ML4Seismic Open Source Software environment

Mathias Louboutin et. al.

October 22, 2021

Released to public domain under Creative Commons license type BY (https://creativecommons.org/licenses/by/4.0)
Copyright (c) 2021, Mathias Louboutin (Georgia Tech)
Some history

SINBAD 2006-2017, UBC

- inversion, data processing, acquisition design
- (parallel) Matlab code
- commercially adopted & in production (e.g. seismic data acquisition w/ Compressive Sensing)

Closed proprietary software framework stifled innovation due to

- restricted access to commercial (Matlab) code
- issues w/ scalability on parallel clusters
- limited uptake by sponsors

Moved to an open-source software (OSS) model in the Cloud.
ML4Seismic – Software

To improve our software stack, we

- improve access via Open Source (MIT license)
- build on years of experience
- leverage hyperscale serverless Cloud computing
- employs high-level abstractions to combat complexity
- are engaged in several (international) collaborations
Open source – https://github.com/slimgroup
Demos

Afternoon breakout session:

- ML4Seismic software environment preinstalled
- showcase code development on simple problems
- allows for evaluation of software
- platform to interact w/ users

https://ml4shub.eastus.cloudapp.azure.com
Active & adopted by the community

https://github.com/slimgroup
High-level Abstractions for Seismic Inversion & ML

**JUDI.jl & JUDI4Cloud.jl:** [1]

- matrix-free linear operators and abstract data (SEGY) containers
- parallel I/O based on look-up tables, parallelism (OMP + soon MPI)
- interface to Julia’s machine learning package Flux.jl
- [https://github.com/slimgroup/JUDI.jl/](https://github.com/slimgroup/JUDI.jl/) (MIT license)

**Devito** [2][3] Devito is a joint project between Imperial & Georgia Tech with Mathias one of the two main developers.

- a domain-specific language for finite-differences in Python
- code generation with automated performance optimizations
- model parallelism (MPI, multithreading)
- [https://github.com/devitocodes/devito](https://github.com/devitocodes/devito) (MIT license)
- Adopted by industry (CVX, BP, DUG, ...)

**InvertibleNetworks.jl:**

- building blocks for invertible neural networks/normalizing flows
- scalable & gradients derived by hand
- [https://github.com/slimgroup/InvertibleNetworks.jl](https://github.com/slimgroup/InvertibleNetworks.jl) (MIT license)
State-of-the-art performance

Modeling performance on AMD 7VI2 2.5 Ghz

- Devito MPI/OpenMP
  - ISO flattened MPI=1
  - ISO Brute force MPI=1
  - ISO tensor MPI=1
  - TTI factorized 1 MPI=1
  - TTI factorized 2 MPI=1
  - TTI factorized 3 MPI=1
  - TTI brute force MPI=1
  - TTI tensor MPI=1
  - VTI factorized 1 MPI=1
  - VTI factorized 2 MPI=1
  - VTI brute force MPI=1
  - VTI tensor MPI=1

- Devito OpenMP
  - ISO flattened MPI=0
  - ISO Brute force MPI=0
  - ISO tensor MPI=0
  - TTI factorized 1 MPI=0
  - TTI factorized 2 MPI=0
  - TTI factorized 3 MPI=0
  - TTI brute force MPI=0
  - TTI tensor MPI=0
  - VTI factorized 1 MPI=0
  - VTI factorized 2 MPI=0
  - VTI brute force MPI=0
  - VTI tensor MPI=0

- Code by hand
  - ISO 2 sockets
  - ISO 1 socket
  - TTI 2 socket
  - TTI 1 socket
  - VTI 2 socket
  - VTI 1 socket
GPU support roadmap

- Support for multiple target languages
  - OpenMP, OpenACC
  - potentially: CUDA, HIP, SYCL, …
- Unreliability of the target languages’ software stack
- Multi-GPU support:
  - Make it possible to run different shots on different GPUs
  - Single-node multi-GPU via domain decomposition
  - Multi-node multi-GPU via domain decomposition
- Data movement (optimized)
- Data streaming (optimized)
- Kernel performance (best so far: 27 GPOINTS on iso-acoustic O(2, 8))
JUDI – true vertical integration

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

- **Python**
  - Devito: symbolic definition of PDE
  - Automatic code generation and JIT compilation

- **C**
  - Solve PDE w/ OpenMP parallelism

Students in math/optimizers/cs/seismic practitioners

Students in CS/math/physics people

Polyhydral compiler people
Example: Compressive seismic imaging

Least-squares migration as an elastic net:

\[
\begin{align*}
\text{minimize} & \quad \lambda \| \mathbf{C} \delta \mathbf{m} \|_1 + \frac{1}{2} \| \mathbf{C} \delta \mathbf{m} \|_2^2 \\
\text{subject to:} & \quad \sum_{i=1}^{n_s} \sum_{j=1}^{n_f} \| \mathbf{M}_l^{-1} \mathbf{J} \mathbf{M}_r^{-1} \delta \mathbf{m} - \mathbf{M}_l^{-1} \mathbf{d}_{ij} \| \leq \sigma
\end{align*}
\]

Solve via the linearized Bregman method:

