Released to public domain under Creative Commons license type BY (https://creativecommons.org/licenses/by/4.0).
Copyright (c) 2018 SINBAD consortium - SLIM group @ The University of British Columbia.

A large-scale time-domain modeling and inversion workflow in Julia

Philipp A. Witte and Felix J. Herrmann

SLIM

University of British Columbia

Why Julia?

SLIM

dynamically typed languages

- interpreted languages:
  - Matlab
  - Python
  - Perl
  - Ruby etc.
- easy to write Code (no type declarations)

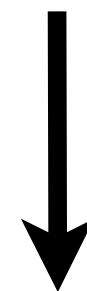
statically typed languages

- compiled languages:
  - C, C++, C#
  - Fortran
  - Java, etc.
- types declared at compilation time
- higher performance

Why Julia?

dynamically typed languages

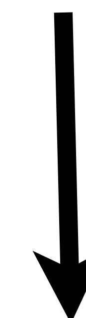
- interpreted languages:
  - Matlab
  - Python
  - Perl
  - Ruby etc.
- easy to write Code (no type declarations)



Scientists

statically typed languages

- compiled languages:
  - C, C++, C#
  - Fortran
  - Java etc.
- types declared at compilation time
- higher performance



Programmers

Why Julia?

dynamically typed languages

- interpreted languages:
  - Matlab
  - Python
  - Perl
  - Ruby etc.
- easy to write Code (no type declarations)

Julia

statically typed languages

- compiled languages:
  - C, C++, C#
  - Fortran
  - Java etc.
- types declared at compilation time
- higher performance

Why Julia?

Main features of Julia:

- flexible dynamic language
- optional typing and just-in-time compilation (JIT)
- multiple dispatch (function overloading)
- performance in range of statically typed languages like C
- easy to write parallel programs (parallel loops, shared arrays)
- easy interaction with other languages
- free and open source
- large community with people from scientific computing and applied maths
- many packages for linear algebra, optimization etc.

Why Julia?

	Fortran gcc 5.1.1	Julia 0.4.0	Python 3.4.3	R 3.2.2	Matlab R2015b	Octave 4.0.0	Mathematica 10.2.0	JavaScript V8 3.28.71.19	Go go1.5	LuaJIT gsl-shell 2.3.1	Java 1.8.0_45
fib	0.70	2.11	77.76	533.52	26.89	9324.35	118.53	3.36	1.86	1.71	1.21
parse_int	5.05	1.45	17.02	45.73	802.52	9581.44	15.02	6.06	1.20	5.77	3.35
quicksort	1.31	1.15	32.89	264.54	4.92	1866.01	43.23	2.70	1.29	2.03	2.60
mandel	0.81	0.79	15.32	53.16	7.58	451.81	5.13	0.66	1.11	0.67	1.35
pi_sum	1.00	1.00	21.99	9.56	1.00	299.31	1.69	1.01	1.00	1.00	1.00
rand_mat_stat	1.45	1.66	17.93	14.56	14.52	30.93	5.95	2.30	2.96	3.27	3.92
rand_mat_mul	3.48	1.02	1.14	1.57	1.12	1.12	1.30	15.07	1.42	1.16	2.36

(http://julialang.org/)

Why Julia?

SLIM's first steps towards Julia:

- workflow for wave-equation based inversion (LSRTM, FWI)
- Devito toolbox as wave-equation solver
- higher level abstractions for optimization
- linear operators, function-handles for black-box optimization packages

coding close to mathematical formulations + high performance

SLIM

Recap of Devito

Optimized FD schemes from symbolic PDEs

Automated code generation and Just-In-Time (JIT) compilation from symbolic Python expressions

Code design

- Level 1: Set up problem geometry
(-> PyObject for forward, adjoint, born modeling)
- Level 2: Once a specific operation is called (e.g. forward solve), set up symbolic PDE and generate FD scheme
- Level 3: Generate C code with optimized FD scheme and compile it
- Level 4: Solve wave equation

Recap of Devito

Optimized FD schemes from symbolic PDEs

Automated code generation and Just-In-Time (JIT) compilation from symbolic Python expressions

Code design

- Level 1: Set up problem geometry
  (-> PyObject for forward, adjoint, born modeling)
- Level 2: Once a specific operation is called (e.g. forward solve), set up symbolic PDE and generate FD scheme
- Level 3: Generate C code with optimized FD scheme and compile it
- Level 4: Solve wave equation

Python

C

Recap of Devito

Optimized FD schemes from symbolic PDEs

Automated code generation and Just-In-Time (JIT) compilation from symbolic Python expressions

Code design

- Level 0: Spot operators (F, J), objective functions, parallelization
- Level 1: Set up problem geometry
  (-> PyObject for forward, adjoint, born modeling)
- Level 2: Once a specific operation is called (e.g. forward solve), set up symbolic PDE and generate FD scheme
- Level 3: Generate C code with optimized FD scheme and compile it
- Level 4: Solve wave equation

Julia

C

Recap of Devito

Optimized FD schemes from symbolic PDEs

Automated code generation and Just-In-Time (JIT) compilation from symbolic Python expressions

Code design

- Level 0: Spot operators (F, J), objective functions, parallelization
- Level 1: Set up problem geometry
  (-> PyObject for forward, adjoint, born modeling)

