.

Step 2: Derive and solve the optimality conditions

Form the Lagrangian:

$$\begin{aligned}\mathcal{L}(\mathbf{u}, \mathbf{m}, \boldsymbol{\lambda}) =& \frac{1}{2} (\mathbf{u} - \mathbf{d})^T M_u (\mathbf{u} - \mathbf{d}) + \frac{\alpha}{2} \mathbf{m}^T M_m \mathbf{m} \\ &+ \boldsymbol{\lambda}^T (D\mathbf{u} - M\mathbf{m})\end{aligned}$$

Step 2: Derive and solve the optimality conditions

Form the Lagrangian:

$$\mathcal{L}(\mathbf{u}, \mathbf{m}, \boldsymbol{\lambda}) = \frac{1}{2} (\mathbf{u} - \mathbf{d})^T M_u (\mathbf{u} - \mathbf{d}) + \frac{\alpha}{2} \mathbf{m}^T M_m \mathbf{m} + \boldsymbol{\lambda}^T (D\mathbf{u} - M\mathbf{m})$$

Optimality conditions:

$$\begin{aligned} \mathcal{L}_u &= M_u(\mathbf{u} - \mathbf{d}) - D^T \boldsymbol{\lambda} &&= 0 && \text{(adjoint PDE)} \\ \mathcal{L}_m &= \alpha M_m \mathbf{m} - M^T \boldsymbol{\lambda} &&= 0 && \text{(gradient equation)} \\ \mathcal{L}_\lambda &= D\mathbf{u} - M\mathbf{m} &&= 0 && \text{(PDE)} \end{aligned}$$

Step 2: Derive and solve the optimality conditions

Form the Lagrangian:

$$\begin{aligned}\mathcal{L}(\mathbf{u}, \mathbf{m}, \boldsymbol{\lambda}) =& \frac{1}{2} (\mathbf{u} - \mathbf{d})^T M_u (\mathbf{u} - \mathbf{d}) + \frac{\alpha}{2} \mathbf{m}^T M_m \mathbf{m} \\ &+ \boldsymbol{\lambda}^T (D\mathbf{u} - M\mathbf{m})\end{aligned}$$

Optimality conditions in matrix form:

$$\begin{bmatrix} M_u & 0 & D^T \\ 0 & \alpha M_m & -M^T \\ D & -M & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ \mathbf{m} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{d} \\ 0 \\ 0 \end{bmatrix} \quad \begin{matrix} (\text{adjoint PDE}) \\ (\text{gradient condition}) \\ (\text{state PDE}) \end{matrix}$$

Solve with MINRES and a carefully chosen preconditioner.

Alternative step 2: Reduce problem, then optimise

$$\min_{\mathbf{u},\,\mathbf{m}\in\mathbb{R}^{|u|+|m|}} \frac{1}{2}\left(\mathbf{u}-\mathbf{d}\right)^T M_u \left(\mathbf{u}-\mathbf{d}\right) + \frac{\alpha}{2}\mathbf{m}^T M_m \mathbf{m}$$

subject to:

$$D\mathbf{u} = M\mathbf{m}$$

Alternative step 2: Reduce problem, then optimise

$$\min_{\mathbf{m}\in\mathbb{R}^{|m|}} \underbrace{\frac{1}{2}\left(D^{-1}M\mathbf{m}-\mathbf{d}\right)^T M_u \left(D^{-1}M\mathbf{m}-\mathbf{d}\right) + \frac{\alpha}{2}\mathbf{m}^T M_m \mathbf{m}}_{:=J(\mathbf{m})}$$

Alternative step 2: Reduce problem, then optimise

$$\min_{\mathbf{m}\in\mathbb{R}^{|m|}} \underbrace{\frac{1}{2}\left(D^{-1}M\mathbf{m}-\mathbf{d}\right)^T M_u \left(D^{-1}M\mathbf{m}-\mathbf{d}\right) + \frac{\alpha}{2}\mathbf{m}^T M_m \mathbf{m}}_{:=J(\mathbf{m})}$$

Optimality condition:

