Frequency Domain Full Waveform Inversion with JAX

Team by:

Emilio Ochoa, Naomi Guevara, Liumei Ma

ECE 485 Inverse Problem in Imaging - Final Project

What's Full Wave Inversión (FWI)?

- Originated from seismic imaging, aims to reconstruct physical properties of the medium, such as speed of sound (SoS) and density.

- Can be used in any wave-based imaging: Geophysics (Oil&Gas Exploration, Earthquake Seismology), Medical Imaging (Ultrasound Tomography, Elastography), Materials Science, etc.

- Considering full wavefield information to produce high-resolution image

Inverse problems

- Non-linear and ill-posed. Solved through iterative optimization

- Starting from an initial estimation, generate synthetic waveforms using wave equation, iteratively update the model to minimize the difference between simulated and observated wavefiles

Why JAX?

"Jax is a library for array-oriented numerical computation, with automatic differentiation and JIT compilation to enable high-performance machine learning research."

Automatic Differentiation (Autograd) — efficient for gradient-based optimization

Just-In-Time (JIT) — uses XLA to accelerate simulations and evaluation

- **Support GPU, TPU** ---- (only for Linux)
- **Efficient Vectorization** ---- parallelize computations with batches of inputs
- **Support Custom Gradient** ---- for complex operation(like wave physics, analytical gradients, adjoint-state formulations)
- **Numpy-like syntax, support Scipy**

Project Aim

- Reconstruct ultrasound image from data recorded from 256 elements ring-shape array transducers.
  - Reconstructed with **one frequency** so far. The result can be used as initial guess for higher frequency reconstruction. Implement frequency sweep in the future.

- Implement JAX version reconstruction algorithm.

- Evaluation: Compare JAX and Matlab algorithm
  - Run time, image quality, code adaptivity

Methodology

- **Initialization**
  - Initialize SoS map, transducer geometry, included elements

- **Iteratively Update**
  - Solve wavefield using Helmhotz function (from SoS map estimation)
  - Forward error (between simulated and recorded wavefield) as virtual source to create adjoint wavefield
  - Back project adjoint wavefield to calculate **gradient**.
  - Calculate **momentum** and then update **search direction**.
  - Forward project the virtual source to the search direction to find the perturbed wavefield.
  - Calculate the step size from the perturbed wavefield. And finally update the slowness (then SoS) according to the step size and search direction.

Implementation

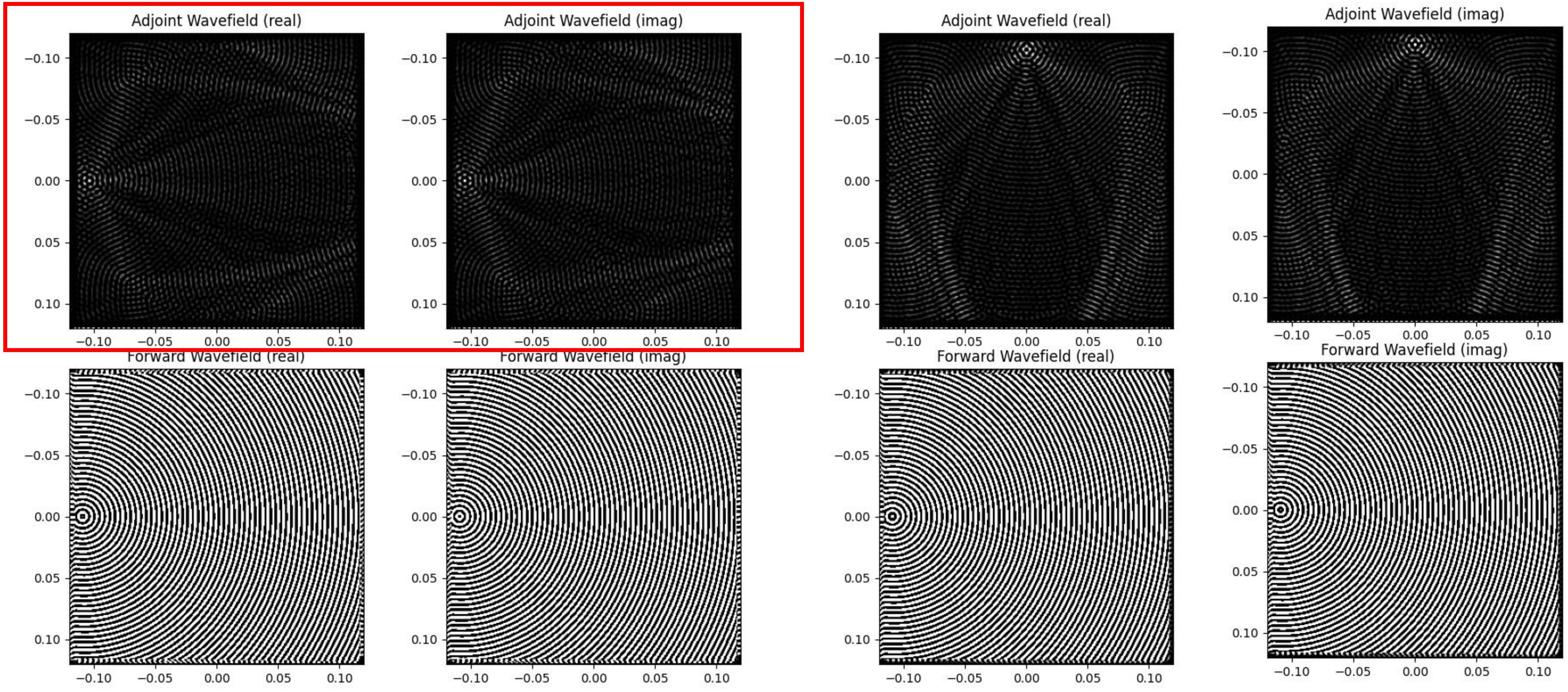
- Matlab Indexation
- Hemholtz equation
- Nonlinear CG