Julia

- Level 2: Once a specific operation is called (e.g. forward solve), set up symbolic PDE and generate FD scheme
- Level 3: Generate C code with optimized FD scheme and compile it

Python

- Level 4: Solve wave equation

C

Intermediate code design

Code design

Level 1: set up problem geometry (user level)

Julia

```
14
15 # Grid points, spacing and origin
16 n = (80, 80, 80)
17 d = (10., 10., 10.)
18 o = (0.,0.,0.)
19
20 # Velocity model
21 v = ones(n) + .4
22 v[:,:,Int(round(n[3]/2)):end] = 4.
23 v0 = smooth10(v,n)
24
25 # Slowness squared
26 m = (1./v).^2.
27 m0 = (1./v0).^2.
28 dm = m - m0
29
30 # Set source function (ricker wavelet)
31 f0 = 0.010  # source peak frequency [kHz]
32 tmax = 1000.    # modeling time end [ms]
33 dt = calculate_dt(n,d,o,v)
34 q = source(tmax,dt,f0)
35
36 # Source coordinates [m]
37 srcnum = 1
38 xsrc = 200
39 ysrc = 400
40 zsrc = 40
41
42 # Setup receiver grid   [m]
43 nxrec = 21
44 nyrec = 101
45 xrec = linspace(50,750,nxrec)   # 21 receivers in x direction ranging from 50 to 950 meters
46 yrec = linspace(50,750,nyrec)   # 101 receivers in y direction
47 zrec = 100  # the same depth for all receivers
48
49 # Set up model structure
50 model = Model(n,d,o,tmax,xsrc,ysrc,zsrc,xrec,yrec,zrec)
51
52 # Nonlinear forward modeling
53 data = time_modeling(model,srcnum,q,m,[],'F',1);
54
```

Code design

```
106
107 function time_modeling(model::TimeDomainInversion.Model, srcnum::Int64, q::Array{Float64,1}, m::Array{Float64,3}, x, op::Char, mode::Int64)
108 # Setup time-domain linear or nonlinear foward and adjoint modeling using OPESCI/devito
109
110     # Set up model structure
111     modelPy = ct.IGrid()
112     modelPy[:shape] = model.n
113     modelPy[:create_model](model.o, model.d, sqrt(1./m))
114     dt = modelPy[:get_critical_dt]()
115     h = modelPy[:get_spacing]()
116     nt = Int(round(1 + model.tmax/ dt))
117     if mode==-1 && length(size(x))==3 ; x = squeeze(x,3); end
118
119     # Set up sources
120     src = ct.IShot()
121     source_coords = [model.xsrc model.ysrc model.zsrc]
122
123     src[:set_receiver_pos](source_coords)
124     src[:set_shape](nt,1)
125     src[:set_traces](reshape(q,length(q),1))
126
127     # Set up receiver grid
128     data = ct.IShot()
129     receiver_coords = setup_receiver_grid(model)
130     data[:set_receiver_pos](receiver_coords)
131     data[:set_shape](nt,size(receiver_coords,1))
132
133     # Initiate acoustic modeling object
134     Acoustic = ac.Acoustic_cg(modelPy, data, src, auto_tune=false)
135
136     # Modeling
137     if op=='F'
138         if mode==1
139             println("Nonlinear forward modeling")
140             (argout1, argout2) = Acoustic[:Forward]()
141
```

Set up Python objects

Call forward modeling

Julia

Code design

- Level 2: Set up symbolic PDE upon function call and generate stencil

```
139
140             println("Nonlinear forward modeling")
141             (argout1, argout2) = Acoustic[:Forward]()
142
```

Python

```
155
156         # Derive stencil from symbolic equation
157         eqn = m / rho * u.dt2 - Lap + damp * u.dt
158         s, h = symbols('s h')
159         stencil = 1.0 / (2.0 * m / rho + s * damp) * \
160             (4.0 * m / rho * u + (s * damp - 2.0 * m / rho) *
161              u.backward + 2.0 * s**2 * Lap)
162         # Add substitutions for spacing (temporal and spatial)
163         subs = {s: dt, h: model.get_spacing()}
164         super(ForwardOperator, self).__init__(nt, m.shape,
165                                               stencils=Eq(u.forward, stencil),
166                                               subs=subs,
167                                               spc_border=spc_order/2,
168                                               time_order=time_order,
169                                               forward=True,
170                                               dtype=m.dtype,
171                                               **kwargs)
172
```

- Define derivative operators
- Insert source/receivers terms
- …

Code design

- Level 3: Generate C code and compile it upon calling apply() function

```
62
63     def Forward(self, save=False, cache_blocking=None, use_at_blocks=False, cse=True):
64         fw = ForwardOperator(self.model, self.src, self.damp, self.data,
65                              time_order=self.t_order, spc_order=self.s_order,
66                              save=save, cache_blocking=cache_blocking, cse=cse)
67         if use_at_blocks:
68             self.at = AutoTuner(fw)
69             fw.propagator.cache_blocking = self.at.block_size
70
71         u, rec = fw.apply()
72         return rec.data, u
73
```

Python

- Level 4: Run loop over time steps and run compiled C code

SLIM

Parallel computing in Julia

Devito solves wave equation for 1 source

- build parallel Julia framework on top

For time-domain modeling/inversion:

- Parallelization over sources (easy, no communication within modeling functions, reduction or collection after function calls)
- Domain decomposition (difficult, communications within time loop)