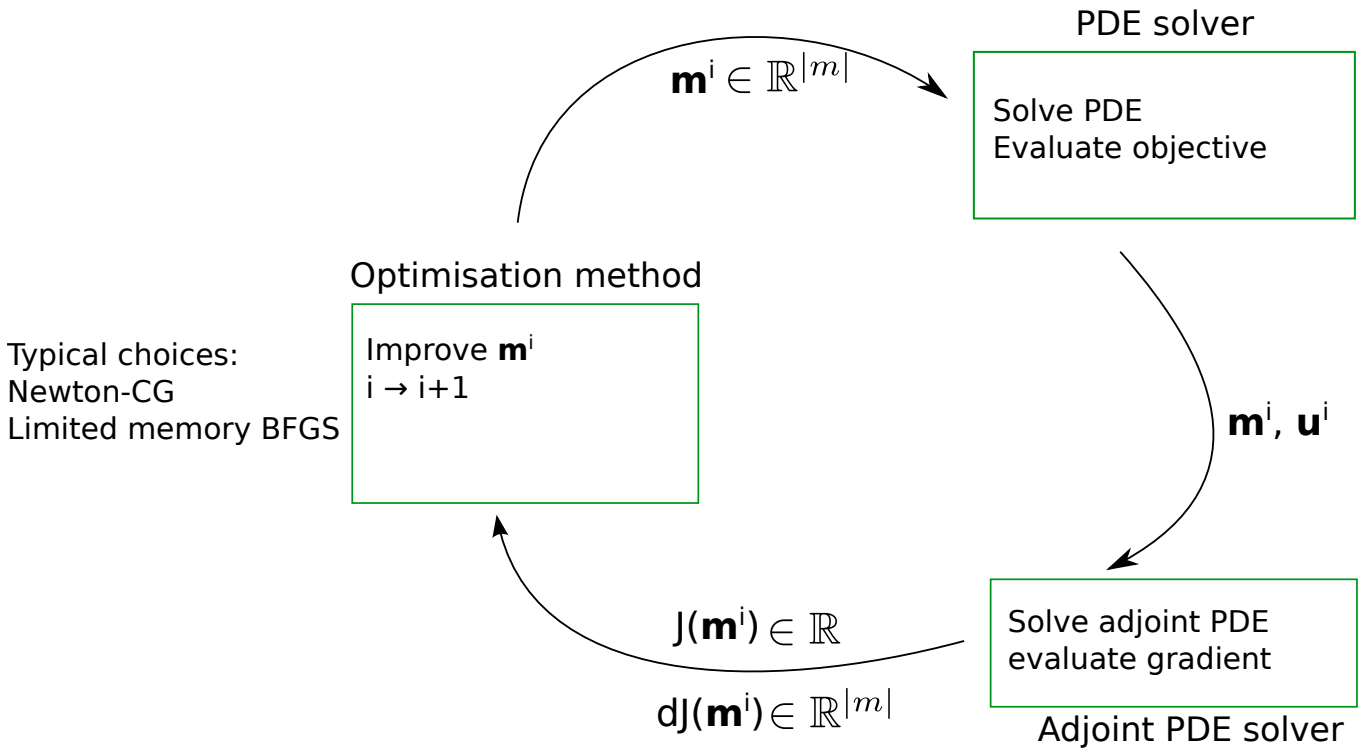
$$dJ(\mathbf{m}) = 0$$

Alternative step 2: Reduce problem, then optimise

$$\min_{\mathbf{m}\in\mathbb{R}^{|m|}} \underbrace{\frac{1}{2}\left(D^{-1}M\mathbf{m}-\mathbf{d}\right)^T M_u \left(D^{-1}M\mathbf{m}-\mathbf{d}\right) + \frac{\alpha}{2}\mathbf{m}^T M_m \mathbf{m}}_{:=J(\mathbf{m})}$$

Optimality condition:

$$dJ(\mathbf{m}) = 0$$



The reduced approach performs a block Gauss-Seidel like iteration on the optimality system

