Julia Devito: A scalable research framework for seismic inversion

Philipp A. Witte, Mathias Louboutin and Felix J. Herrmann
Research goals

SLIM yesterday:

SLIM today:

SLIM tomorrow:
Research goals

SLIM yesterday:

SLIM today:

3 orders of magnitude

SLIM tomorrow:

2 more to go
Motivation

Academic software frameworks:
› only for small 2D/3D problems (Madagascar, SeismicJulia) or
› unmaintainable low-level black-box spaghetti codes
› without detailed knowledge of code, hard to:
  - change a line search
  - keep history of gradients (e.g. for SPG)
  - change parallelization
  - change the underlying physics, correct adjoints

But there is also iWave:
› well designed, abstractions
› high accuracy, testing framework, correct adjoints
› **but**: written in C/C++, not primarily designed for performance, not very intuitive to use
Motivation

Potential applications of software:

- linear least squares problems such as LS-RTM

\[ \min_{\delta m} \frac{1}{2} || \nabla F(m_0, q) \delta m - \delta d ||^2 \]

- non-linear optimization problems such as FWI

\[ \min_{m} \phi \left( F(m, q) - d \right), \text{ where } \phi(x) = \frac{1}{2} ||x||^2 \]

Maintain flexibility:

- change \( F(m, q) \), the underlying wave equation solver
- change the formulation (different misfits \( \phi(x) \), constraints, penalties)
- choose from large variety of optimization algorithms
Overview of Julia Devito

Julia Devito is a wave-equation based inversion framework:

- non proprietary Julia programming language (public license)
- uses Devito to express and solve underlying PDEs
- matrix-free linear operators and out-of-core SEG-Y data containers
- resilient parallelization
- unified 2D-3D environment
- can interact with variety of general-purpose optimization libraries
- designed to push inversion to the next scale
- scalable but also flexible
Overview of Julia Devito

Julia

Linear operators, data containers

parallel modeling function

parallelization: distribute sources, data

serial modeling function

interface to Devito (Python)

return results

call code

set up PDEs, discretization
generate code + JIT compilation

Python

generate code

solve PDE

C
Overview of Julia Devito

Julia

Linear operators, data containers

parallel modeling function

parallelization: distribute sources, data

serial modeling function

interface to Devito (Python)

call code

set up PDEs, discretization generate code + JIT compilation

generate code

solve PDE

Python

C

Friday, October 6, 2017
Linear operators and data containers

Linear algebra notation is intuitive for seismic operations:

- forward modeling/time reversal modeling
  \[ d = \mathcal{P}_r \mathbf{F} \mathcal{P}_s^T q, \quad \hat{q} = \mathcal{P}_s \mathbf{F}^T \mathcal{P}_r^T d \]

- demigration/migration
  \[ \delta d = \mathbf{J} \delta m, \quad \hat{\delta m} = \mathbf{J}^T \delta d \]

- FWI gradients, Gauss-Newton step, etc.
  \[ g = \mathbf{J}^T (\mathcal{P}_r \mathbf{F} \mathcal{P}_s^T q - d_{\text{obs}}) \]
  \[ \delta m = (\mathbf{J}^T \mathbf{J})^{-1} \mathbf{J}^T \delta d \]
Linear operators and data containers

Challenges of this approach for time-domain modeling/inversion:

- seismic data is multidimensional volume with meta data
- simply vectorizing the input data not an option
- data typically too big to fit in memory

\[ d = \mathcal{P}_r \mathbf{F} \mathcal{P}^\top_s \mathbf{q} \]
Linear operators and data containers

Challenges of this approach for time-domain modeling/inversion:

- seismic data is multidimensional volume with meta data
- simply vectorizing the input data not an option
- data typically too big to fit in memory

\[
d = P_r F P_s^\top q
\]

- cannot be formed explicitly
- need physical information (model, source/receiver locations)
Linear operators and data containers

Challenges of this approach for time-domain modeling/inversion:

- seismic data is multidimensional volume with meta data
- simply vectorizing the input data not an option
- data typically too big to fit in memory

\[ d = \mathcal{P}_r F \mathcal{P}_s^\top q \]