For source parallelization: asynchronous (non-blocking) function calls

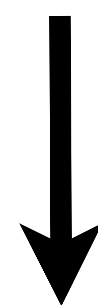
Parallel computing in Julia

Julia's parallel environment differs from MPI

- One-sided communication (manage only master process)

Higher level function calls instead of send/receive operations

- Remote calls: call function on remote process
- Remote references: object on remote process, can be used by any other process



Call modeling/gradient function on remote workers + pull result whenever needed

Source parallelization in Julia

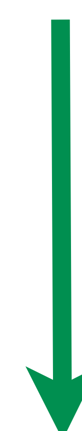
Function overloading helps to structure code

1 source

```
36 # Source coordinates [m]
37 srcnum = 1
38 xsrc = 200
39 ysrc = 400
40 zsrc = 40
41
42 # Set up model structure
43 model = Model(n,d,o,tmax,xsrc,ysrc,zsrc,xrec,yrec,zrec)
44
45 # Nonlinear forward modeling
46 data = time_modeling(model,srcnum,q,m,[],'F',1);
47
```

3 sources

```
36 # Source coordinates [m]
37 srcnum = 1:3
38 xsrc = [200, 500, 700]
39 ysrc = [400, 500, 600]
40 zsrc = [40, 45, 50]
41
42 # Set up model structure
43 model = Model(n,d,o,tmax,xsrc,ysrc,zsrc,xrec,yrec,zrec)
44
45 # Nonlinear forward modeling
46 data = time_modeling(model,srcnum,q,m,[],'F',1);
47
```



serial modeling function

parallelization over sources

SLIM

Linear operators

Linear operator package for matrix-free operations

- SPOT-like toolbox
- define operators from functions

Linear operators for non-linearized and linearized (Born) modeling

```
52
53 # Nonlinear modeling
54 F = opF(model,srcnum,q,m)
55
56 d_obs = F*q
57 q_adj = F'*d_obs
58
59
60 # Linearized born modeling
61 J = opJ(model,srcnum,q,m0)
62
63 dD = J*dm
64 image = J'*dD
65
```

(independent of number of sources)

https://github.com/JuliaSmoothOptimizers/LinearOperators.jl

SLIM

Linear operators

Linear operators can be used to easily implement algorithms

- e.g. sparsity promoting least squares migration

```
51
52 # Nonlinear modeling
53 J = opJ(model,srcnum,q,m0)
54
55 # Observed reflection data
56 y = J*dm
57
58 # Sparsity promoting LSRTM
59 x = zeros(prod(n))
60 z = zeros(prod(n))
61 lambda = 1
62 numIterations = 10
63
64 for j=1:numIterations
65
66     # Stepsize
67     t = norm(J*x - y)^2/norm(J'*(J*x-y))^2
68
69     # Update variables
70     z = z - t*J'*(J*x - y)
71     x = softThresholding(z,lambda)
72
73 end
74
```

Linear operators

Linear operators can be used to easily implement algorithms

- e.g. gradient for full-waveform-inversion

```
52
53 # Set up operators
54 F = opF(model,srcnum,q,m0)
55 J = opJ(model,srcnum,q,m0)
56
57 # FWI gradient
58 d_pred = F*q
59
60 misfit = .5*norm(d_pred - d_obs)^2.
61
62 gradient = J'*(d_pred - d_obs)
63
```

SLIM

Optimization

Objective functions that spin off gradient and function values for black-box optimization routines

```
55
56 # Nonlinear forward modeling
57 D = F*q
58
59 # Linearized forward modeling
60 dD = J*dm
61
62 # FWI
63 fval1, grad1 = pde_objective(model, srcnum, q, m0, [],  D, "fwi")
64
65 # LSRTM
66 image = zeros(model.n)
67 fval2, grad2 = pde_objective(model, srcnum, q, m0, image, dD, "lsrtm")
68
```

Optimization

Objective function can be passed to one of the many available Julia optimization packages

- Optim.jl - standard optimization algorithms for unconstrained and box-constrained optimization (BFGS, Nelder-Mead, CG)
- JuMP - linear, quadratic and non-linear constrained optimization
- LsqFit.jl - least-squares non-linear curve fitting
- NLopt.jl - interface to the NLopt library for (un-)constrained optimization
- and others

Optimization

E.g. FWI with L-BFGS and bound constraints

$$\underset{\mathbf{m}}{\text{minimize}} \quad \frac{1}{2}||\mathbf{d}_{obs} - \mathbf{PA}(\mathbf{m})^{-1}\mathbf{q}||_2^2$$

$$\text{subject to:} \quad \mathbf{m} > \mathbf{m}_{min}$$

$$\mathbf{m} < \mathbf{m}_{max}$$

```
58
59 # Set up spot operator
60 F = opF(model,srcnum,q,m)
61
62 # Generate observed data
63 D = F*q
64
65 # Function for NLopt
66 count = 0
67 function f!(x,grad)
68     fval, grad[1:end] = pde_objective(model,srcnum,q,x,[],D,"fwi")
69     global count
70     count += 1
71     println(count)
72     return fval
73 end
74
75 # LBFGS with bound constraints
76 opt = Opt(:LD_LBFGS, prod(n))
77 lower_bounds!(opt, mmin)
78 upper_bounds!(opt, mmax)
79 min_objective!(opt, f!)
80 maxeval!(opt, 20)
81 (minf, minx, ret) = optimize(opt, vec(m0))
82
```