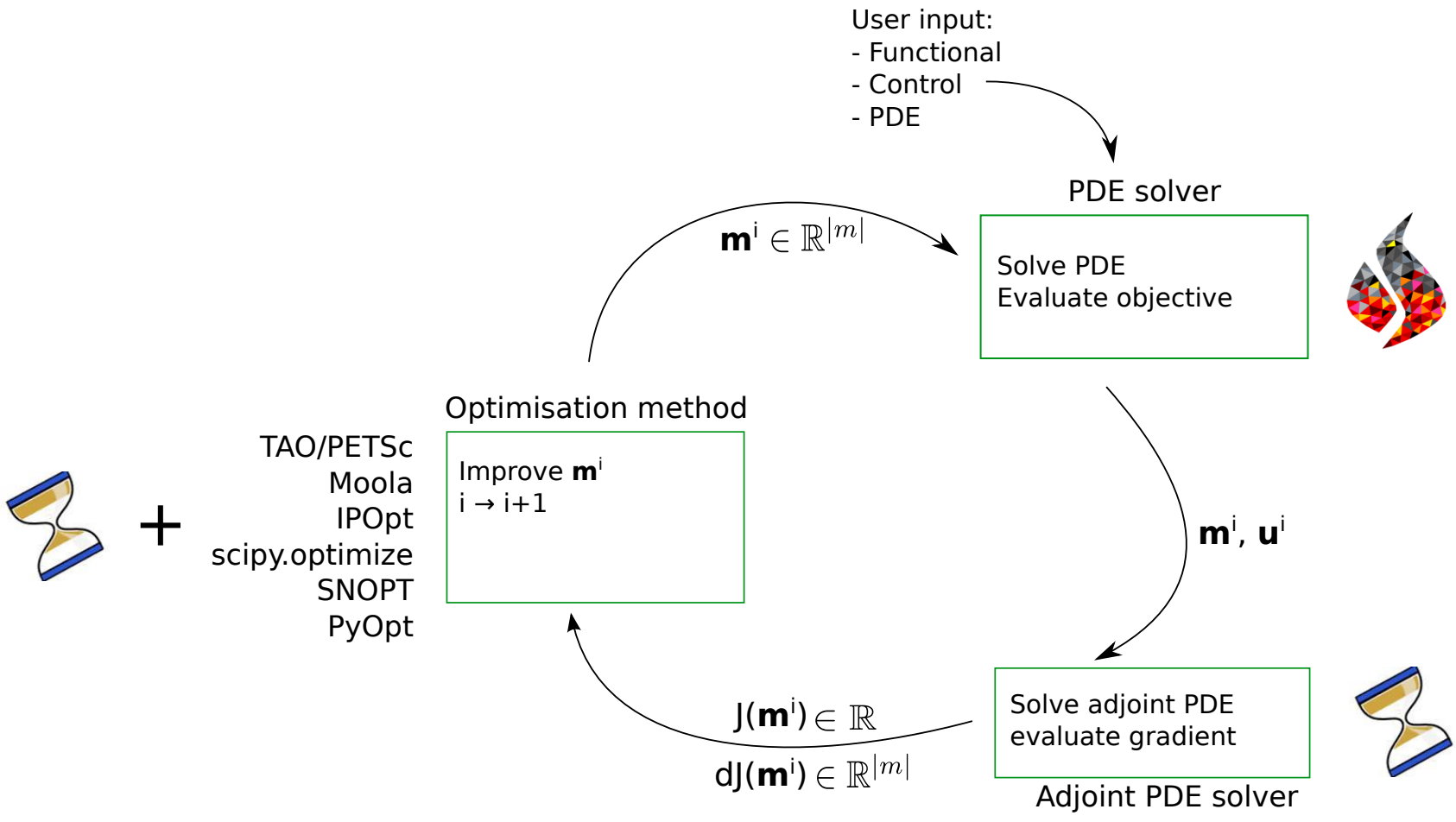
Full-space approach:

$$\begin{array}{l}(2)\\(3)\\(1)\end{array}\quad\begin{bmatrix} M_u & 0 & D^T \\ 0 & \alpha M_m & -M^T \\ D & -M & 0 \end{bmatrix}\begin{bmatrix}\mathbf{u}\\ \mathbf{m}\\ \boldsymbol{\lambda}\end{bmatrix}=\begin{bmatrix}\mathbf{d}\\ 0\\ 0\end{bmatrix}\quad\begin{array}{c}\text{(adjoint PDE)}\\ \text{(gradient condition)}\\ \text{(state PDE)}\end{array}$$

Reduced approach:

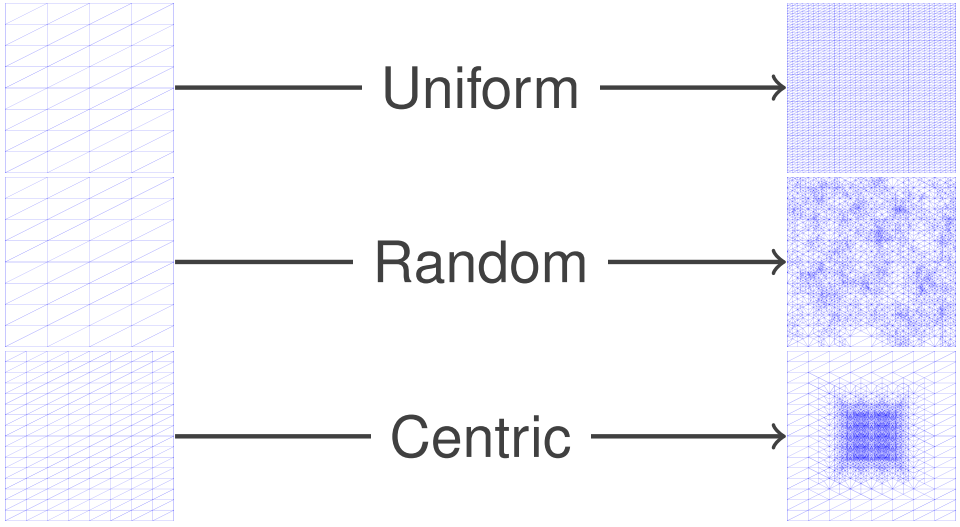
$$\mathbf{m}^0 \xrightarrow[\text{state PDE}]{(1)} \mathbf{u}^0 \xrightarrow[\text{adjoint PDE}]{(2)} \boldsymbol{\lambda}^0 \xrightarrow[\text{optimisation step}]{(3)} \mathbf{m}^1 \to \cdots$$

FEniCS/dolfin-adjoint allows to automated most steps of the optimisation



As a numerical experiment we solve the model problem on three meshes and with two solvers

- Solvers: Limited memory BFGS and Newton-CG.
- Meshes:



- Stopping criteria: $\|\mathrm{d}J\|_{L^2} < 10^{-7}$.
- Regularisation parameter: $\alpha = 10^{-6}$.
- Desired temperature: $d = \left[\frac{1}{2\pi^2} + 2\alpha\pi^2\right] \sin(\pi x) \sin(\pi y)$.

BFGS yields mesh dependent iteration numbers

- Algorithm: TAO limited memory BFGS, initial Hessian approximation based on the Broyden approximation.

Uniform refinement

Refinement level	0	1	2	3
BFGS iterations	14	12	3	4

Random refinement

Refinement level	0	1	2	3	4	5	6
BFGS iterations	14	52	77	69	74	68	81

Centric refinement

Refinement level	0	1	2	3	4	5	6
BFGS iterations	11	39	58	79	80	128	132

Newton-CG yields mesh-dependent CG iterations

- Optimisation algorithm: TAO line-search Newton-CG

Uniform refinement

Refinement level	0	1	2	3
Newton iterations	1	1	1	1

Random refinement

Refinement level	0	1	2	3	4	5	6
Newton iterations	1	1	1	1	1	1	1

Centric refinement

Refinement level	0	1	2	3	4	5	6
Newton iterations	1	1	1	1	1	1	1

Newton-CG yields mesh-dependent CG iterations

- Optimisation algorithm: TAO line-search Newton-CG

Uniform refinement

Refinement level	0	1	2	3
Newton iterations	1	1	1	1
CG iterations	13	13	8	3

Random refinement

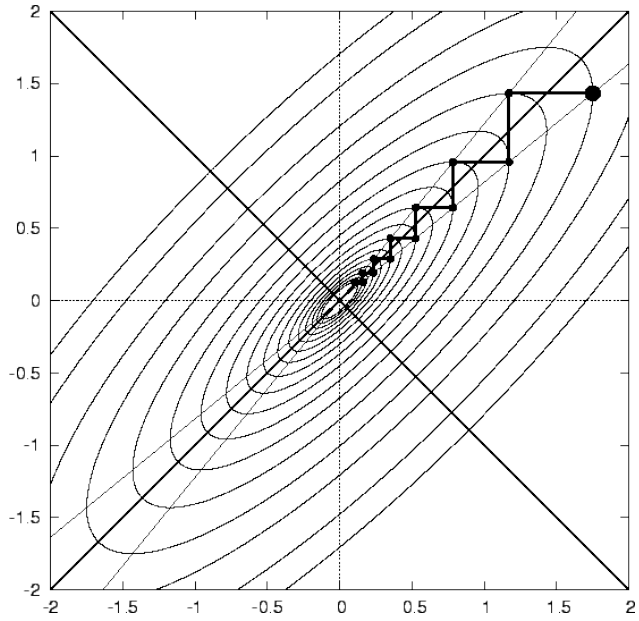
Refinement level	0	1	2	3	4	5	6
Newton iterations	1	1	1	1	1	1	1
CG iterations	13	21	25	30	36	41	49