- cannot be kept in memory
- not a vector
- contains header information
- cannot be formed explicitly
- need physical information (model, source/receiver locations)
Linear operators and data containers

Abstract in-core and out-of-core data vectors:
- inspired by iWave, RVL and others  
  (Symes, Padula)
- can be formed directly from single/multiple SEG-Y files
- parallel read/write chunks of data

(joint work with Keegan Lensink)

```
julia> container = segy_scan(pwd(), "overthrust_shots", ["GroupX","GroupY"]);
Scanning ... /home.slim/pwitte/overthrust_shots_41_60.segy
Scanning ... /home.slim/pwitte/overthrust_shots_21_40.segy
Scanning ... /home.slim/pwitte/overthrust_shots_61_80.segy
Scanning ... /home.slim/pwitte/overthrust_shots_1_20.segy
Scanning ... /home.slim/pwitte/overthrust_shots_81_97.segy

julia> d = joData(container)
(opesciSLIM.TimeModeling.joData{Float32}, "Julia seismic data container", 15029763, 1)

julia> size(d)
(15029763, 1)

julia> norm(d)
7371.35f0

julia> dot(d,d)
5.432854f7

julia> typeof(d.data[1])
SeisIO.SeisCon

```

(Instructional video at: https://www.youtube.com/watch?v=tx530QOPeZo)
Linear operators and data containers

Matrix-free linear operators

- read necessary meta information from data objects
- use like explicit matrices

```julia
julia> F = joModeling(info, model0)
(opesciSLIM.TimeModeling.joModeling{Float32, Float32}, "forward wave equation", 27566740206, 27566740206)

julia> Pr = joProjection(info, d.geometry)
(opesciSLIM.TimeModeling.joProjection{Float32, Float32}, "restriction operator", 15029763, 27566740206)

julia> Ps = joProjection(info, q.geometry)
(opesciSLIM.TimeModeling.joProjection{Float32, Float32}, "restriction operator", 72847, 27566740206)

julia> d_pred = Pr*F*Ps'*q
```
Overview of Julia Devito

Julia

parallelization: distribute sources, data

interface to Devito (Python)

set up PDEs, discretization
generate code + JIT compilation

Python

C

generate code

solve PDE

serial modeling function

parallel modeling function

Friday, October 6, 2017
Parallelization

Modeling multiple shots happens in parallel:

```
julia> d = Pr*F*Ps'*q
   From worker 2: Nonlinear forward modeling (source no. 1)
   From worker 5: Nonlinear forward modeling (source no. 4)
   From worker 3: Nonlinear forward modeling (source no. 2)
   From worker 4: Nonlinear forward modeling (source no. 3)
   (opesciSLIM.TimeModeling.joData{Float32}, "Seismic data vector", 240480, 1)
```

Same for adjoint modeling:

```
julia> ȧ = Ps*F'*Pr'*d
   From worker 2: Nonlinear adjoint modeling (source no. 2)
   From worker 3: Nonlinear adjoint modeling (source no. 3)
   From worker 4: Nonlinear adjoint modeling (source no. 1)
   From worker 5: Nonlinear adjoint modeling (source no. 4)
   (opesciSLIM.TimeModeling.joData{Float32}, "Seismic data vector", 2004, 1)
```
Parallelization

Parallelization in our framework:
- 2 levels of parallelization
- distribution of sources/shots (shared/distributed memory)
- parallelization over modeling domain via OpenMP (shared memory)
- in future Devito release: domain decomposition for distributed memory

Julia’s parallel framework has built-in resilience:
- in case of worker/node failure, workload is redistributed to remaining processors
- program is not interrupted
Overview of Julia Devito

Julia

- Linear operators, data containers
- Parallel modeling function
- Parallelization: distribute sources, data
- Serial modeling function
- Interface to Devito (Python)

Python

- Set up PDEs, discretization
- Generate code + JIT compilation

C

- Solve PDE
Interface to Devito

What is Devito?