Matlab Indexation

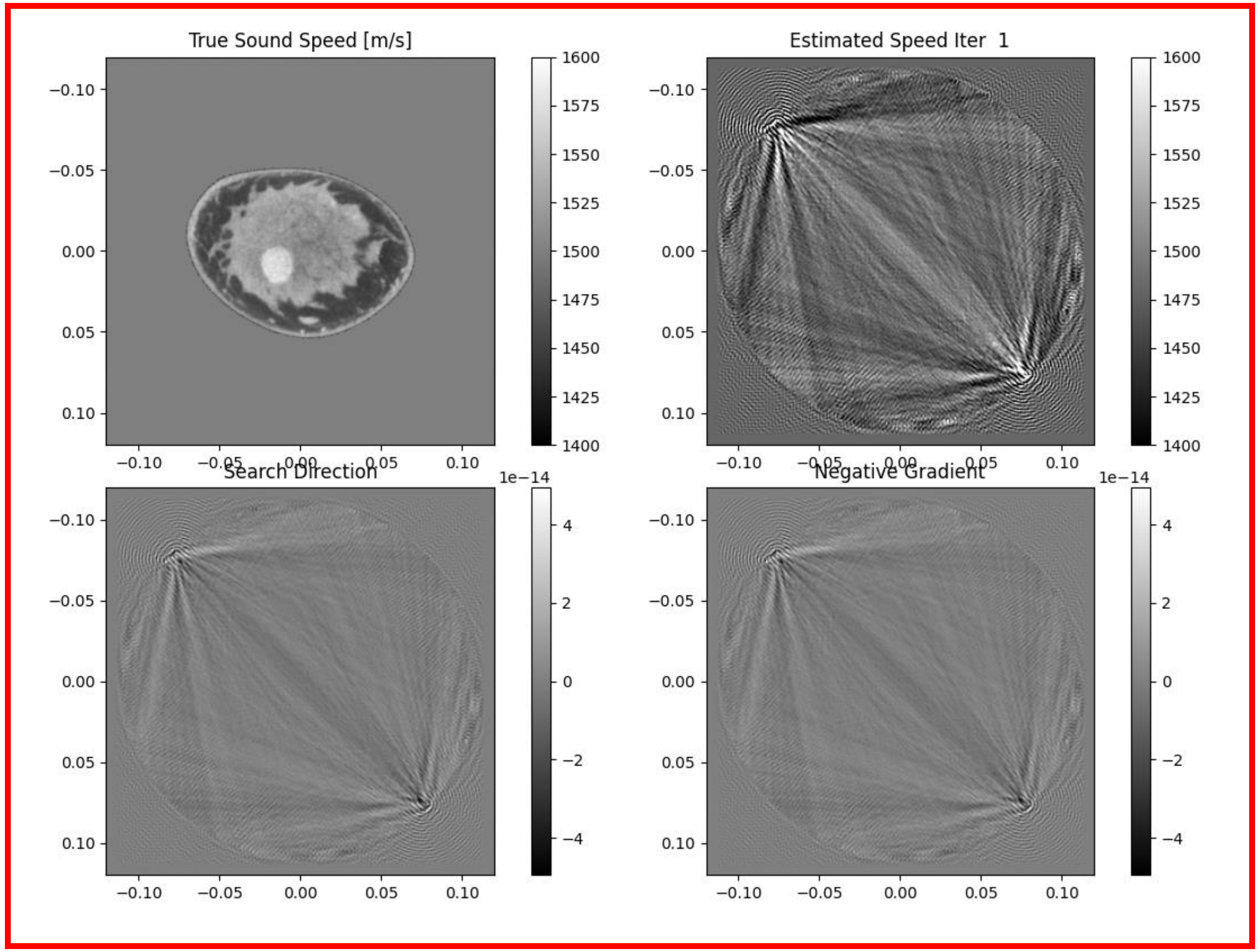
```python
# grid
dxi = 0.8e-3
xmax = 120e-3
xi = jnp.arange(-xmax, xmax + dxi, dxi)
yi = xi.copy()
Nyi, Nxi = yi.size, xi.size

# nearest-neighbor search for element positions
tree_x = cKDTree(xi.reshape(-1, 1))
x_idx = tree_x.query(x_circ.reshape(-1, 1))[1]
tree_y = cKDTree(yi.reshape(-1, 1))
y_idx = tree_y.query(y_circ.reshape(-1, 1))[1]

# MATLAB-style linear index (column-major, zero-based)
# ind_matlab = x_idx * Nyi + y_idx
ind_matlab = y_idx * Nyi + x_idx

xc = x_circ.ravel()  # shape (M,)
yc = y_circ.ravel()  # shape (M,)

x_idx = jnp.argmin(jnp.abs(xi[None, :] - xc[:, None]), axis=1)
y_idx = jnp.argmin(jnp.abs(yi[None, :] - yc[:, None]), axis=1)

ind_matlab = x_idx * Nxi + y_idx  # Row majo
```

