A Unified 2D/3D Software Environment for Large Scale Time-Harmonic Full Waveform Inversion

Curt Da Silva & Felix J. Herrmann
3D Full Waveform Inversion

Complicated process
  • computationally intensive
  • requires lots of memory, time
  • large amount of programmer effort to get things *fast*
  • often speed is the trade off for *correctness*
3D Full Waveform Inversion

Industry codebases, while fast

- are *inflexible* - hard to integrate new changes

- are *incorrect* - no ‘true derivatives’ of the underlying modelling code

- are *poorly maintained* - a new hire will have no idea what’s going on
3D Full Waveform Inversion

As a result

- codes are disconnected from mathematical underpinnings
- bugs are hard to diagnose
- difficult to incorporate new ideas from academia, research labs into production-level codes
Software organization

Software hierarchy manages complexity

• human brains have very limited working memory

• if a particular part of a program only has one function, people using/debugging it only have to think about that one function

• if software is easier to reason about -> it’s easier to work with, easier to test
Software organization

Software hierarchy manages complexity

• we don’t have to sacrifice performance
  • lowest level operations implemented in C w/multithreading

• hiding irrelevant details at each level
  • higher level functions don’t have any idea about C/fortran/that gross stuff
Software organization

Anything that we do that isn’t solving PDEs is essentially irrelevant, computation time-wise
Software organization

Anything that we do that isn’t solving PDEs is essentially irrelevant, computation time-wise

• advantageous for software design -> any overhead introduced is negligible compared to solving PDEs

• if a single wavefield can be stored in RAM - true for low frequency time harmonic FWI
Software organization

PDEs are the computational bottleneck

- design our software for maximum ease of use + “plug and play” components

- speedups made to solving PDEs propagate to whole framework
FWI Problem

\[
\min_m \frac{1}{2N_s} \sum_{i=1}^{N_s} \| P_r H(m)^{-1} q_i - d_i \|_2^2
\]

- \( m \) - discrete model vector
- \( N_s \) - number of shots
- \( P_r \) - receiver restriction operator
- \( H(m)u_i = q_i \) - monochromatic Helmholtz system for shot \( i \)
- \( d_i \) - measured data for shot
New way to organize FWI Software

Modeling matrix: multiplication/division
A SPOT operator
• linear operator class - behaves like a matrix
• knows how to multiply, divide itself
• can handle matrix-free operations or form sparse matrix for 2D problems

Extensions
• Kaczmarz sweeps
• Jacobi iterations
OpAbstractMatrix

Particular matrix-vector products specified at construction

discrete_helmholtz - constructs Helmholtz operator with particular parameters
  - can swap between stencils
  - construct multigrid preconditioner
New way to organize FWI Software

Multithreaded Mat-vec multiply

C-based MVP opAbstractMatrix

Modeling matrix: multiplication/division
C-based Matrix Vector Product

Implementation of 27-pt compact stencil [1]

Multi-threaded along the z-coordinate with openMP

Forward, adjoint modes
Helmholtz matrix

In 2D, we can afford to use explicit sparse matrices + fast direct solvers
  • implementation of [1]

Explicit matrices VS implicit matrices is opaque to the user
  • interface remains the same

C-based Matrix Vector Product

Matlab Compiler

- write stencil-based code in Matlab -> C code with openMP multithreading

- nearly as fast as native C code, much easier to develop
New way to organize FWI Software

Abstract linear solver

Multithreaded Mat-vec multiply

C-based MVP

opAbstractMatrix

LinearSolve

Modeling matrix: multiplication/division
LinearSolve

Abstract interface for “Solve $Ax = b$ with a specified method”

• encourages code reuse - smoothers for multigrid, preconditioner applications

• calls the specified method (GMRES, CG, etc.) with the prescribed number of iterations, right hand side, initial guess, tolerance, and preconditioner
LinSolveOpts

Object for storing
- linear solver method
- maximum outer iterations
- maximum inner iterations (for some solvers)
- tolerance
- preconditioner

As well as default options for these
- Solvers: CG, FGMRES, LU, etc.
- Preconditioners: ML-GMRES, Shifted Laplacian, etc.
Multilevel-GMRES

Smother \text{GMRES}(k_o, k_i) \text{ GMRES}(k_o, k_i)