‣ domain-specific language for Python
‣ symbolically set up variety of wave equations (acoustic, anisotropic)
‣ Devito compiler automatically generates optimized C code
‣ optimizes Flop count, loops, memory alignment, etc.
‣ details in the next talk by Mathias

Interfacing Devito from Julia:

‣ Julia allows direct Python and C function calls
‣ call Devito functions that generate optimized C code
‣ call generated C code directly from Julia (no data copies)
Numerical case studies

Full-waveform inversion:
- vanilla FWI w/ gradient descent and line search
- FWI with different misfit functions
- Interfacing optimization libraries for more advanced algorithms

Least-squares migration:
- parallel algorithms: LS-RTM w/ elastic average SGD
- strategies for large-scale migration: compressive LS-RTM
- imaging in the presence of salt
Example 1: FWI with a line search

Full-waveform inversion w/ least squares misfit:

\[
\text{minimize} \quad \frac{1}{2} \| \mathcal{F}(\mathbf{m}, \mathbf{q}) - \mathbf{d} \|^2
\]

Optimization:
- gradient given by \( \mathbf{g} = \mathbf{J}^\top (\mathcal{F}(\mathbf{m}, \mathbf{q}) - \mathbf{d}) \)
- implement (stochastic) gradient descent w/ approximate line search
- bound constraints for velocity
Example 1: FWI with a line search

Runnable Julia code:

```julia
# Main loop
for j=1:maxiter

    # select current batch
    idx = randperm(dobs.nsrc)[1:batchsize]
    dsub = subsample(dobs,idx)
    qsub = subsample(q,idx)

    # get fwi objective function value and gradient
    f, g = fwi_objective(model0,qsub,dsub)

    # linesearch
    step = backtracking_linesearch(vec(model0.m), g; varargs...)

    # Update model and bound projection
    model0.m = proj(model0.m + step)

    # termination criteria
    if f <= fTerm || norm(g) <= gradTerm
        break
    end
end
```

alternatively:

- line search w/ (strong) Wolfe conditions
- Barzilai-Borwein step size
- constant step size
- etc.
Example 1: FWI with a line search
Example 2: FWI with different misfit functions

Previous example:
- objective function that returns function value and gradient for $l_2$-misfit

```python
# get fwi objective function value and gradient
f, g = fwi_objective(model0, qsub, dsub)
```

Change to different misfit:
- if observed data has strong outliers: (pseudo-) Huber misfit
  $$\phi(x) = \epsilon^2 \sqrt{1 + (x/\epsilon)^2} - 1 \quad \text{(Guitton and Symes, 2003; van Leeuwen et al., 2013)}$$
- gradient given by
  $$\nabla \phi(x) = \frac{x}{\sqrt{1 + (x/\epsilon)^2}}$$
Example 2: FWI with different misfit functions

Objective function w/ $\ell_2$-misfit

```plaintext
# FWI with least squares misfit function
function fwi_objective_l2(model::Model,q::joData,d::joData)

    # Set up operators
    nt = get_computational_nt(q.geometry,d.geometry,model)
    info = Info(prod(model.n),d.nsrc,nt)
    F = joModeling(info,model,q.geometry,d.geometry)
    J = joJacobian(F,q)

    # Data residual, function value and gradient
    r = F*q - d
    f = .5*norm(r,2)^2
    g = J'*r

    return f,g
end
```

Objective function w/ pseudo-Huber misfit:

```plaintext
# FWI with pseudo-huber misfit function
function fwi_objective_huber(model::Model,q::joData,d::joData)

    # Set up operators
    ...  

    # Data residual, function value and gradient
    r = F*q - d
    f = eps^2*sqrt(1 + dot(r,r)/eps^2) - eps^2  # e.g. eps=1
    g = J'*r/sqrt(1 + dot(r,r)/eps^2)

    return f,g
end
```

change misfit independently from the rest of the code
Example 3: FWI using optimization libraries

What if we want to use more complicated algorithms?
- previous misfit functions can be passed to third-party optimization libraries
- access to large variety of optimization methods
- no need to implement everything from scratch

