Recent improvements to the 2/3—D imaging & inversion algorithms in the SLIM software release

Curt Da Silva & Bas Peters

SLIM
University of British Columbia

2D Code

Why?

Old code

- a lot of setup code duplicated across functions
  - only thing that changed was the actual PDE-dependent quantities

- parallelization + computation tied together
  - hard to debug
  - can’t easily extend to parallelizing over sources/sources + frequencies

Why?

Old code

- lots of manual tweaking to set up objective functions, GN Hessian
  - needless recomputation of wavefields

- lots of the same functionality being duplicated across functions
  - hard to maintain + integrate code changes over time

Old code structure

Helm2D.m - generates Helmholtz matrix

F.m - forward modeling kernel

DF.m - Jacobian + Jacobian adjoint of F.m

F.m - original

```
% comp. grid
ot = model.o-model.nb.*model.d;
dt = model.d;
nt = model.n+2*model.nb;
[zt,xt] = odn2grid(ot,dt,nt);

% data size
nsrc   = size(Q,2);
nrec   = length(model.zrec)*length(model.xrec);
nfreq  = length(model.freq);

% define wavelet
w = exp(1i*2*pi*model.freq*model.t0);
if model.f0
    % Ricker wavelet with peak-frequency model.f0
    w = (model.freq).^2.*exp(-(model.freq/model.f0).^2).*w;
end

% mapping from source/receiver/physical grid to comp. grid
Pr = opKron(opLInterp1D(xt,model.xrec),opLInterp1D(zt,model.zrec));
Ps = opKron(opLInterp1D(xt,model.xsrc),opLInterp1D(zt,model.zsrc));
Px = opKron(opExtension(model.n(2),model.nb(2)),opExtension(model.n(1),model.nb(1)));

% model parameter: slowness [s/m] on computational grid.
nu = 1e-3*Px*sqrt(m);

% distribute frequencies according to standard distribution
freq = distributed(model.freq);
w    = distributed(w);

% check source matrix input
if (size(Q,3)==1)&&(isdistributed(Q))
    Q = gather(Q);
end

spmd
    codistr  = codistributor1d(2,[],[nsrc*nrec,nfreq]);
    freqloc  = getLocalPart(freq);
    wloc     = getLocalPart(w);
    nfreqloc = length(freqloc);
    Dloc     = zeros(nrec*nsrc,nfreqloc);
    if size(Q,3)==1
        for k = 1:nfreqloc
            Hk  = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            Uk  = Hk\(wloc(k)*(Ps'*Q));
            Dloc(:,k) = vec(Pr*Uk);
        end
    else
        Qloc = getLocalPart(Q);
        for k = 1:nfreqloc
            Hk  = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            Uk  = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
            Dloc(:,k) = vec(Pr*Uk);
        end
    end
    D = codistributed.build(Dloc,codistr,'noCommunication');
end

% vectorize output, gather if needed
D = vec(D);

% construct pSPOT operator
J = oppDF(m,Q,model);
```


```
% comp. grid
ot = model.o-model.nb.*model.d;
dt = model.d;
nt = model.n+2*model.nb;
[zt,xt] = odn2grid(ot,dt,nt);

% data size
nsrc   = size(Q,2);
nrec   = length(model.zrec)*length(model.xrec);
nfreq  = length(model.freq);

% define wavelet
w = exp(1i*2*pi*model.freq*model.t0);
if model.f0
    % Ricker wavelet with peak-frequency model.f0
    w = (model.freq).^2.*exp(-(model.freq/model.f0).^2).*w;
end

% mapping from source/receiver/physical grid to comp. grid
Pr = opKron(opLInterp1D(xt,model.xrec),opLInterp1D(zt,model.zrec));
Ps = opKron(opLInterp1D(xt,model.xsrc),opLInterp1D(zt,model.zsrc));
Px = opKron(opExtension(model.n(2),model.nb(2)),opExtension(model.n(1),model.nb(

% model parameter: slowness [s/m] on computational grid.
nu = 1e-3*Px*sqrt(m);

% distribute frequencies according to standard distribution
freq = distributed(model.freq);
w    = distributed(w);

% check source matrix input
if (size(Q,3)==1)&&(isdistributed(Q))
    Q = gather(Q);
end
```


Setup code

- duplicated in DF.m

F.m - original

```
% comp. grid
ot = model.o-model.nb.*model.d;
dt = model.d;
nt = model.n+2*model.nb;
[zt,xt] = odn2grid(ot,dt,nt);

% data size
nsrc   = size(Q,2);
nrec   = length(model.zrec)*length(model.xrec);
nfreq  = length(model.freq);

% define wavelet
w = exp(1i*2*pi*model.freq*model.t0);
if model.f0
    % Ricker wavelet with peak-frequency model.f0
    w = (model.freq).^2.*exp(-(model.freq/model.f0).^2).*w;
end

% mapping from source/receiver/physical grid to comp. grid
Pr = opKron(opLInterp1D(xt,model.xrec),opLInterp1D(zt,model.zrec));
Ps = opKron(opLInterp1D(xt,model.xsrc),opLInterp1D(zt,model.zsrc));
Px = opKron(opExtension(model.n(2),model.nb(2)),opExtension(model.n(1),model.nb(1)));

% model parameter: slowness [s/m] on computational grid.
nu = 1e-3*Px*sqrt(m);

% distribute frequencies according to standard distribution
freq = distributed(model.freq);
w    = distributed(w);

% check source matrix input
if (size(Q,3)==1)&&(isdistributed(Q))
    Q = gather(Q);
end

spmd
    codistr  = codistributor1d(2,[],[nsrc*nrec,nfreq]);
    freqloc  = getLocalPart(freq);
    wloc     = getLocalPart(w);
    nfreqloc = length(freqloc);
    Dloc     = zeros(nrec*nsrc,nfreqloc);
    if size(Q,3)==1
        for k = 1:nfreqloc
            Hk  = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            Uk  = Hk\(wloc(k)*(Ps'*Q));
            Dloc(:,k) = vec(Pr*Uk);
        end
    else
        Qloc = getLocalPart(Q);
        for k = 1:nfreqloc
            Hk  = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            Uk  = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
            Dloc(:,k) = vec(Pr*Uk);
        end
    end
    D = codistributed.build(Dloc,codistr,'noCommunication');
end

% vectorize output, gather if needed
D = vec(D);

% construct pSPOT operator
J = oppDF(m,Q,model);
```


```
% comp. grid
ot = model.o-model.nb.*model.d;
dt = model.d;
nt = model.n+2*model.nb;
[zt,xt] = odn2grid(ot,dt,nt);

% data size
nsrc   = size(Q,2);
nrec   = length(model.zrec)*length(model.xrec);
nfreq  = length(model.freq);

% define wavelet
w = exp(1i*2*pi*model.freq*model.t0);
if model.f0
    % Ricker wavelet with peak-frequency model.f0
    w = (model.freq).^2.*exp(-(model.freq/model.f0).^2).*w;
end

% mapping from source/receiver/physical grid to comp. grid
Pr = opKron(opLInterp1D(xt,model.xrec),opLInterp1D(zt,model.zrec));
Ps = opKron(opLInterp1D(xt,model.xsrc),opLInterp1D(zt,model.zsrc));
Px = opKron(opExtension(model.n(2),model.nb(2)),opExtension(model.n(1),model.nb(

% model parameter: slowness [s/m] on computational grid.
nu = 1e-3*Px*sqrt(m);

% distribute frequencies according to standard distribution
freq = distributed(model.freq);
w    = distributed(w);

% check source matrix input
if (size(Q,3)==1)&&(isdistributed(Q))
    Q = gather(Q);
end
```


