User guide for the CREWES frequency domain FWI codes

Scott Keating and Kris Innanen

ABSTRACT

We present a user guide for a frequency domain, elastic FWI MATLAB code frequently used by CREWES. The guide includes instructions on how to run an inversion, some tips on troubleshooting, and suggestions of possible modifications and the corresponding work required. Please contact Kevin Hall for an updated version.

INTRODUCTION

Several of the projects reported on in this and previous research reports are based on a frequency-domain FWI code used by CREWES. This code is two dimensional, and forms using acoustic, elastic and viscoelastic wave propagation exist. The code is not suitable for 3D or large scale problems, but can be useful for analyzing the FWI problem through synthetic examples and relatively small field data sets (e.g. Eaid et al., 2021; Keating et al., 2022).

In this report, we provide a user guide for the use of these frequency-domain codes. In this report, our focus is on the elastic implementation in the example code package "Elastic". This is planned to be the first of a set of user guides, which will be updated in the future. Please contact Kevin Hall in order to get the most recent version.

The report is organized into three main sections. The first section explains how to specify an inversion problem to the code and run it, the second section suggests some troubleshooting strategies for when errors are encountered, and the third section suggests where to modify the code for some common alterations.

RUNNING AN INVERSION

In the "Elastic_inversion" code, several subsections exist. To run an inversion, only changes in this top-level script should generally be required. Only when changes to the method of inversion are desired should edits at deeper levels be required. In this section, we describe what each subsection is doing, and what the user needs to do to give the code a chosen inversion problem.

Defining the initial model

In this section we define the initial model used in the inversion. However you choose to generate your initial model, there are a few required variables that should be defined in this subsection. First, you should produce a model0 (and model_true, if synthetic) that is a vector with index increments first increasing z, then x, then parameter number. This is the ordering of vector you will get if $m = m(z, x, par)$ and you define $m_0 = m(:)$ in MATLAB. Your model should be defined in the elastic parameters you will use for the inversion.

Other key variables to define are your finite-difference grid cell size dz, the number of x and z positions in the model, nz and nx, and the thickness of the perfectly-matched layer (PML) absorbing boundary region, PML_thick. The PML thickness is usually safe to set at about 20 grid cells.

Defining the elastic parameterization

In this section, you need to define a function that translates between the parameters you use in the inversion and the elastic properties that the code uses internally. This function will be an anonymous function of the form "Elastic_def = @(model)yourfunction(model, otherarguments)", where the terms in green need to be decided by the user. The yourfunction code should take an $nz \times nx \times npar$ model defined in your parameters, and output, in this order: 1 - an $nz \times nx$ density, 2 - an $nz \times nx$ c_{11}, 3 - an $nz \times nx$ c_{44}, 4 - an $nz \times nx \times npar$ tensor containing the derivative of density with respect to parameter p in $(:,:,p)$, 5 - the same for the derivative of c_{11}, 6 - the same for the derivative of c_{44}. This might sound confusing, but these codes are typically quite simple in practice. There are several examples in the code package, such as "Elastic_def_IpIsVp" that define the parameterization for common cases.

At the end of this section, the function you define is saved locally as "var_def.mat". This means that it is important not to navigate away from the current folder in MATLAB during the inversion unless you modify the code to specify the path of var_def in the subfunctions that need it. We also save a "Param_num.mat" letting the inversion know how many parameters you are using.

Defining the acquisition geometry

The purpose of this section is to define matrices S and R, which define the properties of the source and receiver sampling. In the simplest formulation, it is assumed that each receiver is active for each source (this can be relaxed by changing R to a cell-array of R matrices, but this version of the code is bulkier and hasn't been included in the code package for this report). S should be $2 * N_{PML} \times ns$, and R should be $nr \times 2 * N_{PML}$, both sparse, where $N_{PML} = (nz + PML\_thick)(nx + PML\_thick)$. Each column of S defines the source term for one source, and each row of R defines the sensitivity of one receiver channel in the full, PML-padded model grid. These matrices can be simply constructed for most cases using the functions "Define_geometry" and "Define_Acquisition_xxx", where xxx is the source type.

The "Define_geometry" function makes vectors for source and receiver locations, provided that they consist of, at most, two vertical and two horizontal, regularly spaced lines of sources and receivers with common start and end points. It is a function of the form "[rz,rx,sz,sx] = Define_geometry(geom_vec, r_acq, s_acq)", where "geom_vec = [rz_start, rz1, rz2, rz_end, rz_inter, rx_start, rx1, rx2, rx_end, rx_inter, sz_start, sz1, sz2, sz_end, sz_inter, sx_start, sx1, sx2, sx_end, sx_inter]". In this expression rz_start is the z-index at which the vertical receiver lines start, _end is the end index, rz_inter is the vertical spacing of receivers in grid cells, and rz1 and rz2 are the z-positions (in grid-index) of the horizontal receiver lines. Similar conventions hold under r/s, x/z changes. The _acq variables tell

the function which of the possible lines are active for a given acquisition (details are in the code). Variables in geom relating to non-existent lines can be left as 0.