SLIM

Unit testing

Adjoint tests for $\mathbf{F}$ and $\mathbf{J}$

$$\hat{\mathbf{d}} = \mathbf{F}\mathbf{q}$$

$$\hat{\mathbf{q}} = \mathbf{F}^T\mathbf{d}$$

$$|\mathbf{d}^T\hat{\mathbf{d}} - \mathbf{q}^T\hat{\mathbf{q}}| \leq \epsilon$$

```
51
52 # Setup spot operator
53 F = opF(model,srcnum,q,m)
54
55 # Test
56 d_hat = F*q
57 d = rand(size(d_hat))
58 q_hat = F'*d
59
60 # Result
61 println("Residual: ", abs(d'*d_hat - q'*q_hat))
62
```

$$\delta\hat{\mathbf{d}} = \mathbf{J}\delta\mathbf{m}$$

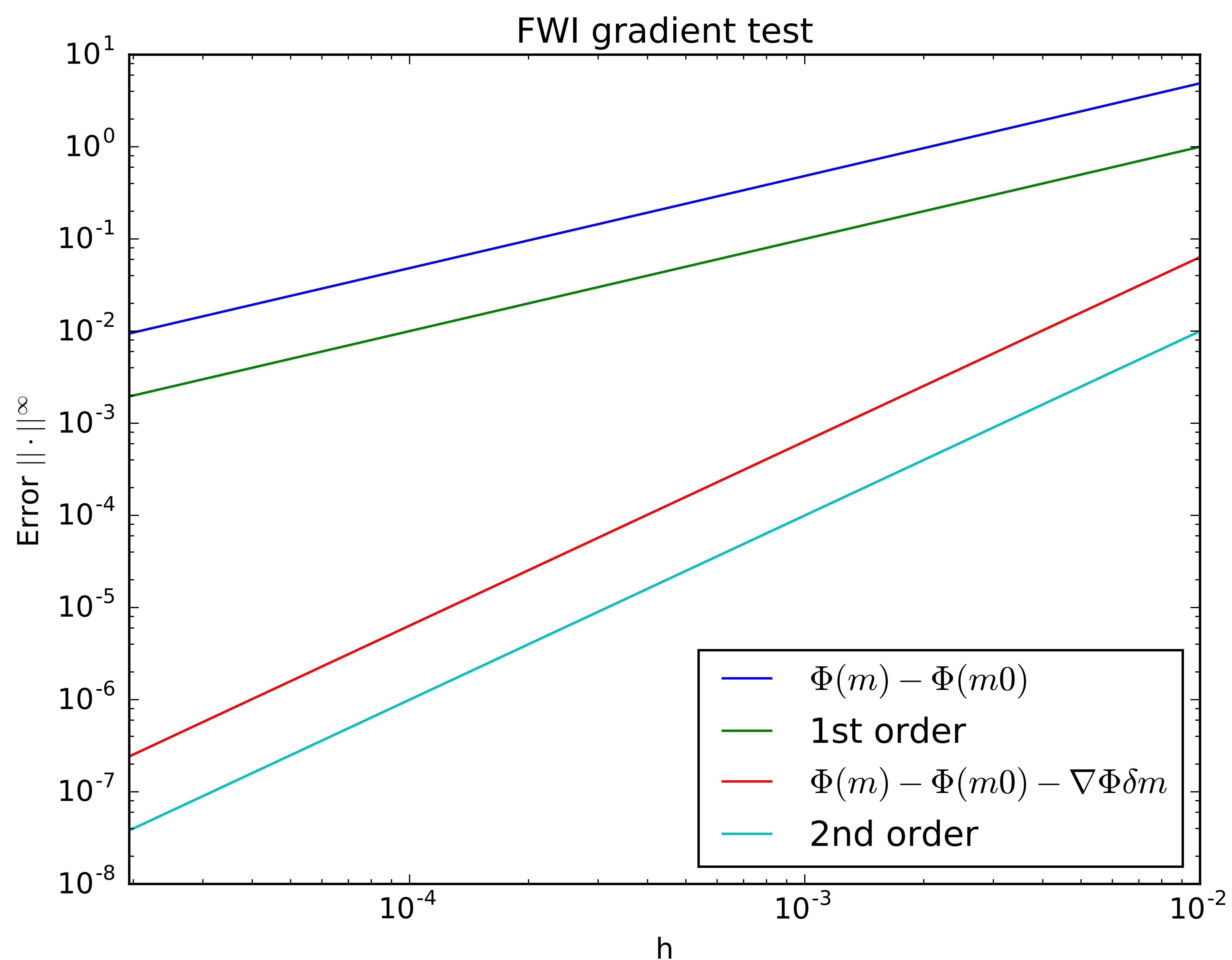
$$\delta\hat{\mathbf{m}} = \mathbf{J}^T\delta\mathbf{d}$$

$$|\delta\mathbf{d}^T\delta\hat{\mathbf{d}} - \delta\mathbf{m}^T\delta\hat{\mathbf{m}}| \leq \epsilon$$

```
65
66 # Setup spot operator
67 J = opJ(model,srcnum,q,m0)
68
69 # Test
70 dD_hat = J*dm
71 dD = rand(size(dD_hat))
72 dm_hat = J'*dD
73
74 # Result
75 println("Residual: ", abs(dD'*dD_hat - dm'*dm_hat))
76
```