Complicated formula - do once + never again

- inhibits readability

F.m - first update

```
% comp. grid
ot = model.o-model.nb.*model.d;
dt = model.d;
nt = model.n+2*model.nb;
[zt,xt] = odn2grid(ot,dt,nt);

% data size
nsrc   = size(Q,2);
nrec   = length(model.zrec)*length(model.xrec);
nfreq  = length(model.freq);

% define wavelet
w = fwi_wavelet(model.freq,model.t0,model.f0);

% mapping from source/receiver/physical grid to comp. grid
Pr = opKron(opLInterp1D(xt,model.xrec),opLInterp1D(zt,model.zrec));
Ps = opKron(opLInterp1D(xt,model.xsrc),opLInterp1D(zt,model.zsrc));
Px = opKron(opExtension(model.n(2),model.nb(2)),opExtension(model.n(1),model.nb(1)));

% model parameter: slowness [s/m] on computational grid.
nu = 1e-3*Px*sqrt(m);

% distribute frequencies according to standard distribution
freq = distributed(model.freq);
w    = distributed(w);

% check source matrix input
if (size(Q,3)==1)&&(isdistributed(Q))
    Q = gather(Q);
end

spmd
    codistr  = codistributor1d(2,[],[nsrc*nrec,nfreq]);
    freqloc  = getLocalPart(freq);
    wloc     = getLocalPart(w);
    nfreqloc = length(freqloc);
    Dloc     = zeros(nrec*nsrc,nfreqloc);
    if size(Q,3)==1
        for k = 1:nfreqloc
            Hk  = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            Uk  = Hk\(wloc(k)*(Ps'*Q));
            Dloc(:,k) = vec(Pr*Uk);
        end
    else
        Qloc = getLocalPart(Q);
        for k = 1:nfreqloc
            Hk  = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            Uk  = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
            Dloc(:,k) = vec(Pr*Uk);
        end
    end
    D = codistributed.build(Dloc,codistr,'noCommunication');
end

% vectorize output, gather if needed
D = vec(D);

% construct pSPOT operator
J = oppDF(m,Q,model);
```

```
% comp. grid
ot = model.o-model.nb.*model.d;
dt = model.d;
nt = model.n+2*model.nb;
[zt,xt] = odn2grid(ot,dt,nt);

% data size
nsrc   = size(Q,2);
nrec   = length(model.zrec)*length(model.xrec);
nfreq  = length(model.freq);

w = fwi_wavelet(model.freq,model.t0,model.f0);

% mapping from source/receiver/physical grid to comp. grid
Pr = opKron(opLInterp1D(xt,model.xrec),opLInterp1D(zt,model.zrec));
Ps = opKron(opLInterp1D(xt,model.xsrc),opLInterp1D(zt,model.zsrc));
Px = opKron(opExtension(model.n(2),model.nb(2)),opExtension(model.n(1),model.nb(

% model parameter: slowness [s/m] on computational grid.
nu = 1e-3*Px*sqrt(m);

% distribute frequencies according to standard distribution
freq = distributed(model.freq);
w    = distributed(w);

% check source matrix input
if (size(Q,3)==1)&&(isdistributed(Q))
    Q = gather(Q);
end
```