Coarse solve \text{GMRES}(k_o, k_i) \text{ GMRES}(k_o, k_i) \text{ GMRES}(k_o, k_i)

Preconditioned by \text{GMRES}(k_o, k_i)

Discretization Spacing

\begin{align*}
h & \quad h \\
2h & \quad 2h \\
4h & \quad 4h
\end{align*}
New way to organize FWI Software

Abstract linear solver

PDE-related quantities
Serial version

Multithreaded Mat-vec multiply

Modeling matrix: multiplication/division
PDEfunc

Main workhorse function

For each source index
  • solve the Helmholtz equation - don’t care how
  • use solution to compute objective + gradient, demigration/migration, hessian/GN hessian matrix vector product - whatever the user requests

Serial code, implicitly multithreaded
Excerpt from PDEfunc

\[ U_k = H_k \backslash Q_k_i; \]

switch func
  case OBJ
    \[ [\phi, d\phi] = \text{misfit}(Pr \cdot U_k, Dobs(:, data_idx), current_src_idx, freq_idx); \]
    \[ f = f + \phi; \]
    if nargout >= 2
      \[ V_k = H_k' \backslash (-Pr' \cdot d\phi); \]
      \[ g = g + \text{sum}(\text{real}(\text{conj}(U_k) \cdot (dH' \cdot V_k)), 2); \]
    end

  case FORW_MODEL
    output(:, data_idx) = Pr \cdot U_k;

  case JACOB_FORW
    \[ dU_k = H_k \backslash (dHdm \cdot (-U_k)); \]
    output(:, data_idx) = Pr \cdot dU_k;

  case JACOB_ADJ
    \[ V_k = H_k' \backslash (-Pr' \cdot \text{input}(:, data_idx)); \]
    output = output + \text{sum}(\text{real}(\text{conj}(U_k) \cdot (dH' \cdot V_k)), 2);
end
New way to organize FWI Software

Abstract linear solver

Multithreaded Mat-vec multiply

C-based MVP

LinearSolve

PDEfunc

PDEfunc_dist

opAbstractMatrix

PDE-related quantities
Parallel version

PDE-related quantities
Serial version

Modeling matrix: multiplication/division
Separable objective function

\[ f_I(m) = \frac{1}{2|I|} \sum_{i \in I} \| P_r H(m)^{-1} q_i - d_i \|^2_2 \]

\[ = \frac{1}{2|I|} \sum_{i \in I} f_i(m) \]

The objective function is *separable* over shots/frequencies
- distribute indices to parallel workers

Objective separable \(\rightarrow\) gradient, GN Hessian, Hessian are separable

PDEfunc_dist does no computation, just parallel distribution + summation
- separate computation from parallelization
- easiest component to ‘swap out’ with your own parallelization scheme
New way to organize FWI Software

Forward modeling  Migration/Demigration  Gauss-Newton Hessian  Full Hessian

F  oppDF  oppHGN  oppH

PDEfunc_dist

PDEfunc

Linear solve

C-based MVP

Abstract linear solver

Multithreaded Mat-vec multiply

PDE-related quantities
Parallel version

PDE-related quantities
Serial version

Modeling matrix: multiplication/division
New way to organize FWI Software

Abstract linear solver

Multithreaded Mat-vec multiply

C-based MVP

Linearsolve

opAbstractMatrix

PDEfunc

PDEfunc_dist

misfit_setup

FWI objective setup

PDE-related quantities
Parallel version

PDE-related quantities
Serial version

Modeling matrix: multiplication/division
misfit_setup

Constructs function handle for objective
  - velocity subsampling
  - frequency slice distribution

Batch mode interface to the objective
  - stochastic inversion algorithm can specify which source indices to use

Fancy wrapper around PDEfunc_dist
PDEopts

Options for specifying

- PDE stencil
- PML width/layout
- preconditioner
- source/receiver interpolation
- source estimation
- ...