Tested for various libraries:
- Julia implementation of minConf (included in software release) (Schmidt et al., 2009)
- Bas’ framework for constrained optimization w/ projections onto intersections of convex sets (Peters and Herrmann, 2017)
- NLopt.jl (native Fortran library) (Johnson et al., 2017)
- Optim.jl (native Julia library) (White et al., 2017)
Example 3: FWI using optimization libraries

Does this scale to 3D?
- case study w/ 3D Overthrust velocity model
- 801 x 801 x 207 grid points + PML (222 million unknowns)
- ~10k shot records, 8 km max. offset, 3 seconds recording time (100 billion data points, over 1.2 TB of data)
- spectral-projected gradient algorithm from minConf library
- backtracking line search
- 15 iterations w/ 1080 shot records per iteration
Example 3: FWI using optimization libraries

```matlab
# Optimization parameters
fevals = 15
batchsize = 1080

# Objective function that is passed to library
function objective_function(x)
    # update model
    model0.m = reshape(x,model0.n);

    # select batch
    idx = randperm(dobs.nsrc)[1:batchsize]
dsub = subsample(dobs,idx)
qsub = subsample(q,idx)

    # fwi function value and gradient
    fval, grad = fwi_misfit(model0,qsub,dsub;options=opt)

    # reset gradient in water column to 0.
    grad = reshape(grad,model0.n)
    grad[:,:,1:21] = 0.f0

    return fval, vec(grad)
end

# Bound projection
Proj(x) = median([mmin x mmax],2)

# FWI with spectral projected gradient from minConf library
x,fval = minConf_SPG(objective_function,vec(model0.m),Proj,opt)
```

Set up 3D FWI in < 50 lines of code:
- possible to work w/ subset of shots
- still access to gradient (mute, scale etc.)
- change from SPG to L-BFGS by modifying 2 lines of code
- works with out-of-core data containers (handle arbitrary sized data sets)
- 4 minutes per gradient (1 node, 20 threads)
Example 3: FWI using optimization libraries

\[
x = 4 \text{ km} \\
y = 4 \text{ km}
\]
Example 3: FWI using optimization libraries

- $z = 400\ m$

- $z = 800\ m$
Example 4: Serial and parallel SGD

What about parallel algorithms?
- parallel version of stochastic gradient descent: elastic average SGD (Zhang et al., 2015)
- change from serial to parallel version in few lines of code

Case study for LS-RTM (but could be used for FWI as well):

\[
\begin{align*}
\text{minimize}_{\hat{\delta} \mathbf{m}} & \quad \frac{1}{2} \left\| \mathbf{M}_l^{-1} \mathbf{J} \mathbf{M}_r^{-1} \hat{\delta} \mathbf{m} - \mathbf{M}_l^{-1} \delta \mathbf{d} \right\|_2^2 \\
\end{align*}
\]

- \( \mathbf{M}_l^{-1}, \mathbf{M}_r^{-1} \) are left- and right preconditioners (model-, data-topmute, scaling, etc.) (e.g. Herrmann et al., 2008; Dai et al., 2012)
Example 4: LS-RTM w/ serial and parallel SGD

Algorithm 1 Preconditioned LS-RTM with SGD

for $j = 1$ to $n$

1. $r_j = M_l^{-1} J_r(j) M_r^{-1} x_j - M_l^{-1} \delta r(j)$
2. $g_j = M_r^{-1} J_r(j) M_l^{-1} r_j$
3. $t_j = \frac{||r_j||^2}{||g_j||^2}$
4. $x_{j+1} = x_j - t_j g_j$