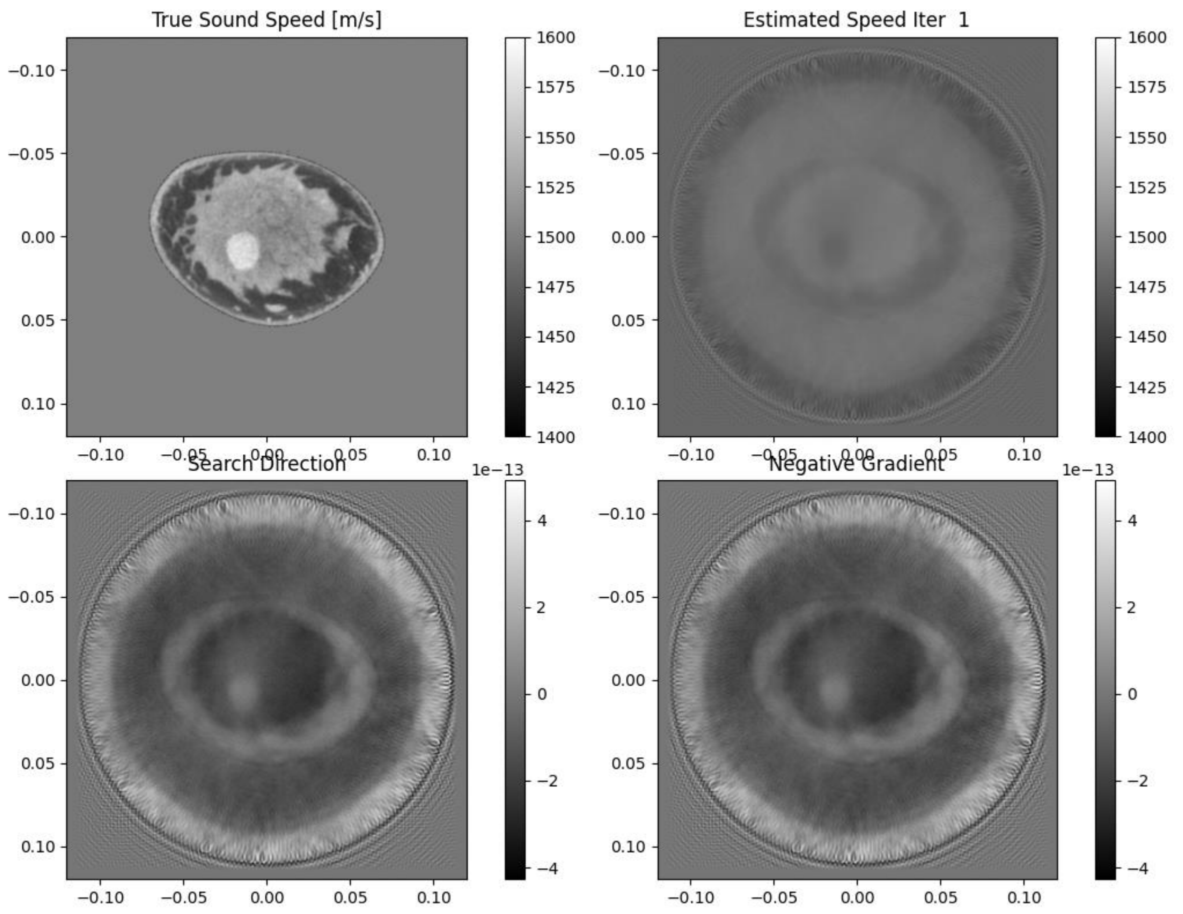
```python
# ind_matlab = np.ravel_multi_index((y_idx, x_idx), dims=(Nyi, Nxi), order="c")
# build source array (one hot per tx)
SRC = jnp.zeros((Nyi, Nxi, tx_include.size), dtype=jnp.complex64)
for i, t in enumerate(tx_include):
    SRC = SRC.at[y_idx[t], x_idx[t], i].set(1.0)

# -------------------------
# 3) Build explicit_indices
# -------------------------
explicit_indices = []
mask_indices = []
for t in tx_include:
    mask = elemInclude[t, :].nonzero()[0]
    explicit_indices.append(mask)
    mask_indices.append(mask)
mask_indices = jnp.stack([jnp.array(m, dtype=int) for m in mask_indices], axis=0)
```

Matlab Indexation



Matlab Indexation





Hemholtz equation

```
% Generate Left-Hand Side of Sparse Array
HelmholtzEqn = sparse(rows, cols, vals, Nx*Ny, Nx*Ny);
% Solve the Helmholtz Equation - Brute-force CPU solution of linear system
if adjoint
    sol = (HelmholtzEqn')\reshape(src,[Nx*Ny, numel(src)/(Nx*Ny)]);
else
    sol = HelmholtzEqn\reshape(src,[Nx*Ny, numel(src)/(Nx*Ny)]);
end
wvfield = reshape(sol, size(src));

end
```

```
H_use = jax.lax.cond(
    adjoint,
    lambda H: jsparse.BCOO(
        (jnp.conj(H.transpose().data), H.transpose().indices), shape=H.shape
    ),
    lambda H: H,
    H_bcoo,
)

H_use = jsparse.BCSR.from_bcoo(H_use)

# reshape src to Nx*Ny and -1 to get the right shape
rhs = jnp.reshape(src, (Nx * Ny, -1))
rhs = jnp.array(rhs, dtype=jnp.complex64)

data, indices, indptr = H_use.data, H_use.indices, H_use.indptr

start = time.time()

sol = jnp.stack(
    [spsolve(data, indices, indptr, rhs[:, i]) for i in range(rhs.shape[1])], axis=1
)
```

Jax.experimental.spsolve: 237 seconds. **Too slow!**

Hemholtz equation

```matlab
% Generate Left-Hand Side of Sparse Array
HelmholtzEqn = sparse(rows, cols, vals, Nx*Ny, Nx*Ny);
% Solve the Helmholtz Equation - Brute-force CPU solution of linear system
if adjoint
    sol = (HelmholtzEqn')\reshape(src,[Nx*Ny, numel(src)/(Nx*Ny)]);
else
    sol = HelmholtzEqn\reshape(src,[Nx*Ny, numel(src)/(Nx*Ny)]);
end
wvfield = reshape(sol, size(src));

end
```

Scipy spsolve: 4.68e-05 seconds

```python
def scipy_solve(data, indices, indptr, rhs_np, shape):
    NxNy = shape[0]
    mat = csr_matrix((data, indices, indptr), shape=(NxNy, NxNy))
    return spsolve_cpu(mat, rhs_np)
```

```python
sol = jax.pure_callback(
    scipy_solve,
    jax.ShapeDtypeStruct((Nx * Ny, rhs.shape[1]), dtype=jnp.complex64),
    data,
    indices,
    indptr,
    rhs,
    (Nx * Ny, Nx * Ny),
)
```