Centric refinement

Refinement level	0	1	2	3	4	5	6
Newton iterations	1	1	1	1	1	1	1
CG iterations	13	28	38	55	65	95	131

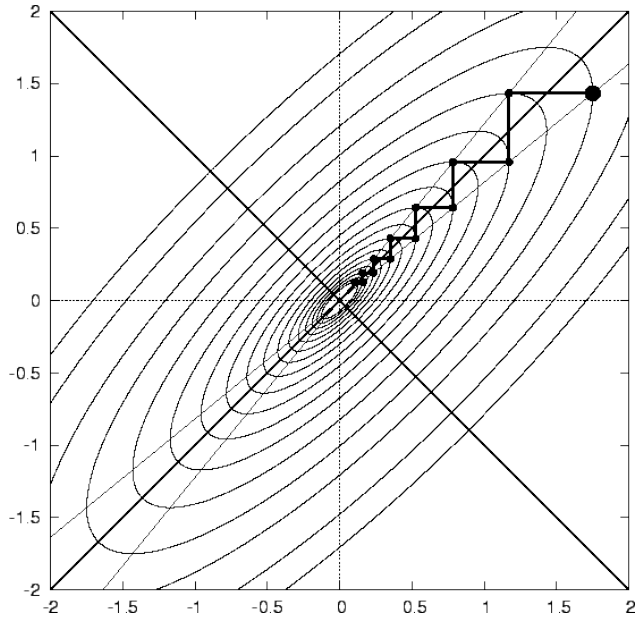
The textbook steepest descent algorithm

$i \leftarrow 0$
while $|dJ(m^i)| < \mathrm{tol}$:
$m^{i+1} \leftarrow m^i - \gamma dJ(m^i)$
$i \leftarrow i + 1$



The textbook steepest descent algorithm

$i \leftarrow 0$
while $|dJ(m^i)| < \mathrm{tol}$:
$m^{i+1} \leftarrow m^i - \gamma dJ(m^i)$
$i \leftarrow i + 1$

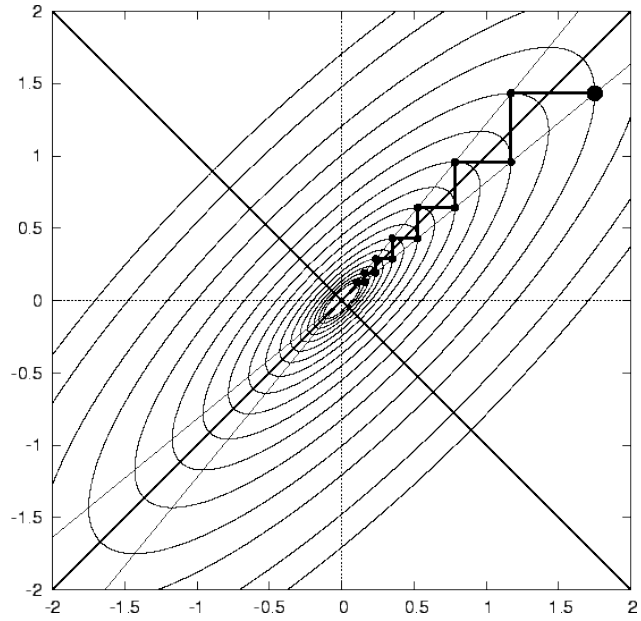


But:

- $m^i \in W$
- $J : W \to \mathbb{R}$
- $dJ(m^i) : W \to \mathbb{R} \in W^*$

Steepest descent algorithm in Hilbert space setting

$i \leftarrow 0$
while $\|dJ(m^i)\|_{W^*} < \mathrm{tol}$:
$m^{i+1} \leftarrow m^i - \gamma \nabla dJ(m^i)$
$i \leftarrow i + 1$