Complicated formula - make it a separate function

DF.m - original

```
if flag==1
    % solve Helmholtz for each frequency in parallel
    spmd
        codistr   = codistributor1d(2,codistributor1d.unsetPartition,[nsrc*nrec,nfreq]);
        freqloc   = getLocalPart(freq);
        wloc      = getLocalPart(w);
        nfreqloc  = length(freqloc);
        outputloc = zeros(nsrc*nrec,nfreqloc);

        if size(Q,3) == 1
            for k = 1: nfreqloc
                [Hk, dHk] = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
                U0k       = Hk\(wloc(k)*(Ps'*Q));
                Sk        = -(2*pi*freqloc(k))*(dnu*(dHk*(U0k.*repmat(Px*input,1,nsrc))));
                U1k       = Hk\Sk;
                outputloc(:,k) = vec(Pr*U1k);
            end
        else
            Qloc = getLocalPart(Q);
            for k = 1: nfreqloc
                [Hk, dHk] = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
                U0k       = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
                Sk        = -(2*pi*freqloc(k))*(dnu*(dHk*(U0k.*repmat(Px*input,1,nsrc))));
                U1k       = Hk\Sk;
                outputloc(:,k) = vec(Pr*U1k);
            end
        end
        output = codistributed.build(outputloc,codistr,'noCommunication');
    end
    output = vec(output);

else
    spmd
        freqloc   = getLocalPart(freq);
        wloc      = getLocalPart(w);
        nfreqloc  = length(freqloc);
        outputloc = zeros(prod(model.n),1);
        inputloc  = getLocalPart(input);

        if size(Q,3)==1
            for k = 1:nfreqloc
                inputloc  = reshape(inputloc,[nsrc*nrec,nfreqloc]);
                [Hk, dHk] = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
                U0k       = Hk\(wloc(k)*(Ps'*Q));
                Sk        = -Pr'*reshape(inputloc(:,k),[nrec nsrc]);
                V0k       = Hk'\Sk;
                r         = (2*pi*freqloc(k))*real(sum(conj(U0k).*(dHk'*(dnu'*V0k)),2));
                outputloc = outputloc + Px'*r;
            end
        else
            Qloc = getLocalPart(Q);
            for k = 1:nfreqloc
                inputloc  = reshape(inputloc,[nsrc*nrec,nfreqloc]);
                [Hk, dHk] = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
                U0k       = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
                Sk        = -Pr'*reshape(inputloc(:,k),[nrec nsrc]);
                V0k       = Hk'\Sk;
                r         = (2*pi*freqloc(k))*real(sum(conj(U0k).*(dHk'*(dnu'*V0k)),2));
                outputloc = outputloc + Px'*r;
            end
        end
        output = pSPOT.utils.global_sum(outputloc);
    end
    output = output{1};
end
```

```
% solve Helmholtz for each frequency in parallel
spmd
    codistr   = codistributor1d(2,codistributor1d.unsetPartition,[nsrc*nrec,nfreq
    freqloc   = getLocalPart(freq);
    wloc      = getLocalPart(w);
    nfreqloc  = length(freqloc);
    outputloc = zeros(nsrc*nrec,nfreqloc);

    if size(Q,3) == 1
        for k = 1: nfreqloc
            [Hk, dHk] = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            U0k       = Hk\(wloc(k)*(Ps'*Q));
            Sk        = -(2*pi*freqloc(k))*(dnu*(dHk*(U0k.*repmat(Px*input,1,nsrc
            U1k       = Hk\Sk;
            outputloc(:,k) = vec(Pr*U1k);
        end
    else
        Qloc = getLocalPart(Q);
        for k = 1: nfreqloc
            [Hk, dHk] = Helm2D(2*pi*freqloc(k)*nu,ot,dt,nt,model.nb);
            U0k       = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
            Sk        = -(2*pi*freqloc(k))*(dnu*(dHk*(U0k.*repmat(Px*input,1,nsrc
            U1k       = Hk\Sk;
            outputloc(:,k) = vec(Pr*U1k);
        end
    end
    output = codistributed.build(outputloc,codistr,'noCommunication');
end
output = vec(output);
```

Doing basically the same thing, complicated formula

DF.m - first update

- if else branches for different cases
- matches the analytical formulas much closer
- still repeated code between F.m, DF.m
- even more repetition for Hessian functions

```
if flag==1
    input = Px*input;
    % solve Helmholtz for each frequency in parallel
    spmd
        codistr   = codistributor1d(2,codistributor1d.unsetPartition,[nsrc*nrec,nfreq]);
        freqloc   = getLocalPart(freq);
        wloc      = getLocalPart(w);
        nfreqloc  = length(freqloc);
        outputloc = zeros(nsrc*nrec,nfreqloc);
        if size(Q,3)>1
            Qloc = getLocalPart(Q);
        end

        for k=1:nfreqloc
            fm = f(m,freqloc(k));
            dfdm = df(m,freqloc(k)) .* input;

            % Hk = A * diag( (B*fm).^2 ) + const
            [Hk, A,B] = Helm2D(fm,ot,dt,nt,model.nb);

            % Derivative of mapping, fm -> Hk(fm)
            dHk = opMatrix(A)*opDiag((2*B*fm) .* (B*dfdm));

            if size(Q,3)==1
                U0k = Hk\(wloc(k)*(Ps'*Q));
            else
                U0k = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
            end

            % DU[dm] = H[m]\( -dH(m)[dm] * U(m) )
            Sk = -dHk * U0k;

            U1k = Hk\Sk;
            outputloc(:,k) = vec(Pr*U1k);
        end

        output = codistributed.build(outputloc,codistr,'noCommunication');
    end
    output = vec(output);

else
```

```
spmd
    codistr   = codistributor1d(2,codistributor1d.unsetPartition,[nsrc*nrec,nfreq
    freqloc   = getLocalPart(freq);
    wloc      = getLocalPart(w);
    nfreqloc  = length(freqloc);
    outputloc = zeros(nsrc*nrec,nfreqloc);

   if size(Q,3)>1
        Qloc = getLocalPart(Q);
    end

    for k=1:nfreqloc
        fm = f(m,freqloc(k));
        dfdm = df(m,freqloc(k)) .* input;

        % Hk = A * diag( (B*fm).^2 ) + const
        [Hk, A,B] = Helm2D(fm,ot,dt,nt,model.nb);

        % Derivative of mapping, fm -> Hk(fm)
        dHk = opMatrix(A)*opDiag((2*B*fm) .* (B*dfdm));

        if size(Q,3)==1
            U0k = Hk\(wloc(k)*(Ps'*Q));
        else
            U0k = Hk\(wloc(k)*(Ps'*Qloc(:,:,k)));
        end

        % DU[dm] = H[m]\( -dH(m)[dm] * U(m) )
        Sk = -dHk * U0k;

        U1k = Hk\Sk;
        outputloc(:,k) = vec(Pr*U1k);
    end
    output = codistributed.build(outputloc,codistr,'noCommunication');
end
output = vec(output);
```

Objective function - old way

```
function [f,g,h,w,f_aux] = misfit_red(m,Q,D,model,params)

[Dt,Jt] = F(m,Q,model);

D  = reshape(D, [nrec*nsrc,nfreq]);
Dt = reshape(Dt,[nrec*nsrc,nfreq]);

dR = D - Dt;

f = 0.5*norm(dR,'fro')^2;

if nargout > 1
    g = Jt'*vec(dR);
end

end
```

1 forward PDE solve

1 forward PDE re-solve, 1 adjoint PDE solve

Old code

Main components

- common setup code
  - helmholtz solving environment
  - model extension for PML, conversion from s^2/km^2 -> s/m
  - etc

- common computational code
  - forward wavefield solve
  - other wavefield solves, depending on what we need (forward modeling, forward jacobian, adjoint jacobian, hessian, gauss-newton hessian, LS objective)