30
Taylor error test

\[ f(m + h\delta m) - f(m) = O(h) \]

\[ f(m + h\delta m) - f(m) - h\nabla f(m)^T \delta m = O(h^2) \]

\[ f(m + h\delta m) - f(m) - h\nabla f(m)^T \delta m - \frac{h^2}{2} \delta m^T \nabla^2 f(m) \delta m = O(h^3) \]
## Adjoint Test

<table>
<thead>
<tr>
<th></th>
<th>( \langle Ax, y \rangle )</th>
<th>( \langle x, A^H y \rangle )</th>
<th>Relative Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Helmholtz system matrix</strong></td>
<td>(1.903020 + 2.087502i \cdot 10^1)</td>
<td>(1.903020 + 2.087502i \cdot 10^1)</td>
<td>(1.51 \cdot 10^{-15})</td>
</tr>
<tr>
<td><strong>Jacobian</strong></td>
<td>(-6.204229 \cdot 10^{-2})</td>
<td>(-6.204229 \cdot 10^{-2})</td>
<td>(6.8525 \cdot 10^{-10})</td>
</tr>
<tr>
<td><strong>Hessian</strong></td>
<td>(-5.842717 \cdot 10^{-3})</td>
<td>(-5.842717 \cdot 10^{-3})</td>
<td>(7.9767 \cdot 10^{-11})</td>
</tr>
</tbody>
</table>
Results
Algorithm

\[
\min_m \frac{1}{N_s} \sum_{i=1}^{N_s} f_i(m)
\]

s.t. \( m_L \leq m \leq m_U \)

- \( m \) - discrete model vector
- \( m_L, m_U \) - point-wise model bounds (water layer + constant min/max velocities)

\[
f_i(m) = \frac{1}{2} \| P_r H(m)^{-1} q_i - d_i \|_2^2 \] - per-shot misfit function

- \( P_r \) - receiver restriction operator
- \( H(m) u_i = q_i \) - discrete Helmholtz system for shot \( i \)
- \( d_i \) - measured data for shot
We have too many shots to process at once
- Can process $p$ shots at a time when we have $p$ Matlab workers

Typically $N_s \gg p$
Algorithm

\[ m_k = \arg \min_m \frac{1}{|I_k|} \sum_{i \in I_k} f_i(m) \]

s.t. \[ m_L \leq m \leq m_U \]

At the \( k \)th iteration, randomly draw a subset of sources \( I_k \subset \{1, \ldots, N_s\} \) with \( |I_k| = p \)

Approximately solve the above problem with constrained LBFGS or spectral projected gradient

Repeat for \( T \) iterations
Algorithm

Inner subproblem
- solved with $\frac{N_e}{p}$ function evaluations
- each subproblem is equivalent to one pass over the full data

We use three outer iterations
- equivalent to three gradient steps with all the shots
3D FWI Example

Overthrust model

- 20 km x 20 km x 4.6 km - 50 m spacing, 500m water layer
- 50 x 50 sources, 200m spacing - 2500 shots
- 401 x 401 receivers, 50m spacing
- 3Hz - 6Hz frequency range, single freq. inverted at a time
Computational Environment

SENAI Yemoja cluster
- 100 nodes, 128 GB RAM each, 20-core processors
- 400 Parallel Matlab workers (4 per node), Helmholtz MVP uses 5 threads - full core utilization
$z=1000\text{m}$ slice

![True model](image1)

![Initial model](image2)
$z=1000\text{m slice}$

**True model**

**Stochastic LBFGS**
z=2000m slice

True model

Initial model
$z=2000\text{m slice}$

**True model**

**Stochastic LBFGS**
x=12.5km slice

True model

Initial model
x=12.5km slice

True model

Stochastic LBFGS
x=17.5km slice

True model

Initial model
x=17.5km slice

True model

Stochastic LBFGS
$y=10\text{km slice}$

**True model**

**Initial model**
y=10km slice

True model

Stochastic LBFGS
Summary

Performance and correctness don’t have to be mutually exclusive

• Design software in a modular, hierarchical way yields benefits of both

Modularity -> flexibility

• Very easy to swap out modules (PDE discretizations, preconditioners) without changing code
Summary

Modularity -> Easier to test
  • Easier to test -> easier to get right

We can design code that is *demonstrably* correct
  • Reduce scope of potential problems in FWI
Summary

Right abstractions for FWI ->
- ease of use
- computationally efficient
- flexible
- easy to extend, understand, optimize
- can prototype algorithms in 2D, run immediately in 3D
Acknowledgements

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

The authors wish to acknowledge the SENAI CIMATEC Supercomputing Center for Industrial Innovation, with support from BG Brasil, Shell, and the Brazilian Authority for Oil, Gas and Biofuels (ANP), for the provision and operation of computational facilities and the commitment to invest in Research & Development.
Thank you for your attention