Jax.pure_callback --- Avoid JIX

Non-linear CG: Gradient/Backprojection

```matlab
for iter = 1:Niter
    % Step 1: Calculate Gradient/Backprojection
    % (1A) Solve Forward Helmholtz Equation (H is Helmholtz matrix and u is the wavefield)
    tic; WVFIELD = solveHelmholtz(xi, yi, VEL_ESTIM, SRC, f, a0, L_PML, false);

    % (1B) Estimate Forward Sources and Adjust Simulated Fields Accordingly
    SRC_ESTIM = zeros(1,1,numel(tx_include));
    for tx_elmt_idx = 1:numel(tx_include)
        WVFIELD_elmt = WVFIELD(:,:,tx_elmt_idx);

        REC_SIM = WVFIELD_elmt(ind(elemInclude(tx_include(tx_elmt_idx),:)));
        REC = REC_DATA(tx_elmt_idx, elemInclude(tx_include(tx_elmt_idx),:));
        SRC_ESTIM(tx_elmt_idx) = (REC_SIM(:)'*REC(:)) / ...
            (REC_SIM(:)'*REC_SIM(:)); % Source Estimate

    end
```

```python
(VEL_F, _, sd_F, grad_F, ADJ_WV, WV), _ = jax.lax.scan(
    body_fun, (VEL, SLOW, sd, gprev, ADJ_WV, WV), jnp.arange(Niter)
)
return VEL_F, sd_F, grad_F, ADJ_WV, WV
```

```python
def body_fun(state, it):
    VEL, SLOW, sd, gprev, ADJ_WV, WV = state
    t3 = time.time()

    # 1a) forward Helmholtz
    WV = solve_helmholtz(xi, yi, VEL, SRC, f, a0, L_PML, False)

    # 1b) estimate source strengths
    SRC_EST = jnp.zeros((len(tx_include),), dtype=jnp.complex64)
    for t in range(len(tx_include)):
        W = WV[:, :, t]

        flat = W.ravel(order="F")
        mask = mask_indices[t]  # array de 193 índices 0-based
        REC_SIM = flat[ind_matlab[mask]]  # == REC_SIM(:)
        REC = REC_DATA[t, mask]  # == REC(:)
        SRC_EST = SRC_EST.at[t].set(estimate_src_strength(REC_SIM, REC))

    WV = WV * SRC_EST[jnp.newaxis, jnp.newaxis, :]
```

Non-linear CG : Gradient/Backprojection

```matlab
% (1C) Build Adjoint Sources - Based on Errors
ADJ_SRC = zeros(Nyi, Nxi, numel(tx_include));
REC_SIM = zeros(numel(tx_include), numElements);
for tx_elmt_idx = 1:numel(tx_include)
    WVFIELD_elmt = WVFIELD(:,:,tx_elmt_idx);
    REC_SIM(tx_elmt_idx,elemInclude(tx_include(tx_elmt_idx),:)) = ...
        WVFIELD_elmt(ind(elemInclude(tx_include(tx_elmt_idx),:)));
    ADJ_SRC_elmt = zeros(Nyi, Nxi);
    ADJ_SRC_elmt(ind(elemInclude(tx_include(tx_elmt_idx),:))) = ...
        REC_SIM(tx_elmt_idx, elemInclude(tx_include(tx_elmt_idx),:)) - ...
        REC_DATA(tx_elmt_idx, elemInclude(tx_include(tx_elmt_idx),:));
    ADJ_SRC(:,:,tx_elmt_idx) = ADJ_SRC_elmt;

end
```

```python
# 1c) build adjoint sources
# Pre-allocate exactly like MATLAB
ADJ_SRC = jnp.zeros((Nyi, Nxi, len(tx_include)), dtype=jnp.complex64)
REC_SIM = jnp.zeros((len(tx_include), numElements), dtype=jnp.complex64)

for t in range(len(tx_include)):
    # 1) Flatten in Fortran (column-major) order
    W = WV[:, :, t]
    flat_W = W.ravel(order="F")
    mask = mask_indices[t]  # 0-based receiver indices

    # 2) Gather simulated data into REC_SIM exactly like MATLAB
    REC_SIM = REC_SIM.at[t, mask].set(flat_W[ind_matlab[mask]])

    # 3) Compute difference
    diff = REC_SIM[t, mask] - REC_DATA[t, mask]

    # 4) Build adj_src_elmt and use the same 'ind' indexing inside
    #    just like MATLAB's
    adj_src_elmt = jnp.zeros((Nyi, Nxi), dtype=jnp.complex64)
    flat_adj = adj_src_elmt.ravel(order="F")
    flat_adj = flat_adj.at[ind_matlab[mask]].set(diff)
    adj_src_elmt = flat_adj.reshape((Nyi, Nxi), order="F")

    # 5) Store into the 3D ADJ_SRC volume
    ADJ_SRC = ADJ_SRC.at[:, :, t].set(adj_src_elmt)
```

```matlab
% (1D) Calculate Virtual Source [dH/ds u] where s is slowness
VIRT_SRC = ((2*(2*pi*f).^2).*SLOW_ESTIM).*WVFIELD;
% (1E) Backproject Error (Gradient = Backprojection)
ADJ_WVFIELD = solveHelmholtz(xi, yi, VEL_ESTIM, ADJ_SRC, f, a0, L_PML, true);


BACKPROJ = -real(conj(VIRT_SRC).*ADJ_WVFIELD);
gradient_img = sum(BACKPROJ,3);
```

```python
# 1d) virtual source
VIRT = (2 * (2 * jnp.pi * f) ** 2) * SLOW[:, :, None] * WV

# 1e) backpropagate

# because of the error the element is in other part
ADJ_WV = solve_helmholtz(xi, yi, VEL, ADJ_SRC, f, a0, L_PML, True)
BACK = -jnp.real(jnp.conj(VIRT) * ADJ_WV)
grad = jnp.sum(BACK, axis=2)
```