New code

If you have a lot of duplication -> consolidation

PDEfunc.m

- general function for computing various quantities depending on Helmholtz solutions for FWI
  - forward model, migration, demigration, hessian, gauss-newton hessian, least squares objective + gradient
- options for solving PDEs iteratively, with LU, backslash

New code

If you have a lot of duplication -> consolidation

PDEfunc.m

- **additive** function over sources + receivers
- **serial** function intended to be run on each Matlab worker

Example - F.m, new code

```
if exist('params','var')==0, params = []; end

J = oppDF(m,Q,model,params);

freq = distributed(model.freq);
nsrc = length(model.xsrc); nrec = length(model.xrec); nfreq = length(model.freq);

spmd,
    [fStart,fEnd] = globalIndices(freq,2);
    model_loc = model;
    model_loc.freq = model_loc.freq(fStart:fEnd);
    Dloc = PDEfunc('forw_model',m,Q, [],[], model_loc, params);
    codist_f = getCodistributor(freq);
    codist = codistributor1d(2,codist_f.Partition,[nsrc*nrec,nfreq]);
    D = codistributed.build(Dloc,codist);
end

D = vec(D);
```

New code

For WRI

```
varargout = PDEfunc_wri( func, m, Q, dm, D, model, params);
```

Additional required options:

```
params.lambda - WRI lambda parameter
```

Check the documentation for more details

Code organization

F.m - forward modeling

oppDF.m - migration/demigration

oppH.m - Hessian
oppHGN.m - GN Hessian

PDEfunc - main setup, driver code

Benefits of new code

Single function where heavy computation is done

- any future optimizations to PDE solves propagate to entire code base

Separation of computation, parallelization

- can split up data in arbitrary ways without affecting the result, easier to test + show correctness

Benefits of new code

Now we have

- Gauss-Newton Hessian for FWI with no unnecessary PDE solves
- Full Hessian for FWI
- Gauss-Newton, Full Hessian for WRI
- All tested with correct Taylor error, adjoint test behaviour, so you know that they're actually correct

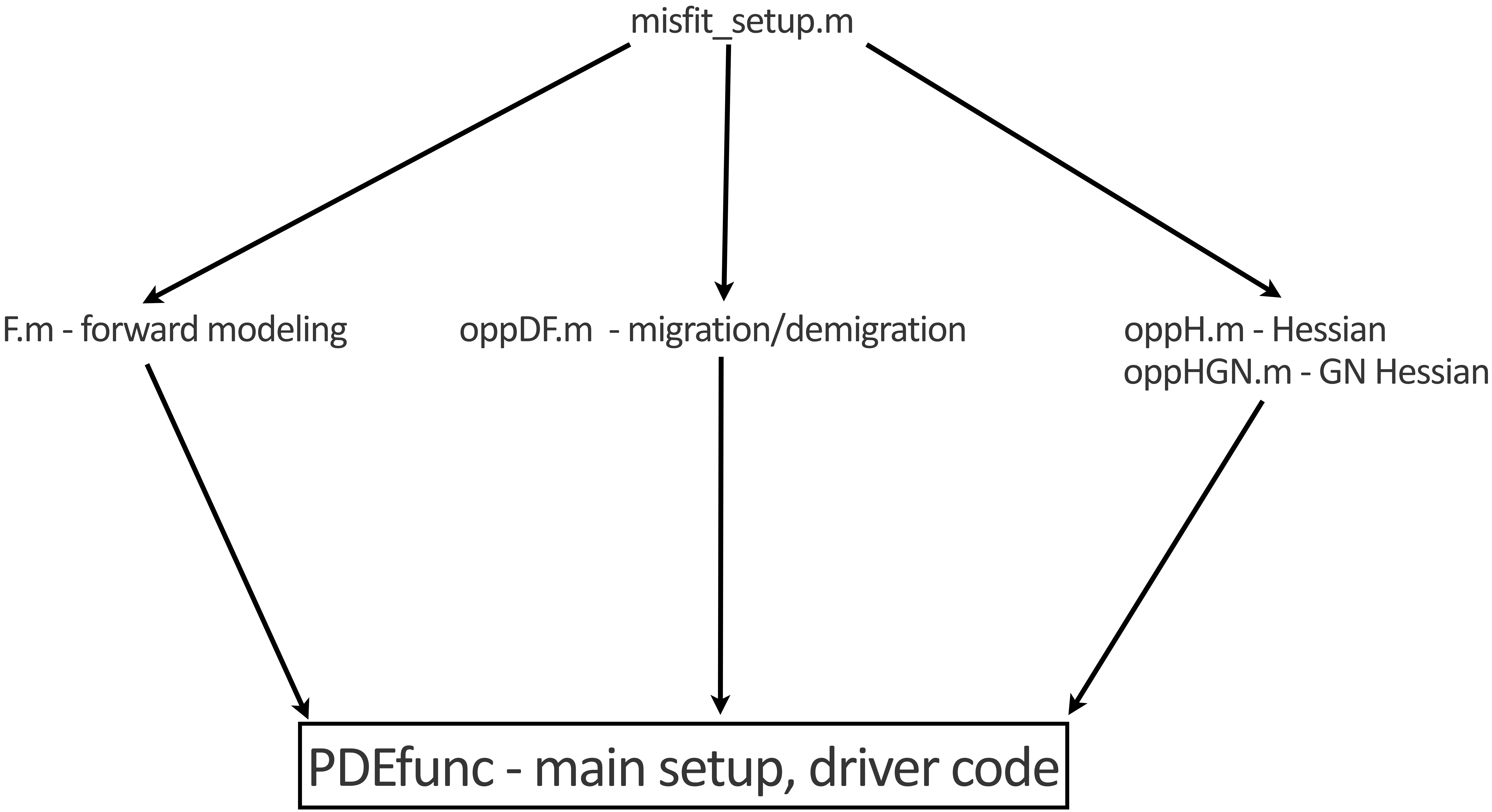
New code

You don't have to interact directly with PDEfunc, PDEfunc_wri

- F.m, oppDF.m, oppH.m, oppHGN.m just implement the parallelization over frequencies, use PDEfunc as their main driving code

Instead, there is the misfit_setup.m function for generating least-squares FWI, WRI objectives