end for

# Stochastic gradient descent
batchsize = 10
niter = 32

for $j=1$:niter

# Select batch
idx = randperm(dD.nsrc)[1:batchsize]
Jsub = subsample(J,idx)
dsub = subsample(dD,idx)

# Compute residual and gradient
r = Ml*Jsub*Mr*x - Ml*dsub
g = Mr'*Jsub'*Ml'*r

# Step size and update variable
t = norm(r)^2/norm(g)^2
x -= t*g

end
Example 4: LS-RTM w/ serial and parallel SGD

Algorithm 2 Preconditioned LS-RTM with elastic average SGD

for $j = 1$ to $n$ do
  for $k = 1$ to $p$ do
    $r_j = M_l^{-1} J_{jk} M_r^{-1} x^k - M_l^{-1} \delta_{jk}$
    $g_j = M_r^{-T} J_{jk} M_l^{-T} r_j$ and
    $x^k_{j+1} = x^k_j - \eta g_j^T(x^k_j) - \alpha(x^k_j - \bar{x}_j)$
  end for
  $\bar{x}_{j+1} = (1 - \beta) \bar{x}_j + \beta (1/p \sum_{i=1}^p x^i_j)$
end for

# Gradient function
@everywhere function update_x(Ml,J,Mr,x,d,eta,alpha,xav)
  $r = M_l J Mr x - M_l d$
  $g = M_r^T J^T M_l^T r$
  return $x - \eta g - \alpha (x - xav)$
end

update_x_par = remote(update_x) # Parallel function wrapper

for $j=1:niter$
  @sync begin
    for $k=1:p$
      # Select batch
      idx = randperm(dd.nsrc)[1:batchsize]
      Jsub = subsample(J,idx)
      dsub = subsample(dD,idx)
      # Calculate x update in parallel
      xnew[:,k] = update_x_par(Ml,Jsub,Mr,x[:,k],
                                dsub,eta,alpha,xav)
    end
  end
  # Update average variable
  xav = (1 - beta)*xav + beta*(1/p *sum(x,2))
  x = copy(xnew)
end

Friday, October 6, 2017
Example 4: LS-RTM w/ serial and parallel SGD

- marine streamer acquisition
- 320 shots
- 4 km maximum offset
- 32 iterations w/ 10 shots per iteration
- 1 pass through data
Example 5: Compressive inversion

Challenges of large-scale 3D inversion:

- save forward wavefields for gradient
- domain decomposition, checkpointing, boundary reconstruction
- on-the-fly DFT for frequency domain gradients  \textit{(Sirgue et al., 2010)}

Devito allows to easily implement:

- boundary reconstruction
- on-the-fly DFT
- domain decomposition + checkpointing require more effort (w.i.p.)
Example 5: Compressive inversion

Implement inversion with on-the-fly DFT in Devito:

- sum wavefields in the forward time loop
- two extra lines of Python code

```python
90  # On-the-fly real-valued DFT
91  eqn_f_r = [Eq(ufr, ufr + u*cos(2*np.pi*f*time*dt))]
92  eqn_f_i = [Eq(ufi, ufi + u*sin(2*np.pi*f*time*dt))]
```

- similar change for gradients

Run FWI or LS-RTM **at any scale:**

- only save few frequency-domain wavefields
- integrates seamlessly into Julia framework
- applications: source encoded/simultaneous source FWI and LS-RTM

(e.g. Romero et al., 2000; Krebs et al., 2009; Herrmann et al., 2009; Dai et al., 2013)
Example 5a: Compressive FWI

FWI revisited:
- same data and script as before
- SGD w/ line search
- overlapping frequency bands from 3-15 Hz
- only save 5 wavefields in memory
Example 5b: Compressive imaging

Imaging with frequency subsampling more challenging:
- subsampling creates noisy images
- want sharp image (broad frequency band) from few frequencies

Sparsity-promoting “compressive” LS-RTM:  
(Herrmann and Li, 2012; Dai et al., 2013)
- frequency-domain imaging w/ time-domain modeling
- work with subsets of random shots and frequencies
- sparsity-promotion to address artifacts

\[
\begin{align*}
\text{minimize} \quad & \lambda \|\mathbf{C} \, \delta \mathbf{m}\|_1 + \frac{1}{2} \| \mathbf{C} \, \delta \mathbf{m} \|_2^2 \\
\text{subject to:} \quad & \mathcal{F} \mathbf{J} \, \delta \mathbf{m} = \mathcal{F} \delta \mathbf{d}
\end{align*}
\]
Example 5b: Compressive imaging