Non-linear CG: New CG and SD

```matlab
case 4 % Hestenes-Stiefel
    beta = (gradient_img(:)'*...
        (gradient_img(:)-gradient_img_prev(:))) / ...
        (search_dir(:)'*(gradient_img(:)-gradient_img_prev(:)));

% (2B) Search Direction Based on Conjugate Gradient Momentum
search_dir = beta*search_dir-gradient_img;
```

```python
# 2) Conjugate-gradient update (Hestenes-Stiefel)
dg = grad - gprev

raw_beta = jnp.vdot(grad.ravel(order="F"), dg.ravel(order="F")) / (
    jnp.vdot(sd.ravel(order="F"), dg.ravel(order="F"))  # + 1e-12
)

beta = jax.lax.cond((it == 0), lambda _: 0.0, lambda _: raw_beta, operand=None)

sd_new = beta * sd - grad
```

Non-linear CG: Forward projection

```matlab
% Step 3: Compute Forward Projection of Current Search Direction
PERTURBED_WVFIELD = solveHelmholtz(xi, yi, VEL_ESTIM, ...
    -VIRT_SRC.*search_dir, f, a0, L_PML, false);
dREC_SIM = zeros(numel(tx_include), numElements);
for tx_elmt_idx = 1:numel(tx_include)
    % Forward Projection of Search Direction Image
    PERTURBED_WVFIELD_elmt = PERTURBED_WVFIELD(:,:,tx_elmt_idx);
    dREC_SIM(tx_elmt_idx,elemInclude(tx_include(tx_elmt_idx),:)) = ...
        PERTURBED_WVFIELD_elmt(ind(elemInclude(tx_include(tx_elmt_idx),:)));
end
```

```python
# 3) forward project search direction
PERT = solve_helmholtz(
    xi, yi, VEL, -VIRT * sd_new[:, :, None], f, a0, L_PML, False
)

# 4) line search
dREC = jnp.zeros((len(tx_include), numElements), dtype=jnp.complex64)

for t in range(len(tx_include)):
    # flatten in column-major order
    Wp = PERT[:, :, t].ravel(order="F")
    # restrict to the included receivers
    mask = mask_indices[t]  # 0-based indices of included elements
    vals = Wp[ind_matlab[mask]]  # simulated search-direction data

    # write only into those positions
    dREC = dREC.at[t, mask].set(vals)
```

Non-linear CG

```
% Step 4: Perform a Linear Approximation of Exact Line Search
switch stepSizeCalculation
    case 1 % Not Involving Gradient Nor Search Direction
        stepSize = real(dREC_SIM(:)'*(REC_DATA(:)-REC_SIM(:))) / ...
            (dREC_SIM(:)'*dREC_SIM(:));
        % REVIEW STEP SIZE CALC TO EXPLAIN WHY REAL() IS USED HERE
    case 2 % Involving Gradient BUT NOT Search Direction
        stepSize = (gradient_img(:)'*gradient_img(:)) / ...
            (dREC_SIM(:)'*dREC_SIM(:));
    case 3 % Involving Gradient AND Search Direction
        stepSize = -(gradient_img(:)'*search_dir(:)) / ...
            (dREC_SIM(:)'*dREC_SIM(:));

end
SLOW_ESTIM = SLOW_ESTIM + stepSize * search_dir;
VEL_ESTIM = 1./real(SLOW_ESTIM); % Wave Velocity Estimate [m/s]
```

```python
@jit
def compute_step_size(dREC, REC_DATA, REC_SIM, SLOW, sd_new):
    num = jnp.real(
        jnp.vdot(dREC.ravel(order="F"), (REC_DATA - REC_SIM).ravel(order="F"))
    )
    den = jnp.real(jnp.vdot(dREC.ravel(order="F"), dREC.ravel(order="F")))
    step = num / den  # + 1e-12)
    SLOW_new = SLOW + step * sd_new
    VEL_new = 1.0 / SLOW_new

    return VEL_new, SLOW_new
```