Unit testing

Check correct gradient implementation of FWI objective: $\Phi(\mathbf{m}) = \frac{1}{2}||\mathbf{d}_{obs} - \mathbf{P}\mathbf{A}(\mathbf{m})^{-1}\mathbf{q}||_2^2$

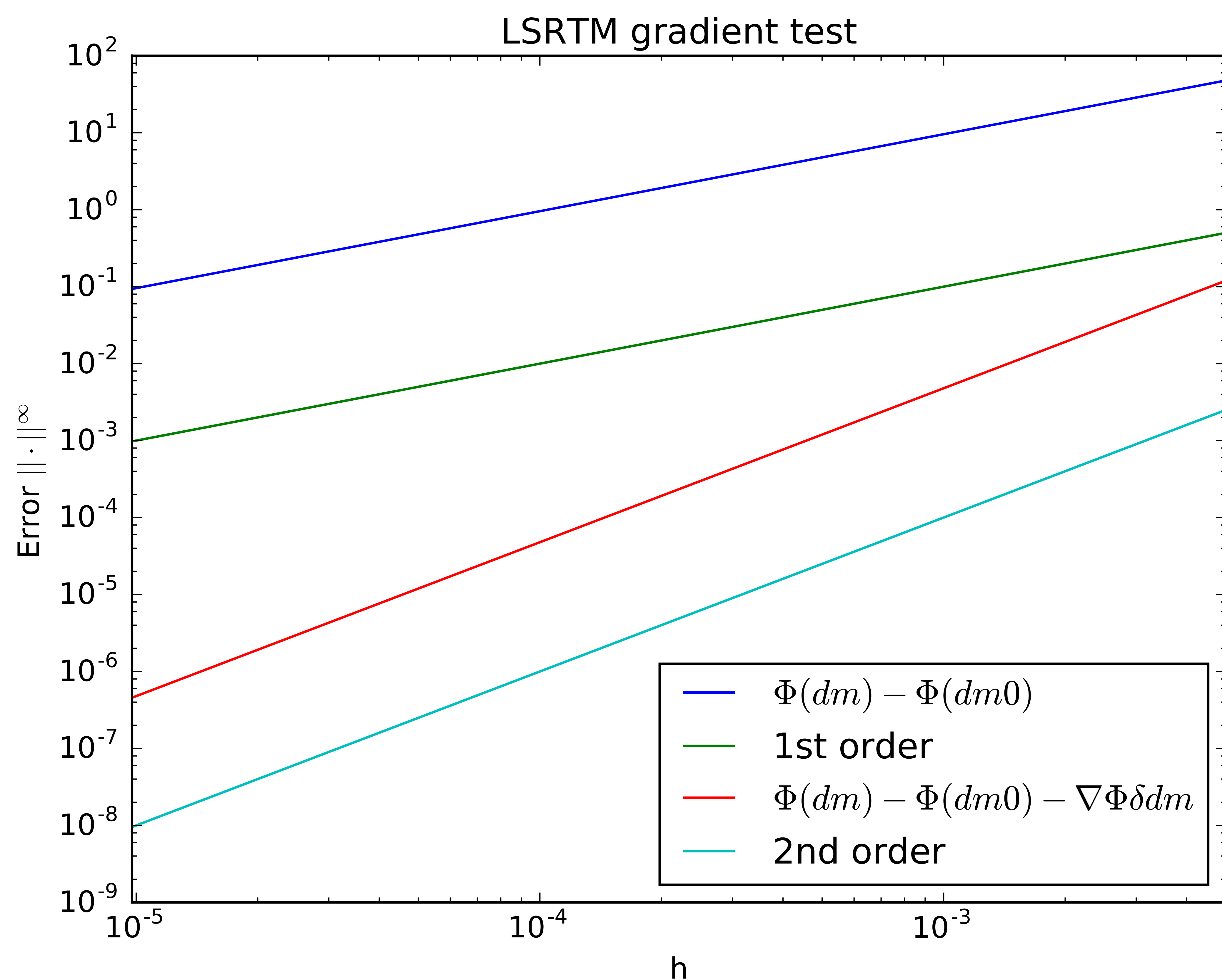


$$\Phi(\mathbf{m_0} + h \cdot \delta\mathbf{m}) - \Phi(\mathbf{m_0})$$

$$\Phi(\mathbf{m_0} + h \cdot \delta\mathbf{m}) - \left(\Phi(\mathbf{m_0}) - h \cdot \nabla_m \Phi(\mathbf{m_0})^T \delta\mathbf{m}\right)$$

Unit testing

Check correct gradient implementation of LSRTM objective: $\Phi(\delta \mathbf{m}) = \frac{1}{2}||\delta \mathbf{d}_{obs} - \mathbf{J}\delta \mathbf{m}||_2^2$



$$\Phi(\delta \mathbf{m_0} + h \cdot \Delta\delta \mathbf{m}) - \Phi(\delta \mathbf{m_0})$$