- at each iteration: random subsets of sources + frequencies
- memory per source: \( \mathcal{O}(\bar{n}_f) \)
- compressive sensing: no. of samples \( \sim \) no. of grid points, non-zero entries

Compressive seismic imaging

Linearized Bregman method with JUDI:

```
for j=1:maxiter
    # Compute residual and gradient
    i = randperm(d_obs.nsrc)[1:batchsize_source]
    select_frequencies!(J, batchsize_freq)
    r = Ml*J[i]*Mr*x - Ml*d_obs[i]
    g = Mr'*J[i]'*Ml'*proj_l2(r)

    # Residual and function value
    res[j] = norm(r, 2)
    fval[j] = λ*norm(C*z, 1) + .5f0*norm(C*z, 2)^2

    # Update variables
    global z -= α*g
    global x = C*soft_thresholding(C*z, λ)
end
```
Compressive seismic imaging

Linearized Bregman method with JUDI:

```matlab
for j=1:maxiter
    # Compute residual and gradient
    i = randperm(d_obs.nsrc)[1:batchsize_source]
    select_frequencies!(J, batchsize_freq)
    r = Ml*J[i]*Mr*x - Ml*d_obs[i]
    g = Mr'*J[i]'*Ml'*proj_l2(r)

    # Residual and function value
    res[j] = norm(r, 2)
    fval[j] = λ*norm(C*z, 1) + .5f0*norm(C*z, 2)^2

    # Update variables
    global z -= α*g
    global x = C*soft_thresholding(C*z, λ)
end
```
Ultra-long offset SEG workshop
– 5 iterations 4 shots each
Serverless seismic imaging on Azure

“Small” 3D Imaging case study w/ Devito DSL + JUDI

- Data set: 1,500 source locations (~2.1 TB data)
- Model: 10 x 10 x 3.325 km (270 million unknowns)
- PDE: tilted transversely isotropic (TTI) wave equation, 3,500 time steps
- Cost: < 10,000$ on 100 E64/E64s instances (2 VMs per gradient with MPI)
- Peak performance: 140 TFLOPs
- cost single image is comparable to training a large NN
JUDI4Cloud

Simple swap-in

- **Julia**
  - Linear operators, data containers, IO
  - Parallel modeling function

- **Azure Batch parallelization**
  - Serial modeling function

- **Interface to Devito (Python)**

- **Devito: symbolic definition of PDE**
  - Automatic code generation and JIT compilation

- **C**
  - Solve PDE w/ OpenMP parallelism

---

**Students**
- Math/optimizers/cs/seismic practitioners

**CS/math/physics people**

**Polyhydral compiler people**
Clusterless implementation on Azure

**JUDI operators in Azure Batch w/ JUDI4Cloud.jl**

```julia
# Setup operators
Pr = judiProjection(info, recGeometry)
F = judiModeling(info, model; options=opt)
F0 = judiModeling(info, model0; options=opt)
Ps = judiProjection(info, srcGeometry)
J = judiJacobian(Pr*F0*adjoint(Ps), q)

# Nonlinear modeling
dobs = Pr*F*adjoint(Ps)*q
```

Once again, abstractions pay off:
- no need to refactor code
- readable codes

Setup Azure Batch (see previous talk)

Run on Azure & gather results
Compressive seismic imaging

Linearized Bregman method with JUDI:

```matlab
for j=1:maxiter
    # Compute residual and gradient
    i = randperm(d_obs.nsrc)[1:batchsize_source]
    select_frequencies!(J, batchsize_freq)
    r = Ml*J[i]*Mr*x - Ml*d_obs[i]
    g = Mr'*J[i]'*Ml'*proj_l2(r)

    # Residual and function value
    res[j] = norm(r, 2)
    fval[j] = λ*norm(C*z, 1) + 0.5*θ*norm(C*z, 2)^2

    # Update variables
    global z = α*g
    global x = C*soft_thresholding(C*z, λ)
end
```

Now runs on Azure
JUDI
the Julia Devito Inversion framework

Additional advantages:

- Devito propagators usable as standalone python
- SEGY I/O
- Extended source imaging (rank 1 source $q[t]w[x]$)
- GPU offloading, simple ENV-variable change
- Low-memory gradients with randomized trace estimation (See Tue)

Future work: expose in JUDI

- Devito’s MPI domain decomposition
- AD support
Designed for Interoperability

Combine JOLI matrix-free linear operators w/ COFII’s Jets

```julia
using JOLI, JetPack, Jets
# 1D real DFT with 1000 samples
J = joDFT(1000; DDT=Float64, RDT=Float64)
# Jet operator for circular shift by 25 samples
jop = JopCircShift(JetSpace(Float64, 1000), 25);

a = randn(1000)
# Parenthesis needed to prevent julia to compute J*jop first
a = randn(1000); b = J*(jop*a); c = jop*(J'*b);
# dot test
@show dot(b, b), dot(c, a), dot(b, b) - dot(c, a), dot(b, b) / dot(c, a)
# 490.00330785, 490.003307859, 1.1368683772161603e-13, 1.0000000000000002)
```

Once again, abstractions pay off:

- no need to refactor code
- readable codes

Seamless integration JUDI & COFII ⟷ super fast developments cycle = innovation.
Mored advanced integration synergetic

**SetIntersectionProjection.jl**
Projection onto intersection of constraints

**SlimOptim.jl**
quasi-newton

**COFII propagator**

https://github.com/slimgroup/ConstrainedFWIExamples/blob/master/notebooks/02_constr_fwi_jetpack.ipynb
Mored advanced integration
constrains from SLIM & wave propagators from CVX’s COFII

## COFII vs SLIM

<table>
<thead>
<tr>
<th></th>
<th>SLIM</th>
<th>COFII</th>
<th>Compatible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>julia, python</td>
<td>julia, C++</td>
<td>yes</td>
</tr>
<tr>
<td>Software philosophy</td>
<td>Linear Algebra</td>
<td>Jets</td>
<td>yes (see above)</td>
</tr>
<tr>
<td>Propagators</td>
<td>Devito</td>
<td>Legacy C++, potential migration to Devito</td>
<td>NA</td>
</tr>
<tr>
<td>IO</td>
<td>SEGY (via SegyIO), no cloud support yet</td>
<td>JavaSeis CloudSeis</td>
<td>not yet</td>
</tr>
<tr>
<td>Cloud</td>
<td>Azure Batch (via AzureClusterlessHPC) fully remote (local desktop/laptop)</td>
<td>Azure Virtual Machine ScaleSet Custom cloud cluster manager Azure Network only</td>
<td>no</td>
</tr>
<tr>
<td>Cloud storage</td>
<td>OpenVDS (future plan) Cloud native OSDU compliant</td>
<td>Blob manager</td>
<td>yes</td>
</tr>
<tr>
<td>OSS</td>
<td>Yes, all</td>
<td>Yes, transitioning</td>
<td>Yes</td>
</tr>
</tbody>
</table>
**ML Software**

*InvertibleNetworks.jl*: A Julia package for invertible CNNs

- Manually implemented derivatives and Jacobians that make use of invertibility
- Unit testing (adjoints, gradients, invertibility)
- [https://github.com/slimgroup/InvertibleNetworks.jl](https://github.com/slimgroup/InvertibleNetworks.jl) (MIT license)
- GPU support through CUDA.jl

**Main features:**

- Additive and affine invertible coupling layers (e.g. as in NICE, Glow)
- Invertible recurrent inference machines (i-RIM)
- Hyperbolic networks
- Hierarchical invertible neural transport (HINT) and conditional HINT
- 1 x 1 convolutions, wavelet squeezing, etc.
- Integration of Flux.jl layers
- AD with ChainRules.jl
InvertibleNetworks.jl

Class structure:

```plaintext
struct CouplingLayerGlow <: NeuralNetLayer
    C::Conv1x1
    RB::ResidualBlock
    logdet::Bool
    forward::Function
    inverse::Function
    backward::Function
end
```

Usage:

```plaintext
# Create Layer
LayerGlow = CouplingLayerGlow(nx, ny, nchannel, nhidden, batchsize)

# Operations
Y = LayerGlow.forward(X)
X_ = LayerGlow.inverse(Y)
ΔX, X_ = LayerGlow.backward(ΔY, Y)
```
Loop unrolling revisited

Invertible version of loop-unrolled network:
- Inspired by i-RIM and Real NVP
- Invertible + tractable logdet

1. function $G(J, d)$
2. $x = 0$
3. for $j = 1, ..., n$
4. $x = Qx$
5. $x'_1 = x_1$
4. $g = J^T (Jx'_1[1] - d)$
5. $s', t = \text{NN}([g, x'_1[2:end]])$
6. $s = \sigma(s')$
6. $x'_2 = x_2 \odot s + t$
3. $x = Q^\top x'$
12. end
13. return $x$
14. end
Uncertainty Quantification

Examples:


Uncertainty Quantification

Posterior sampling implemented with conditional layers in InvertibleNetworks.jl

```julia
struct ConditionalLayerHINT <: NeuralNetLayer
    CL_X::CouplingLayerHINT
    CL_Y::CouplingLayerHINT
    CL_YX::CouplingLayerBasic
    C_X::Union{Conv1x1, Nothing}
    C_Y::Union{Conv1x1, Nothing}
    logdet::Bool
    is_reversed::Bool
end
```

First, train with joint distribution target then sample from posterior distribution:

```julia
Zy_fixed = H.forward_Y(y_obs);
post_sampling_size = 50
Zx = randn(Float32, nx, ny, n_in, post_sampling_size)
X_post = H.inverse(Zx, Zy_fixed.*ones(Float32, nx, ny, n_in, post_sampling_size))[1];

#statistics over pixels in batch
mean(X_post; dims=4)
std(X_post; dims=4)
```

Next steps

Scaling of ML in 3D

Integration w/ OLIVES & generative classifiers

Further integration AD tools to combine wave-equation solvers w/ ML

Integration of Julia’s new AD tool when available

- leverages JIT compiler
- offers more control → scales well
This research was carried out with the support of Georgia Research Alliance and partners of the ML4Seismic consortium.