Vectorization example

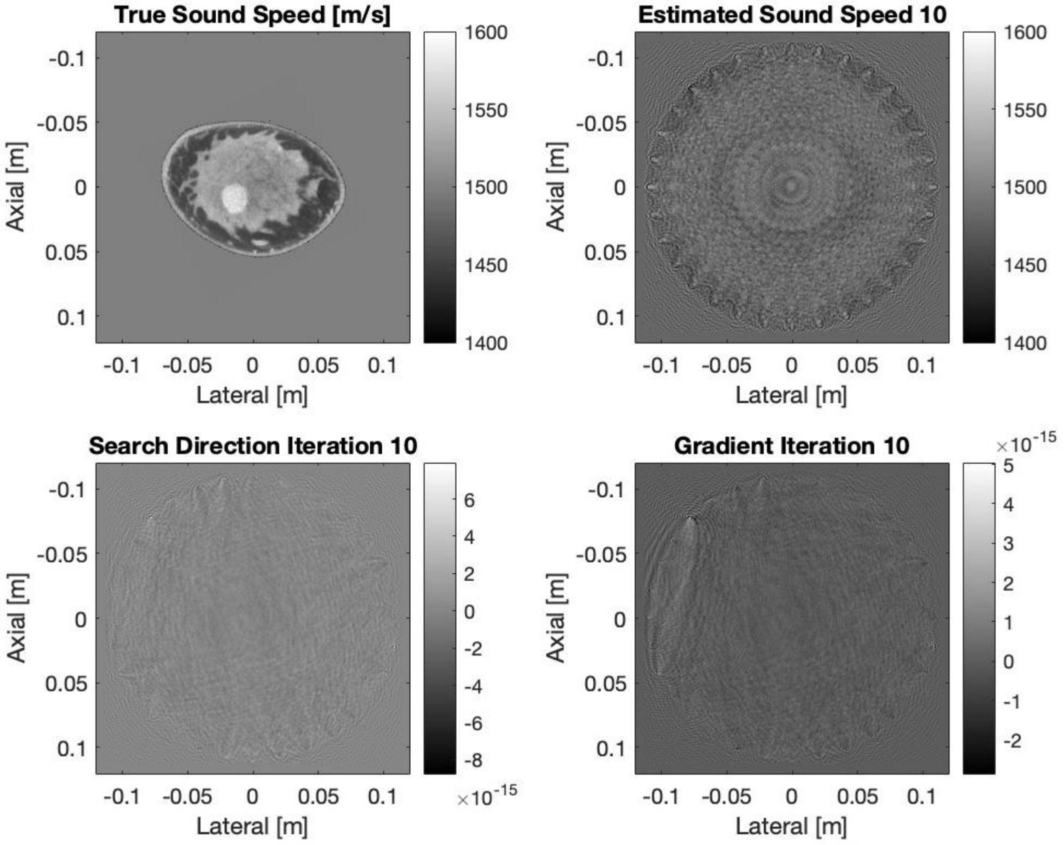
```python
# 1b) estimate source strengths
SRC_EST = jnp.zeros((len(tx_include),), dtype=jnp.complex64)
for t in range(len(tx_include)):
    W = WV[:, :, t]

    flat = W.ravel(order="F")
    mask = mask_indices[t]  # array de 193 índices 0-based
    REC_SIM = flat[ind_matlab[mask]]  # == REC_SIM(:)
    REC = REC_DATA[t, mask]  # == REC(:)
    SRC_EST = SRC_EST.at[t].set(estimate_src_strength(REC_SIM, REC))

WV = WV * SRC_EST[jnp.newaxis, jnp.newaxis, :]
```

```python
@jit
def estimate_src_strength(REC_SIM, REC):
    return jnp.vdot(REC_SIM.ravel(order="F"), REC.ravel(order="F")) / (
        jnp.vdot(REC_SIM.ravel(order="F"), REC_SIM.ravel(order="F"))
    )

estimate_src_strength_batched = vmap(estimate_src_strength, in_axes=(0, 0))
```

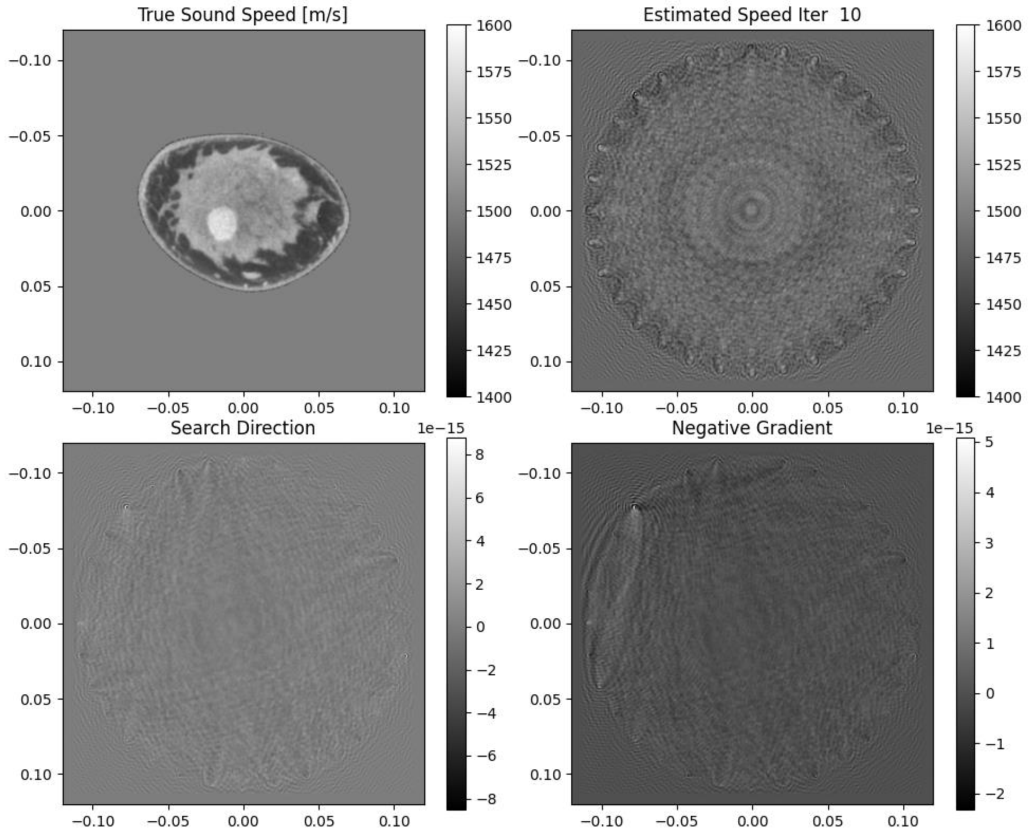
```python
# --- Vectorization ---
N1, N2, Nt = WV.shape
Nflat = N1 * N2

# 1b) estimate source strengths
flat_WV = jnp.reshape(jnp.transpose(WV, (1, 0, 2)), (N1 * N2, Nt))
global_inds = jnp.take(ind_matlab, mask_indices)
rec_sim = jnp.take_along_axis(flat_WV.T, global_inds, axis=1)
rec = jnp.take_along_axis(REC_DATA, mask_indices, axis=1)
SRC_EST = estimate_src_strength_batched(rec_sim, rec)  # shape (Nt,)

# 5) Update WV with the estimated source strengths
WV = WV * SRC_EST[None, None, :]
```