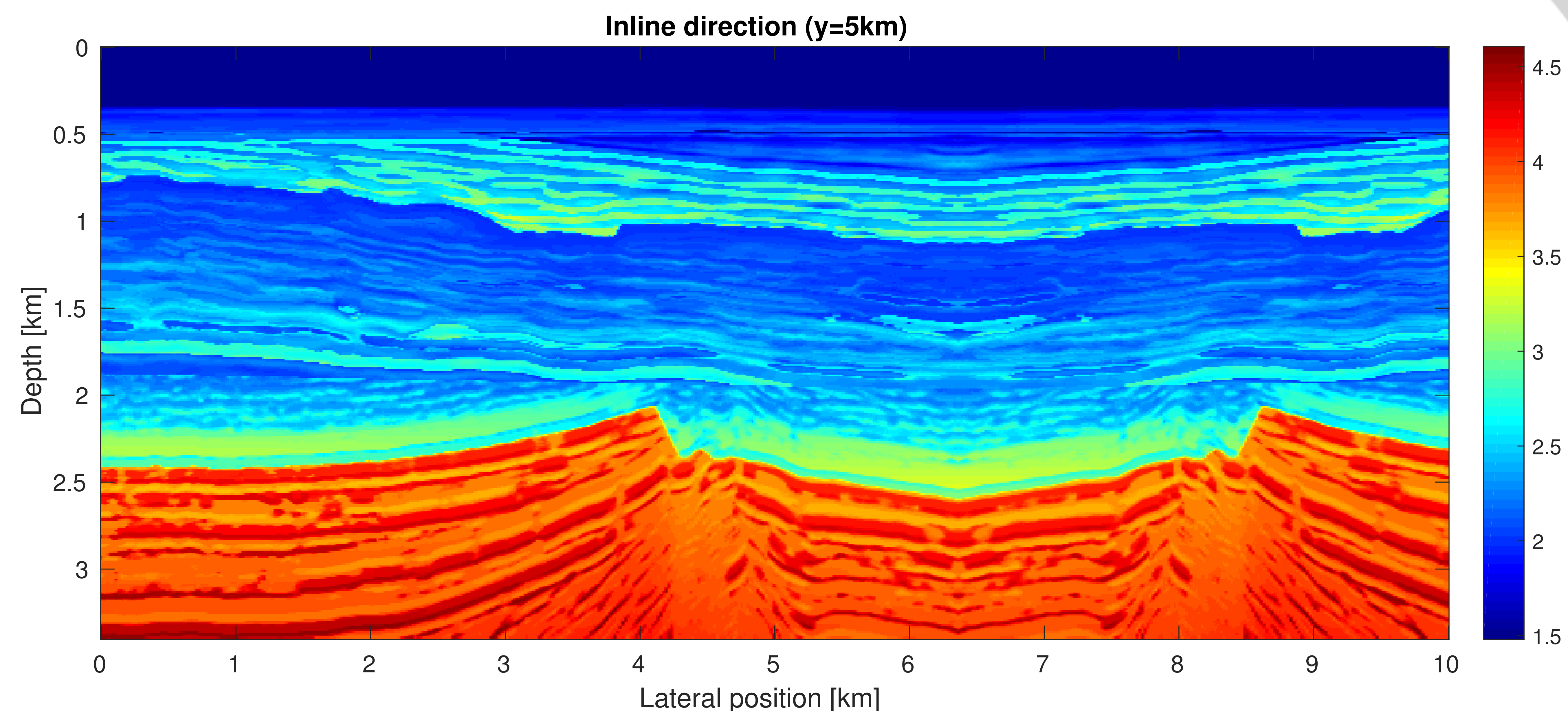
$$\Phi(\delta \mathbf{m_0} + h \cdot \Delta\delta \mathbf{m}) - \left(\Phi(\delta \mathbf{m_0}) - h \cdot \nabla_m \Phi(\mathbf{m_0})^T \Delta\delta \mathbf{m}\right)$$