Code organization



New code example - easy frequency continuation

Parallel over frequencies (default)

```
params.wri = false;                    %FWI objective
%params.dist_mode = ‘freq’;            %distribute over frequencies
freq_batch = {1:2,3:4,5:6,7:8};
mk = m0;

for j=1:length(freq_batch)
    params.freq_index = freq_batch{j}; %frequency selection
    obj_fwi = misfit_setup(Q,Dobs,model,params);
    mk = minFunc(obj_fwi,mk,opts);
end
```

New code example - easy frequency continuation

Parallel over sources

```
params.wri = false;                    %FWI objective
params.dist_mode = ‘src’;              %distribute over sources
freq_batch = {1:2,3:4,5:6,7:8};
mk = m0;

for j=1:length(freq_batch)
    params.freq_index = freq_batch{j};
    obj_fwi = misfit_setup(Q,Dobs,model,params);
    mk = minFunc(obj_fwi,mk,opts);
end
```

New code example - easy frequency continuation

Parallel over sources + frequencies

```
params.wri = false;                  %FWI objective
params.dist_mode = 'srcfreq';        %distribute over sources +
                                      frequencies

freq_batch = {1:2,3:4,5:6,7:8};
mk = m0;

for j=1:length(freq_batch)
    params.freq_index = freq_batch{j};
    obj_fwi = misfit_setup(Q,Dobs,model,params);
    mk = minFunc(obj_fwi,mk,opts);
end
```

New code example - easy frequency continuation

Parallel over frequencies

```
params.wri = true;                  %WRI objective
params.lambda = 10;                 %WRI penalty parameter
%params.hessian = ‘sparse’;         %sparse hessian for WRI,
                                    default

freq_batch = {1:2,3:4,5:6,7:8};
mk = m0;

for j=1:length(freq_batch)
    params.freq_index = freq_batch{j};
    obj_wri = misfit_setup(Q,Dobs,model,params);
    mk = minFunc(obj_wri,mk,opts);
end
```

New code example - easy frequency continuation

Parallel over frequencies

```
params.wri = true;                 %WRI objective
params.lambda = 10;                %WRI penalty parameter
params.hessian = ‘gn’;             %GN Hessian for WRI
freq_batch = {1:2,3:4,5:6,7:8};
mk = m0;

for j=1:length(freq_batch)
    params.freq_index = freq_batch{j};
    obj_wri = misfit_setup(Q,Dobs,model,params);
    mk = minFunc(obj_wri,mk,opts);
end
```

Summary

New code is

- modular and maintainable
  - parallelization, computation separated
  - easier to test, also passes tests
- easy to set up objectives + hessians for FWI/WRI
  - no unnecessary PDE solves
  - choice of modes of distributing data, not just over frequencies
- found in /tools/algorithms/
  - /2DFreqModeling - FWI
  - /WRI - WRI

3D Code

3D Modeling and Inversion

NEW 27 point stencil Helmholtz implementation

NEW linear system solvers

NEW multigrid-based preconditioner

All work by Rafael Lago

3D Modeling and Inversion

NEW 27 point stencil Helmholtz implementation

in /tools/algorithms/3DFreqModeling

- as little as 4 points per wave-length needed to stably invert the Helmholtz equation
- compared to ~10 points per wave-length of the standard 7 point stencil