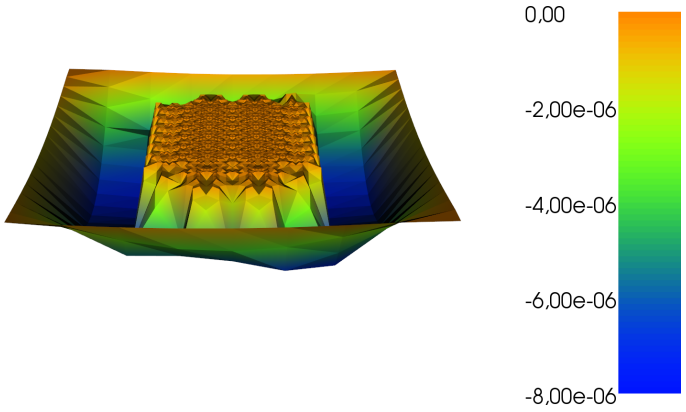


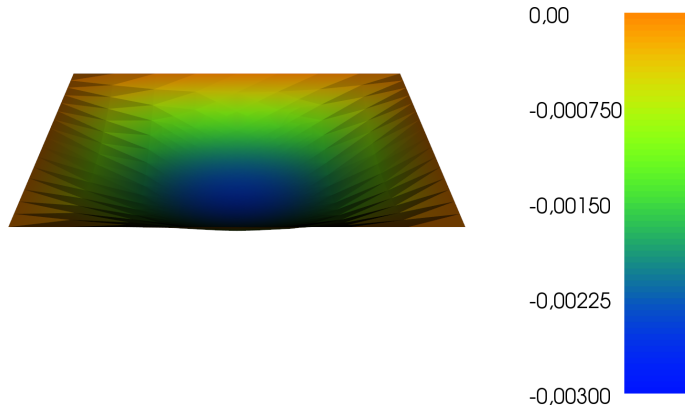
- $m^i \in W$
- $J : W \to \mathbb{R}$
- $dJ(m^i) : W \to \mathbb{R} \in W^*$
- $\nabla J(m^i) \in W$

With Riesz isomoprohism $R : W^* \to W$ the gradient is defined as $\nabla J(m) = R(dJ(m))$, i.e.

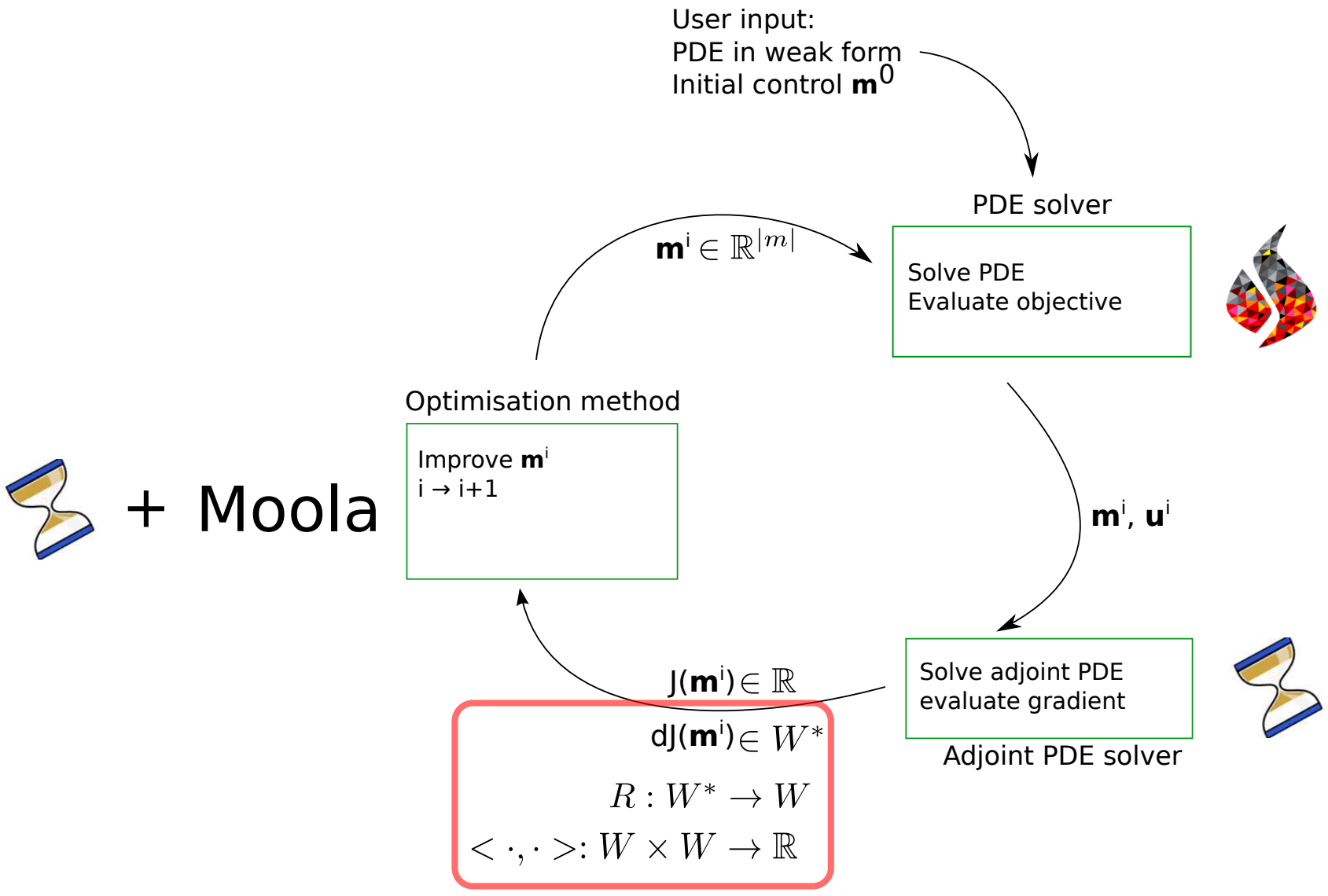
$$\langle dJ(m), \delta m \rangle_{W^*,W} = (\nabla f(m), \delta m)_W$$