Iteration 2

- per iteration: 20 randomly select shots w/ 10 random frequencies each
Example 5b: Compressive imaging

Iteration 5

- per iteration: 20 randomly select shots w/ 10 random frequencies each
Example 5b: Compressive imaging

Iteration 10

- per iteration: 20 randomly select shots w/ 10 random frequencies each
Example 5b: Compressive imaging

Iteration 15

- per iteration: 20 randomly select shots w/ 10 random frequencies each
Example 5b: Compressive imaging

Iteration 20

- per iteration: 20 randomly select shots w/ 10 random frequencies each
Example 5b: Compressive imaging

Iteration 25

- per iteration: 20 randomly select shots w/ 10 random frequencies each
Example 5b: Compressive imaging

Iteration 30

- per iteration: 20 randomly select shots w/ 10 random frequencies each
Example 6: Imaging in the presence of salt

Imaging with salt models:
- backscattered energy from salt interfaces causes low-frequency artifacts
- Laplacian filtering, wavefield filtering, alternative imaging conditions

SPLS-RTM with linearized inverse scattering imaging condition:
- derive forward/adjoint pair of ISIC (Whitmore et al., 2012; Witte and Herrmann, 2017)

\[
\hat{J}^T \delta d = \sum_t \left\{ \text{diag} \left( \hat{u}[t] \otimes m \right) \left( F^T P_r^T \delta d \right)[t] + \sum_{i=1}^3 \text{diag} \left( \frac{\partial u[t]}{\partial x_i} \right) \frac{\partial}{\partial x_i} \left( F^T P_r^T \delta d \right)[t] \right\}
\]

\[
\hat{J} \delta m = \left\{ P_r F \text{diag} \left( \hat{u}[t] \otimes m \right) \delta m + P_r \sum_{i=1}^3 F \frac{\partial}{\partial x_i} \text{diag} \left( \frac{\partial u[t]}{\partial x_i} \right) \delta m \right\}
\]
Example 6: Imaging in the presence of salt

- sparsity-promoting LS-RTM
- linearized inverse scattering imaging condition
- 960 shots, 10 seconds recording
- 15 Hz peak frequency
- estimate source wavelet

(joint work with Mengmeng Yang)
Example 6: Imaging in the presence of salt

- sparsity-promoting LS-RTM
- linearized Bregman w/ 18 iterations
- 960 shots, 10 seconds recording
- 100 shots per iteration, 2 data passes
- estimate source wavelet on the fly

(joint work with Mengmeng Yang)
Summary

Julia framework for seismic modeling and inversion

- modular software structure
- matrix-free linear operators and out-of-core SEG-Y data containers
- implement variety of inverse problems in few lines of code
- efficient and fast PDE solves through Devito
- parallelization w/ resilience to hardware failures
- interface optimization libraries
- all ingredients for LS-RTM: correct adjoints, artifact-free salt imaging
- scales to large-scale 3D problems
The road ahead

Map Julia Devito to the cloud:
- provider independent (AWS, Google, Microsoft Azure or possibly others)
- utilize full range of cloud services (auto-scaling, elastic cache, etc.)
- scale algorithms to **ANY** number of workers

Possible future workflows:
- data sets stored in cloud (already some SEG datasets available)
- bring algorithms to the cloud and to the data
- anyone can buy cloud time and run certain algorithms on a data set
- no need to buy and maintain expensive HPC clusters
Reproducible examples

Examples from this talk can be found in the software release:

- FWI with a line search
  

- 2D and 3D FWI with spectral-projected gradient descent
  


- Preconditioned LS-RTM w/ SGD
  

- more to come!
Tutorial for data IO with SeisIO.jl

Demonstration and tutorial on Youtube:

- read and write SEG-Y in Julia (chunking, read/write blocks, create look-up tables)
- parallel scanning and reading of arbitrary size data sets

https://www.youtube.com/watch?v=tx530QOPeZo

- SeisIO.jl on git: https://github.com/slimgroup/SeisIO.jl
Acknowledgements

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