SLIM

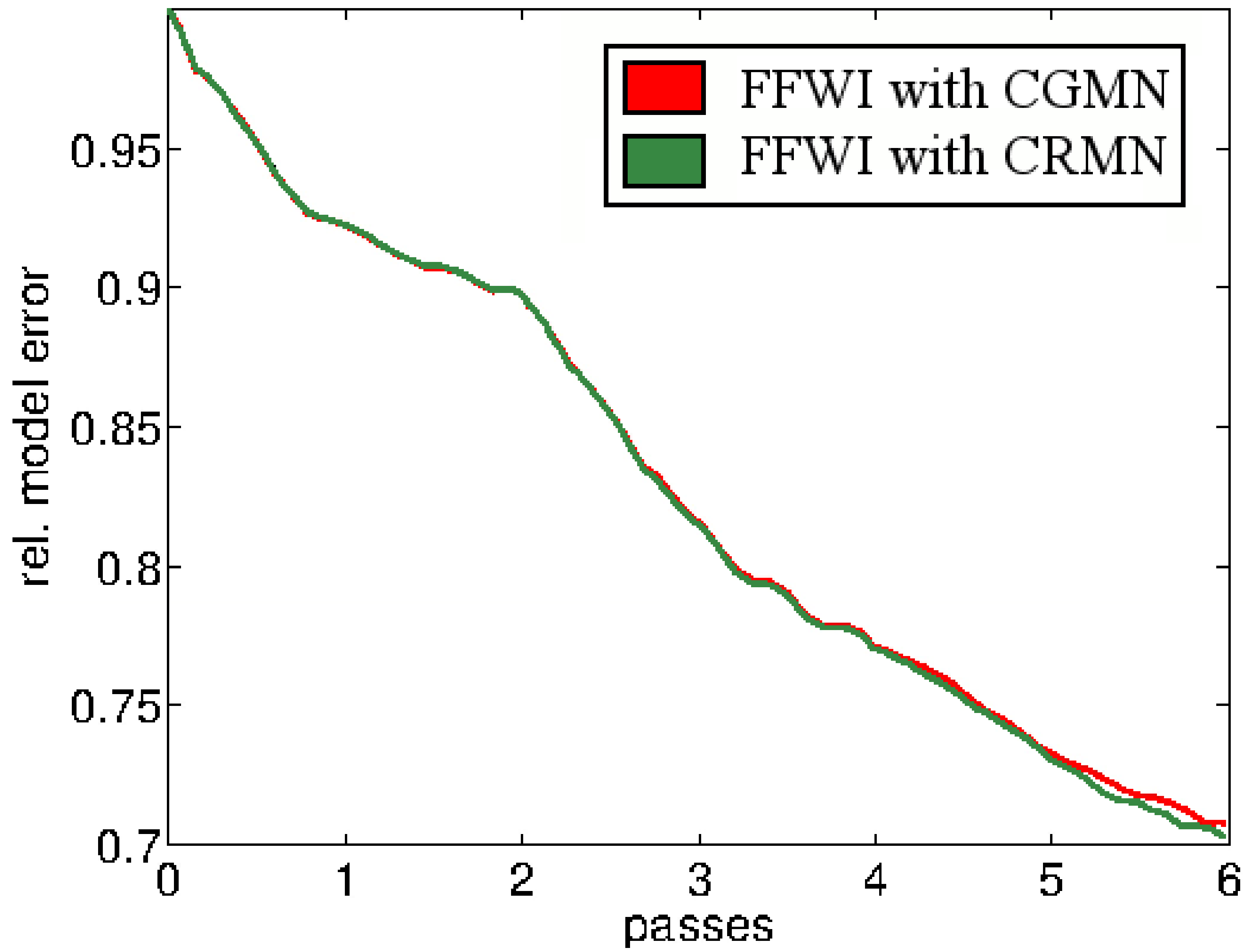
3D Modeling and Inversion

NEW linear system solvers

/tools/solvers/Krylov

- CGMN - CG preconditioned w/ double Kaczmarz sweeps
- CRMN - CR preconditioned w/ double Kaczmarz sweeps
- FGMRES - flexible GMRES, for use with the new preconditioner

3D Modeling and Inversion



PDE solver iterations

	FFWI with CGMN	FFWI with CRMN	Speedup
4 *Hz*	23,403	19,846	18%
6 *Hz*	30,189	24,387	24%
8 *Hz*	34,724	26,265	32%
Total	88,316	70,498	25%

Frugal FWI with a fixed number of PDE solves per frequency
Loose PDE tolerance at early iterations, tightens as iterations proceed

[1] Calandra, H., Gratton, S., Pinel, X. and Vasseur, X. [2013] An improved two-grid preconditioner for the solution of three-dimensional Helmholtz problems in heterogeneous media.

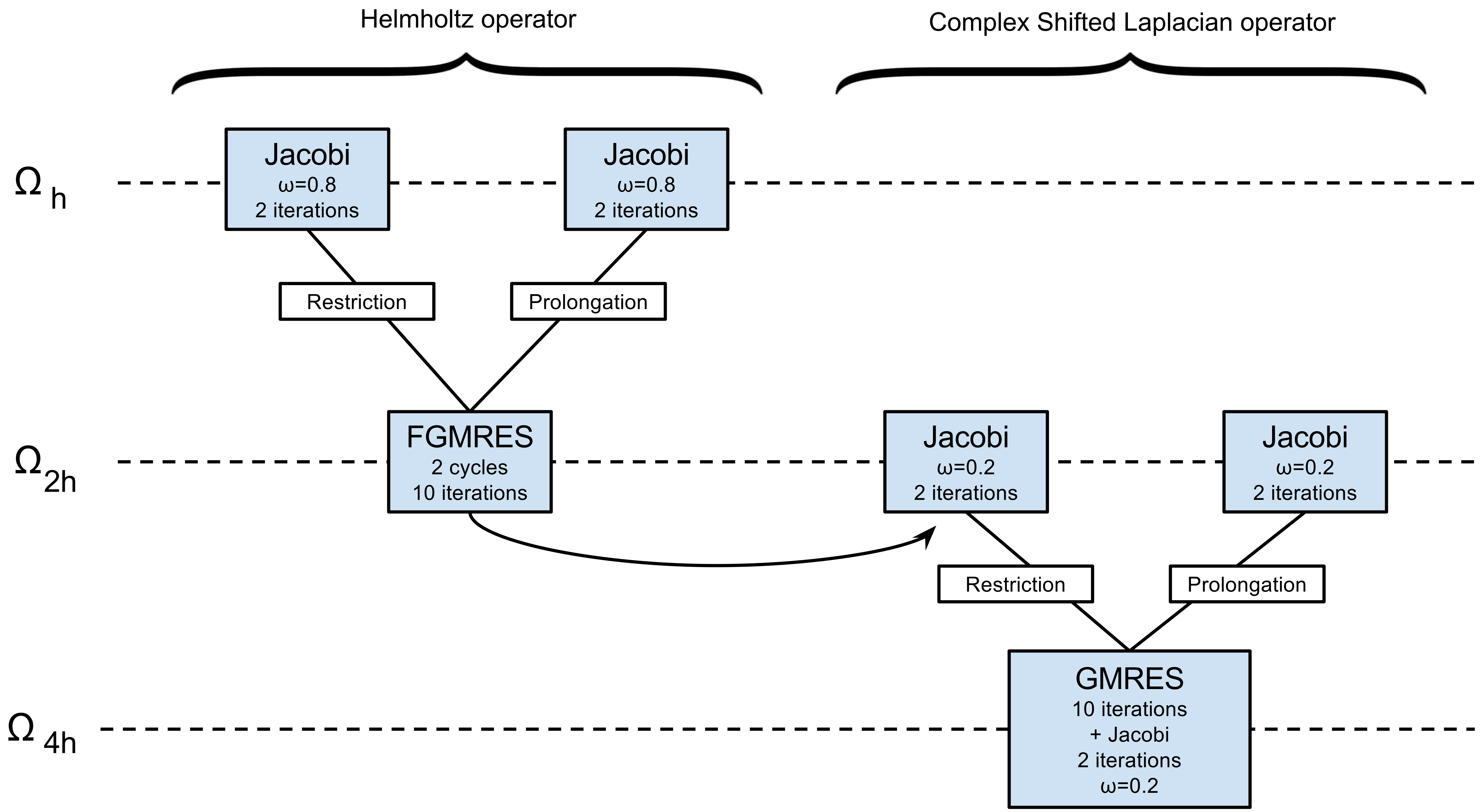
3D Modeling and Inversion

NEW multigrid-based preconditioner

in /tools/solvers/Multigrid

Extends previous work in [1] for a multigrid, shifted-Laplacian approach for a 7-point stencil to the 27-point case