Gradient and derivative operator are, and should be treated as different objects





Moola is a new Python library that implements optimisation algorithms in Hilbert spaces



Moola treats vectors in primal and dual space as different objects

The BFGS algorithm in Hilbert setting

$i \leftarrow 0$
while $|dJ(m^i)| < \mathrm{tol}$:
$d^i \leftarrow -B_i^{-1} dJ(m^i)$
$m^{i+1} \leftarrow m^i - \gamma d^i$
$i \leftarrow i + 1$

with

$s_i = m^{i+1} - m^i \in W$
$y_i = dJ(m^{i+1}) - dJ(m^i) \in W^*$

and the inverse Hessian approximation $H_{i+1} : W^* \to W$

$$B_{i+1}^{-1}(x) = \left(\mathrm{id} - \frac{y_i(\mathrm{id})}{y_i(s_i)} s\right) \left[B^{-1}\left(x - \frac{x(s_i)}{y_i(s_i)} y_i\right)\right] - \frac{x(s_i)}{y_i(s_i)} s_i$$

The resulting code is relatively compact

Hilbert space based BFGS yields mesh independent iterations

- Algorithm: Moola, limited memory BFGS algorithm, identity as initial Hessian, history length: 20
- L^2 inner product.

Uniform refinement

Refinement level	0	1	2	3
BFGS iterations	16	20	20	20

Random refinement

Refinement level	0	1	2	3	4	5	6
BFGS iterations	18	18	18	20	21	20	20

Centric refinement

Refinement level	0	1	2	3	4	5	6
BFGS iterations	20	19	20	20	19	19	19

Hilbert space based Newton-CG yields mesh independent Newton and CG iterations

- Algorithm: Moola, Newton-CG
- L^2 inner product.

Uniform refinement

Refinement level	0	1	2	3
Newton iterations	2	2	2	2
CG iterations	20	24	27	28

Random refinement

Refinement level	0	1	2	3	4	5	6
Newton iterations	2	2	2	2	2	2	2
CG iterations	20	19	26	23	25	25	29

Centric refinement

Refinement level	0	1	2	3	4	5	6
Newton iterations	2	2	2	2	2	2	2
CG iterations	24	24	26	27	27	27	27