Results: Reconstruction for 32 elements

MATLAB – 10 iterations with 7 element excluded

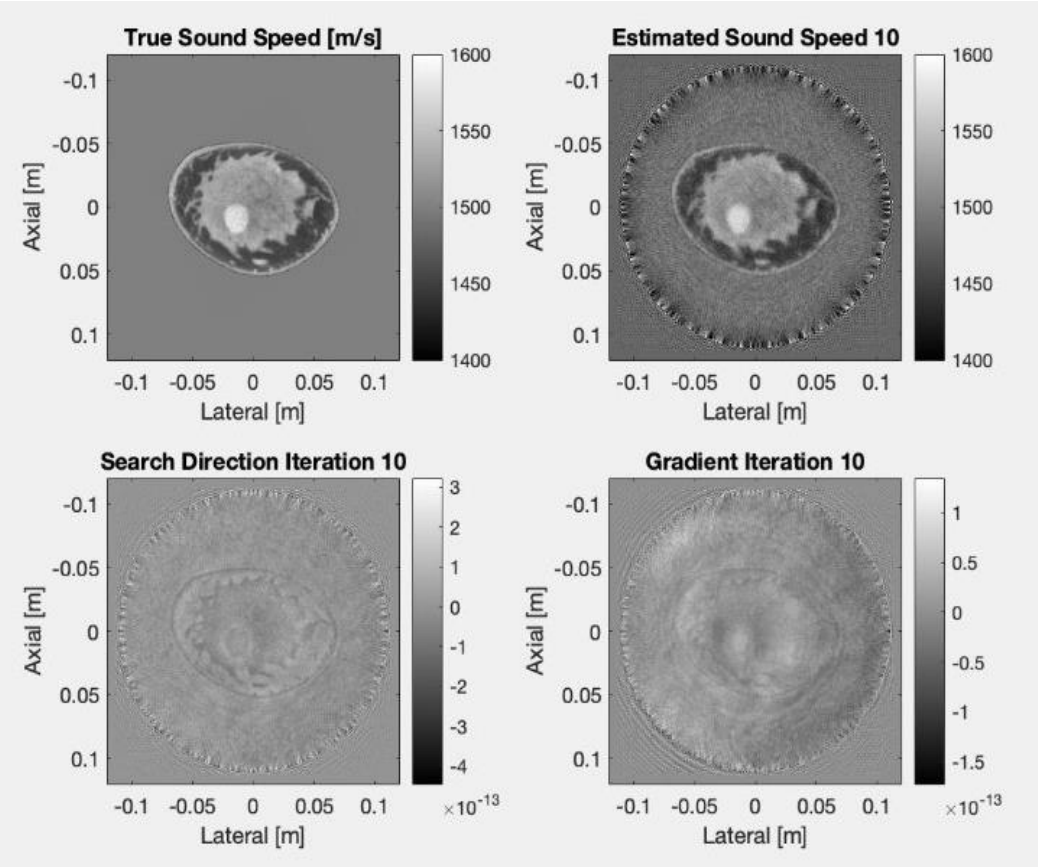


Jax – 10 iterations with 7 element excluded

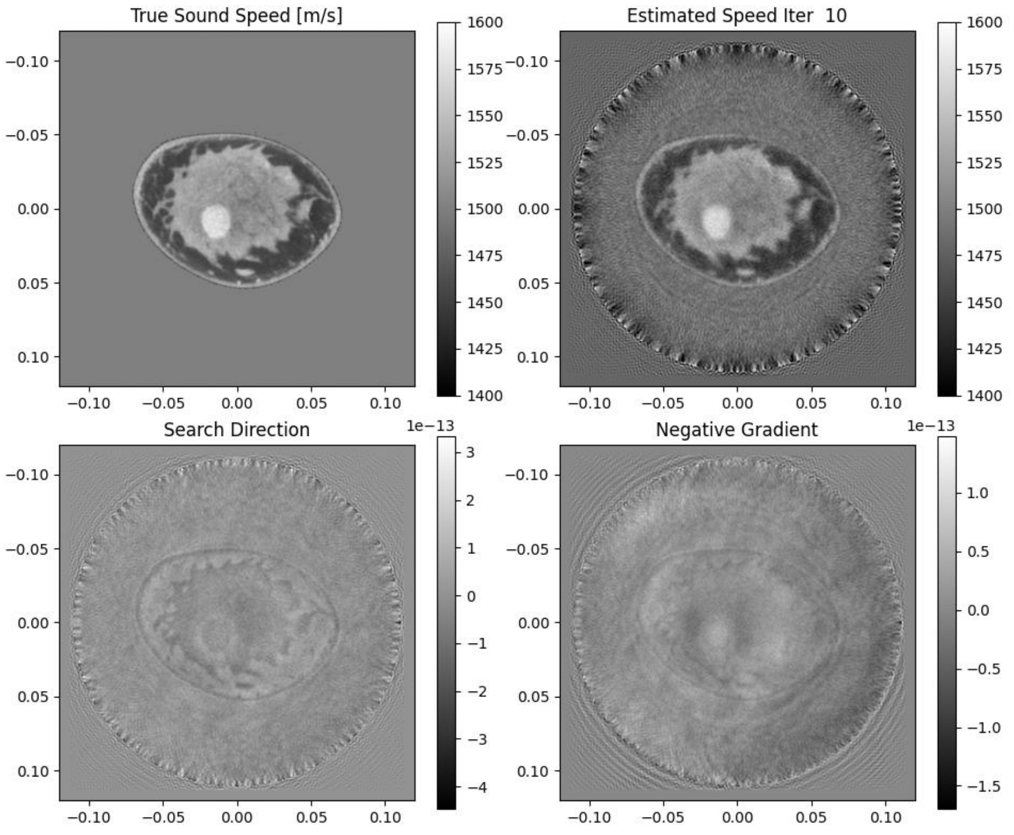


Results: Reconstruction for 256 elements

MATLAB – 10 iterations with 63 element excluded



Jax – 10 iterations with 63 element excluded



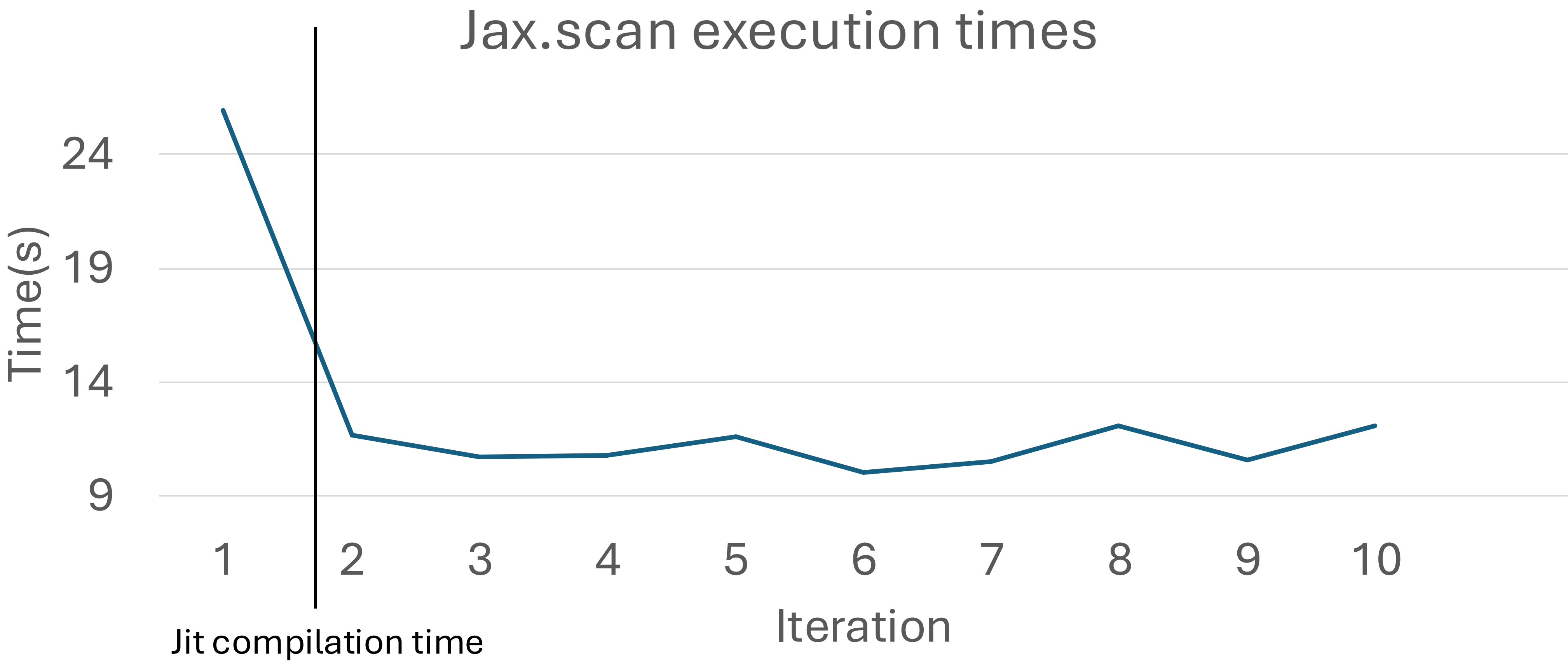
Results: Time performance

On MacBook M4 Pro 24Gb RAM

10 iterations	CG (s)	CG Vectorized (s)
MATLAB	52.42	
JAX – without jit (hemholtz)	134.27	105.59
JAX – jit (hemholtz)	124.49	104.81
spsolve from jax (hemholtz)	237	
spsolve from scipy (hemholtz)	4.68 e-5	

Results: Time performance

On MacBook M4 Pro 24Gb RAM



Results: Time performance

		Iteration	1	2	3	4	5	6	7	8	9	10
	Initialization		0.07	0	0	0	0	0	0	0	0	0
Scan	Compilation		24.64	10.36	9.43	9.47	10.29	8.74	9.25	10.78	9.32	10.82
	body_fun	Hemholtz	4.48-05	4.48-05	4.48-05	4.48-05	4.48-05	4.48-05	4.48-05	4.48-05	4.48-05	4.48-05
		Total body_fun	1.28	1.28	1.28	1.28	1.28	1.28	1.28	1.28	1.28	1.28
	Total scan		25.92	11.64	10.71	10.75	11.57	10.02	10.53	12.06	10.6	12.1
	Total time		25.99	37.56	48.27	59.02	70.59	80.61	91.14	103.2	113.8	125.9

Challenges

- Hard to debug, since values are always abstract tracers and don't have real value.
  - Can not use normal python control flow (if, switch). Need to use the wrapped function ***(jax.lax.cond, jax.lax.switch)***