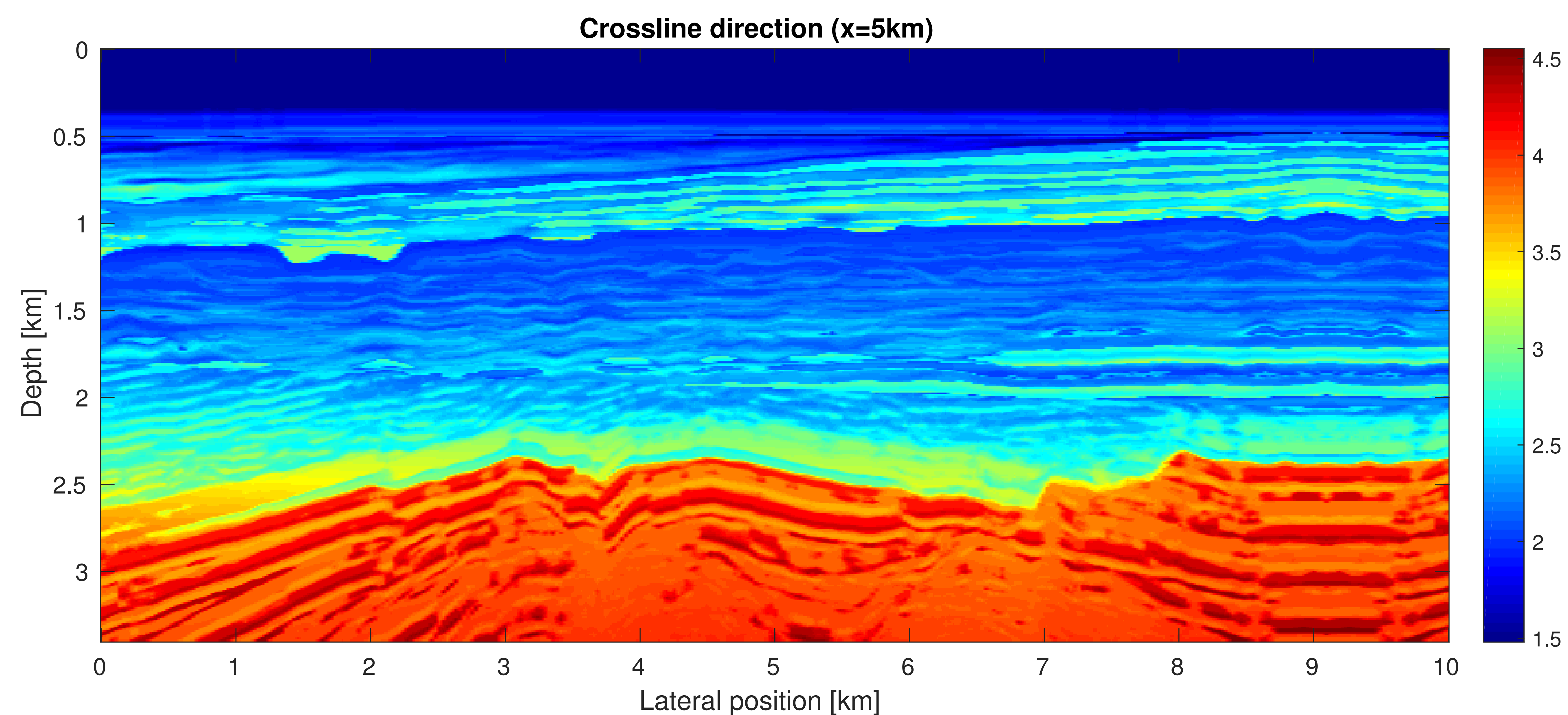
Modeling example

Model shot record on the 3D BG model:

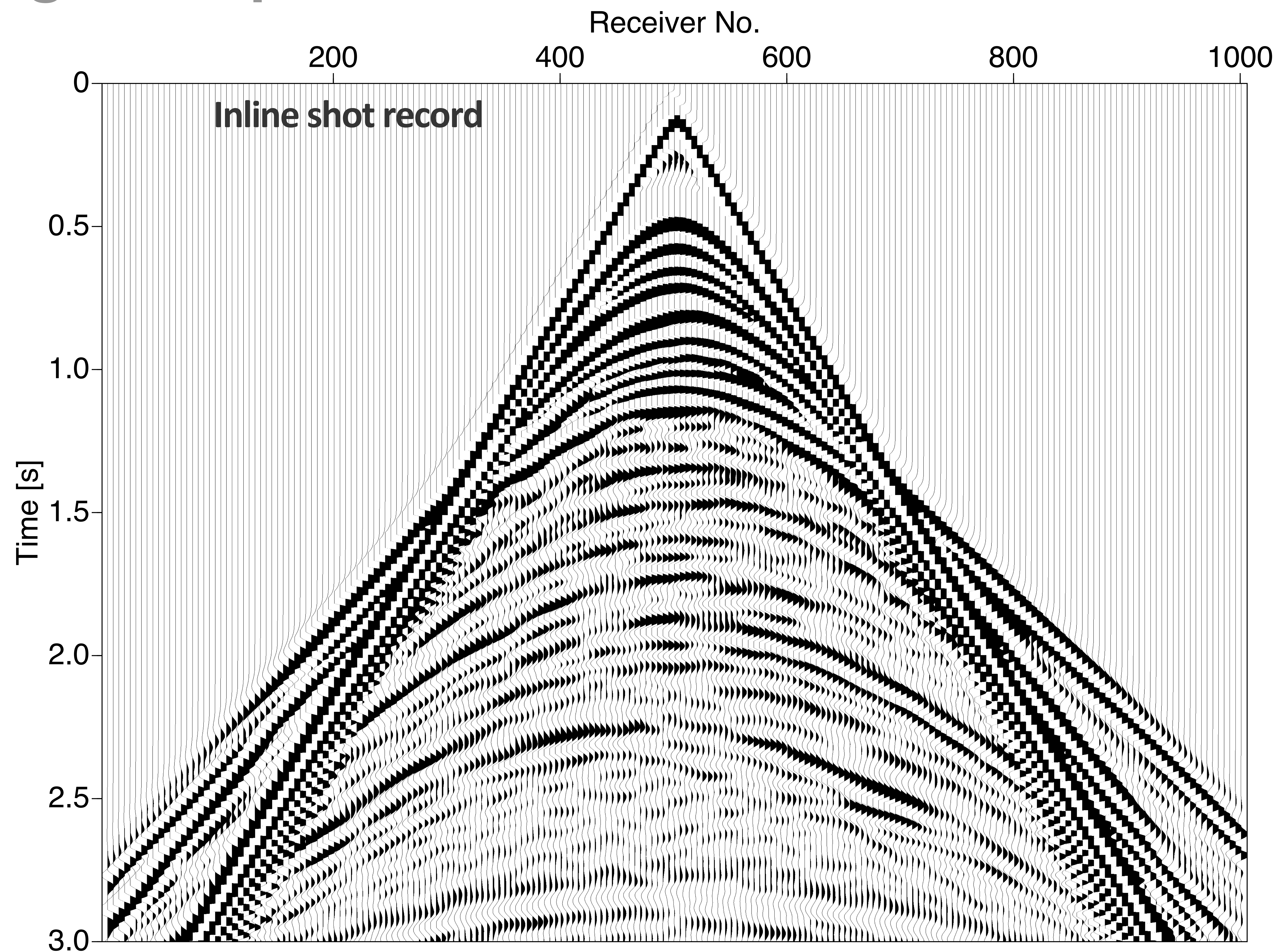
- 10 x 10 x 3.4 km
- 1000 x 1000 x 340 grid points
- 3 s recording time (3700 time steps)
- 1001 inline receivers
- 201 crossline receivers
- source in model center (10 Hz Ricker wavelet)