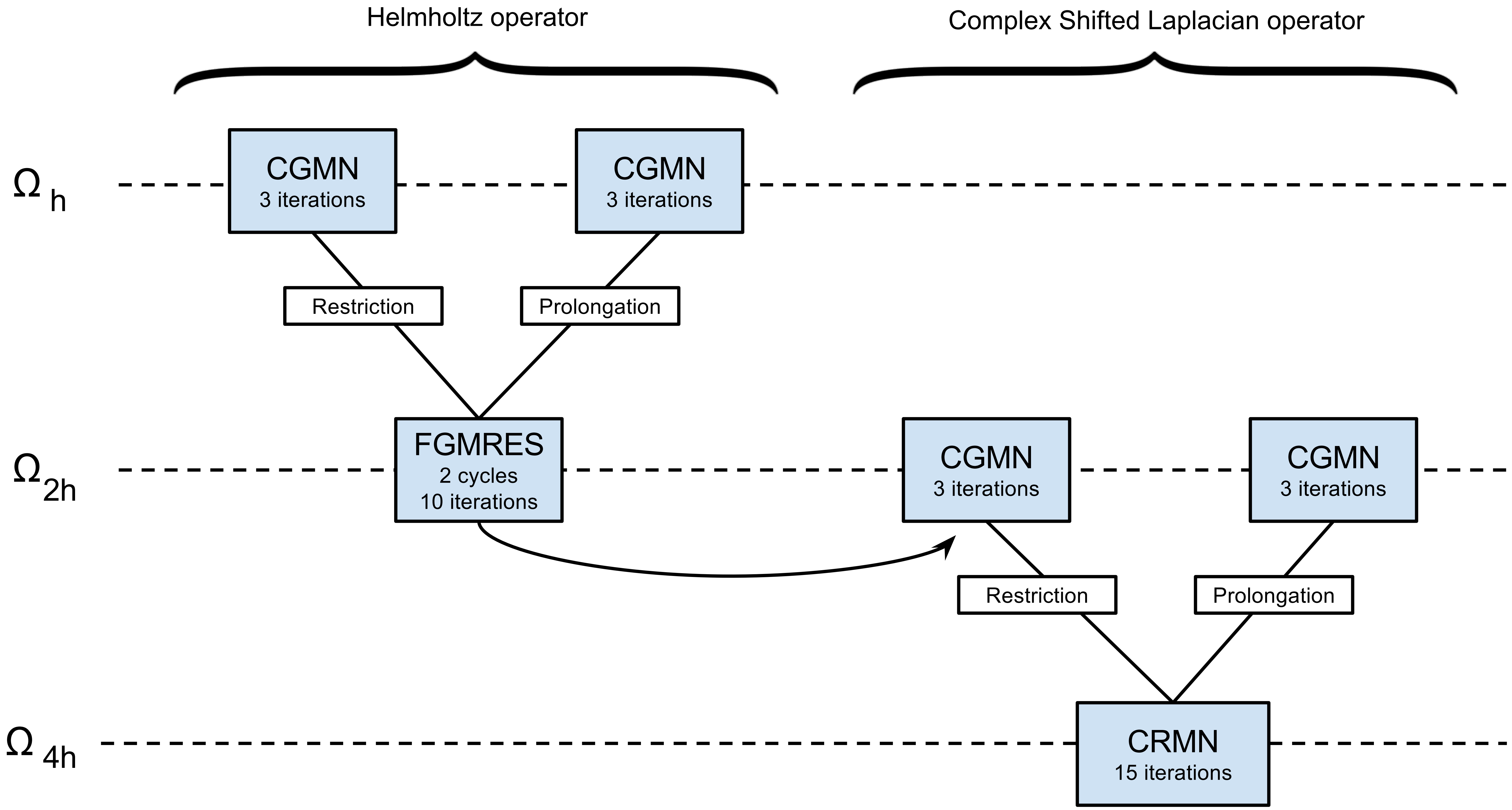
3D Modeling and Inversion



Preconditioner of [1]
Only works for 7 point stencil

[1] Calandra, H., Gratton, S., Pinel, X. and Vasseur, X. [2013] An improved two-grid preconditioner for the solution of three-dimensional Helmholtz problems in heterogeneous media.