The choice of inner product is crucial to obtain mesh independent convergence

- Optimisation algorithm: Moola, limited memory BFGS

Random refinement

*H*1 *regularisation, L2 inner product in Moola*

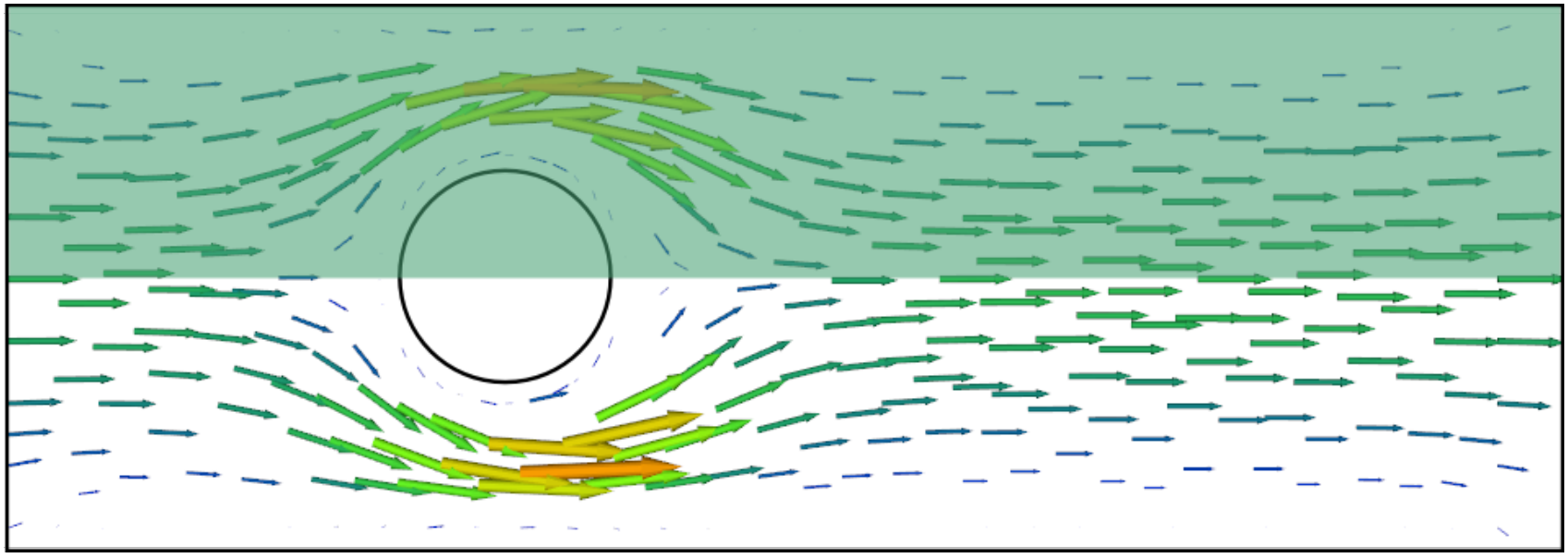
Refinement level	0	1	2	3	4	5	6
BFGS iterations	31	38	64	118	> 200	> 200	

*H*1 *regularisation, H*1 *inner product in Moola*

Refinement level	0	1	2	3	4	5	6
BFGS iterations	5	5	5	5	5	6	4

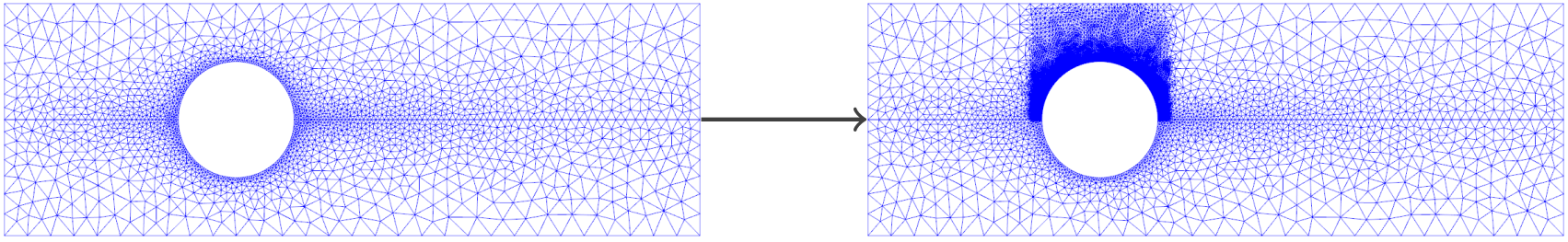
A more advanced example

Consider Navier-Stokes flow around a cylinder driven by a pressure difference at the left and right boundaries:



The choice of inner product is crucial to obtain mesh independent convergence

- Optimisation algorithm: Moola, Newton-CG
- Stopping criteria: $\|\mathrm{d}J\|_{L^2} < 10^{-4}$.
- Regularisation parameter: $\alpha = 10^{-4}$.



Refinement

Refinement level	0	1	2	3	4
Newton iterations	6	5	5	5	5
CG iterations	11	10	10	10	10

Conclusions

- It is important to keep in mind that we solve instances of infinite dimensional problems.
- Mesh-independence convergence is obtained by respecting the inner product of the underlying function spaces.
- Alternatively the optimisation algorithm in Euclidian norm may be preconditioned with a Riesz-preconditioner.
- The choice of inner product is important to achieve mesh-independent convergence.