→ Computational time:
50 minutes



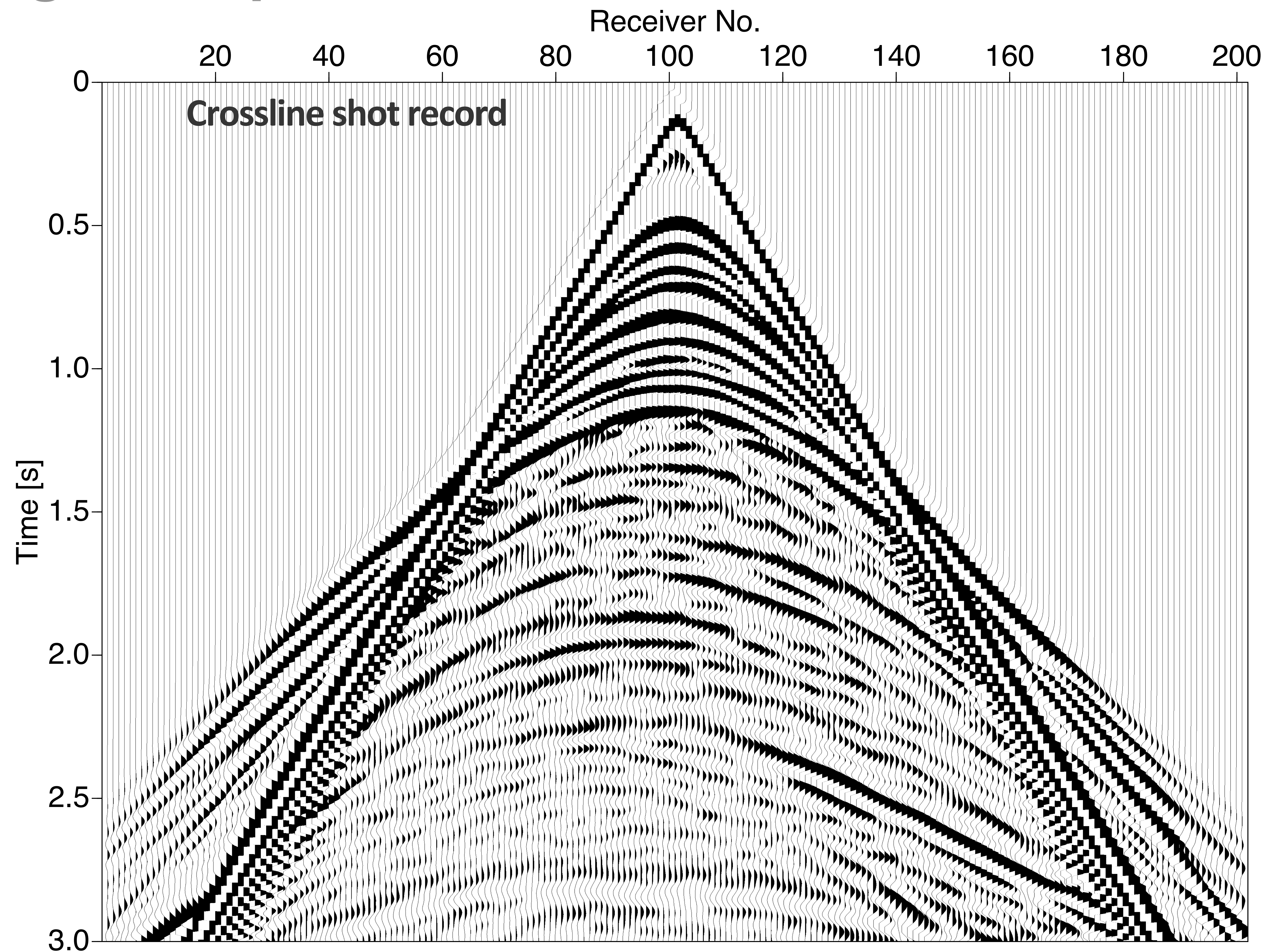


Modeling example



SLIM

Modeling example



SLIM

Outlook

We're just getting started with Julia and there's still a lot to do:

- Reading/writing SEG-Y data
- Resolve memory issues (some memory leaks)
- Translate most important Matlab functions to Julia
- Devito: replace Python data/model objects with Julia objects (prevents data copies)

Acknowledgements

This research was carried out as part of the SINBAD project with the support of the member organizations of the SINBAD Consortium.