3D Modeling and Inversion



New preconditioner

Works with 27 point stencil

3D Modeling and Inversion

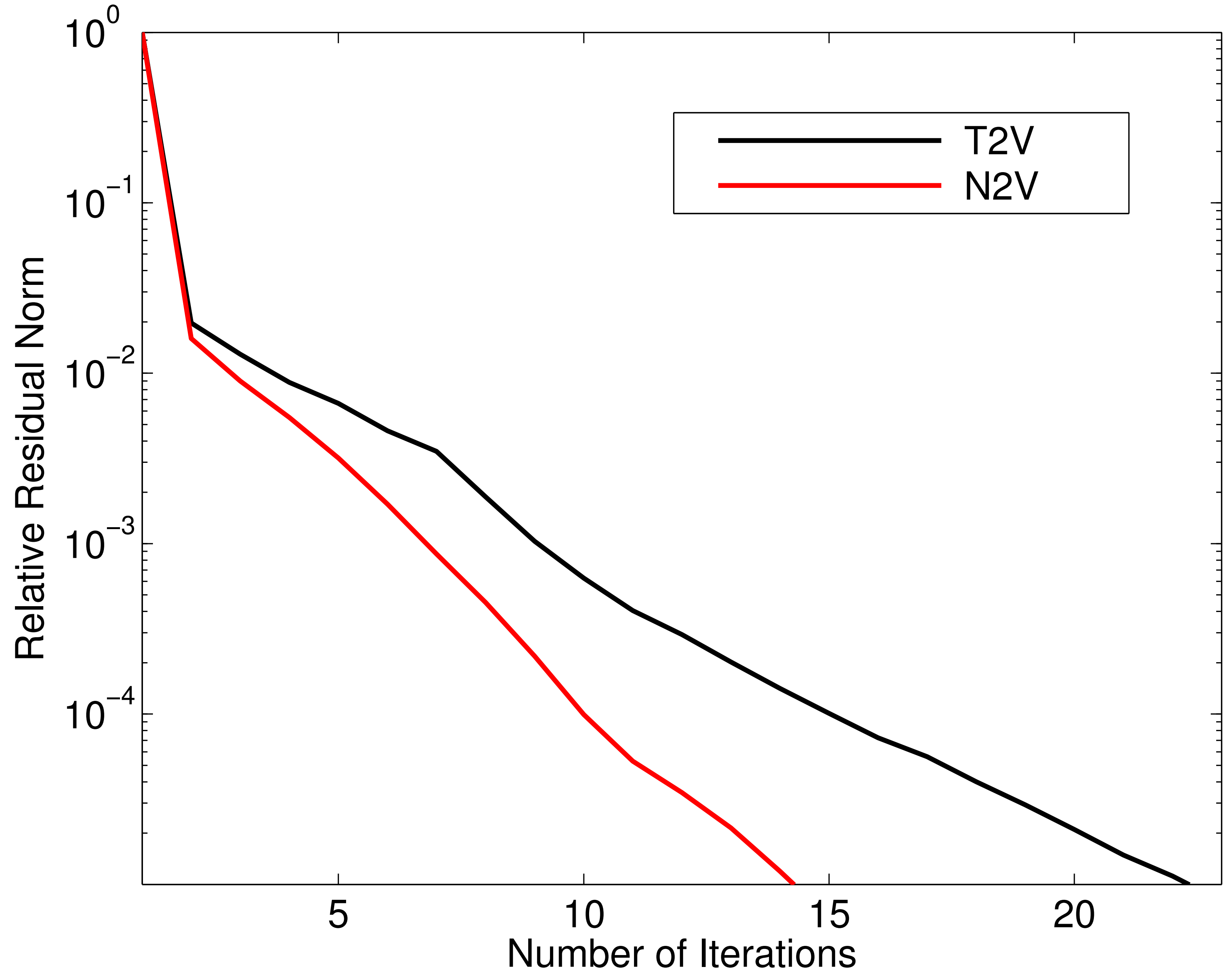


Figure 3 *Second numerical experiment, using 7 points stencil and 10 points per wavelength for SEG/EAGE Overthrust at 8Hz*

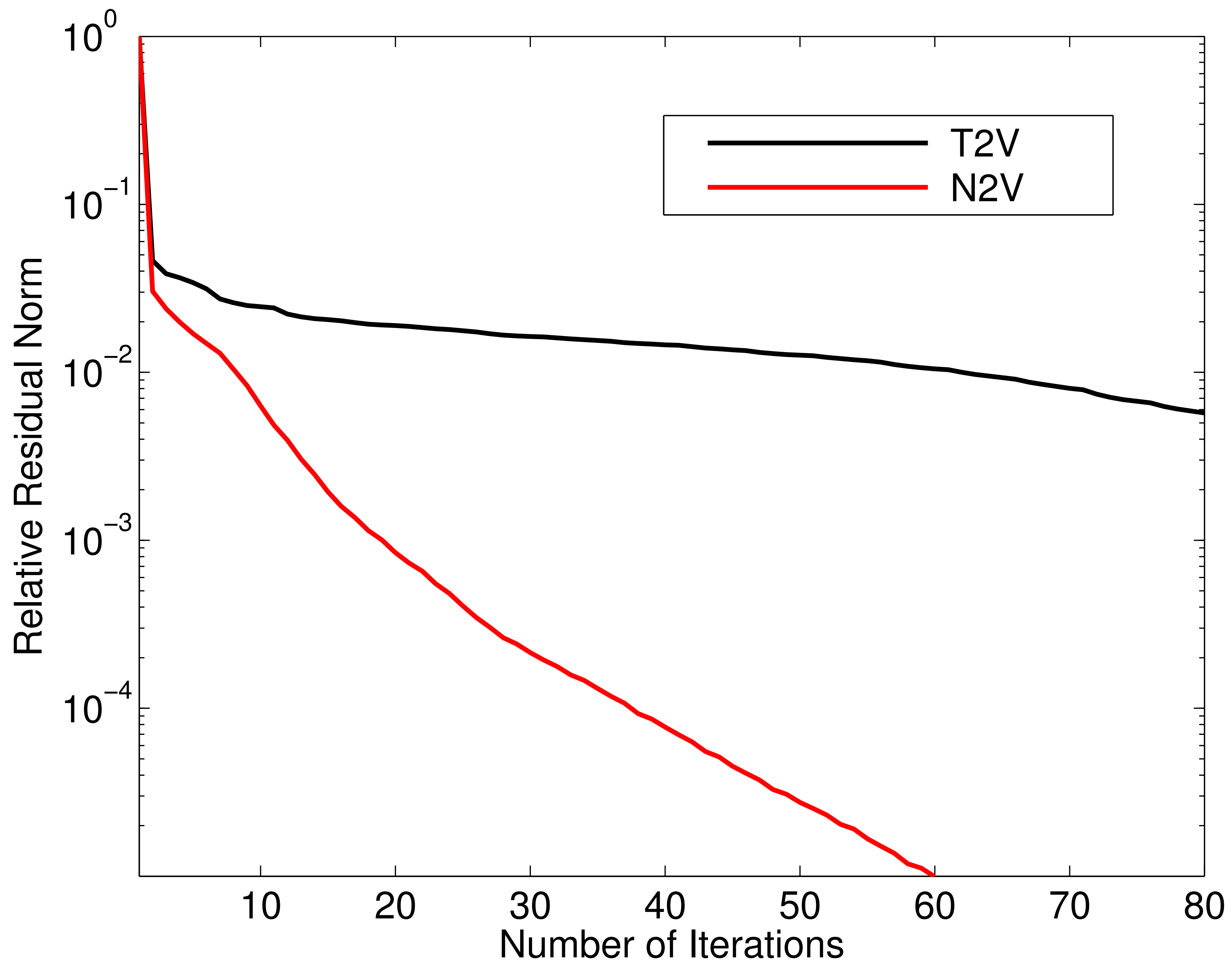


Figure 4 *Second numerical experiment, using 27 points stencil and 6 points per wavelength for SEG/EAGE Overthrust at 8Hz*

Summary

New 27 point stencil - less points per wavelength

New Krylov solvers - less iterations, smooth error decrease for Frugal FWI

New Multigrid preconditioner - less iterations

SLIM

Acknowledgements

Thank you for your attention

SINBAD

NSERC CRSNG

This work was in part financially supported by the Natural Sciences and Engineering Research Council of Canada via the Collaborative Research and Development Grant DNOISEII (375142--08). This research was carried out as part of the SINBAD II project which is supported by the following organizations: BG Group, BGP, CGG, Chevron, ConocoPhillips, DownUnder GeoSolutions, Hess, Petrobras, PGS, Schlumberger, Sub Salt Solutions and Woodside.