- Does not allow in-place update. ***(use x=x.at[i].set(v))***
- Need to use ***jax.lax.scan*** as 'for loop'. Managing state and memory inside the loop is more complex
- Must pass the static shape and dtype
  - Not allow dynamic indexing or using the shape of a variable to control logic

Future Steps

- Implementation of different gradient and search direction methods
- Implementation of reconstruction with multiple frequencies

Conclusion

- We successfully implemented **Frequency Domain Full Waveform Inversion (FWI)** using **JAX**, reproducing the core logic of the MATLAB version.
- **Reconstruction quality** was consistent between MATLAB and JAX implementations, validating the correctness of our JAX-based pipeline across multiple configurations (32 and 256 elements, different element exclusions)
- **Time performance analysis** revealed:
  - Significant overhead in JAX due to JIT compilation and use of lax.scan.
  - Solve_helmholtz calls were **efficiently accelerated** using scipy.sparse.linalg.spsolve (≈47 μs vs. 237 s for jax.experimental.sparse.spsolve).
  - Vectorized implementations further reduced compute time and improved memory handling.
- **Key advantages of JAX**:
  - Transparent support for gradients and batched computations.
  - Modular and differentiable wave simulation suitable for inverse problems.
- **Challenges** included managing static shapes, memory inside lax.scan, and limited Python-native control flow.