The "Define_Acquisition_xxx" functions are of the form "[S, R] = Define_Acquisition_Explosive($sz, sx, rz, rx, nz, nx, PML\_thick, plot\_out$);". They should take the vectors just created and generate appropriate S and R matrices. These codes assume multicomponent point receivers in all cases, and sources of the type in the function name. If this is not the case, you will need to construct S, R, or both yourself. When doing this, it is important to note that the wavefields are indexed z-position first, alternating x and z component: u = [ux1, uz1, ux2, uz2, . . .].

Defining the spatial component of inversion variables

Earlier, we defined the parameterization of the inversion, that is, which elastic properties the inversion works in. To fully define our inversion variables, we need to also specify the spatial component of our inversion variables; that is, when we change an inversion variable's value, where in the model does a given parameter change? The simplest choice here is to have each inversion variable define a parameter at one grid cell, and this formulation often works well. In some cases, though, we may want to choose different spatial components. For instance, if we want to invert for a layered, but 2D, medium, we may want to define our inversion variables as the properties of entire layers rather than just grid cells. We can do this by defining P matrices. Simply put, if our inversion variables are in the vector m_{INV}, we define the elastic model (in our inversion parameterization) on the finite-difference grid as $m_{FD} = P * m_{INV}$. So, each column of P defines the spatial component of the corresponding element of our inversion variable model. In the simplest case, P is an identity matrix and each inversion variable corresponds to a single grid cell. In our layering case, however, P is only as wide as the number of z elements, and each column contains ones at positions corresponding to a given depth, and zeros elsewhere. The code requires both P, which is defined on the normal model grid, and P_big, which is defined on the PML-padded grid, as these are used at different points in the code. In general, P_big adds the PML region on to inversion variables that take a value at the edge of the PML region. This prevents a contrast between the PML region and the model interior from generating unwanted reflections, but it can also lead to correct, but problematically large amplitude gradients on the model edge.

Defining the frequency bands used

A major consideration in a frequency-domain FWI approach is the choice of frequency bands to invert over. In general, a multiscale approach, in which lower frequencies are inverted before higher ones, tends to be useful in avoiding potential local minima. In this section, we need to define the number of frequency bands considered, "numbands", the number of frequencies per band, "step", the set of frequencies considered, "freq", and the wavelet value at each frequency, "fwave". The choice of each of these parameters requires some care, and a set of guidelines are provided below.

freq – This is the set of frequencies being inverted, in order. In the example codes, this is constructed with a fixed lowest frequency, and bands that have an increasing maximum

frequency. It is important to make sure that grid dispersion is largely avoided; try to choose a maximum frequency such that $f_max < \frac{v_{S_{min}}}{4dz}$.

step – This is the number of frequencies per band. This number is directly related to cost, but also simple to parallelize over. Too few frequencies per band (e.g. 1-3) can lead to prominent artifacts, but too many can be wasteful. It's usually a good idea to set step equal to the number of parallel MATLAB workers available, provided this number is < 10.

fwave – The wavelet plays an important role for field data, in which case it should be carefully estimated based on the data. For synthetics, it plays less of a role, especially the phase, which has minimal impact. The amplitude is also fairly arbitrary in the noise-free case: re-weighting can trivially undo any non-uniform amplitude spectrum. When considering noisy examples this term has more of a role. The fwave variable is defined as the frequency domain value of the wavelet at the frequencies in freq.

numbands – The number of bands can be quite small (e.g. 3-4) and still provide good cycle-skipping behaviour, so this parameter largely plays the role of deciding total computational effort. Generally, we have found that very large numbers of inversion iterations on small numbers of bands provide inferior results to fewer iterations on more bands. The 10-20 range seems to work well generally.

Defining forward modeling / Generating observed data

Here, we need to specify the forward modeling function and, if synthetic, use it to create our "measured data". Our forward modeling function needs to be an anonymous function of the form "FDFDfunc = @(frequency, fwave, model)Forward_modeling_code(frequency, fwave, model, other parameters);"

If everything has been set up properly, we should now be able to generate a data-set with the line "D = Get_data_general(FDFDfunc, freq, fwave, model_true, R);". If noise is going to be included in a synthetic case, it should be added to the data here. This is a good place to check if the code is working: look at the data and make sure nothing clearly wrong is happening. This can save a lot of time, as relatively little computation has been done by this point.

Setting optimization parameters and regularization function

In this section, we specify the parameters used to control the inversion process, specifically numits, maxits and optype. The optype variable controls the optimization strategy, and can be steepest-descent (1), truncated Newton (2), or L-BFGS (3). The numits variable decides on the number of iterations per band, while the maxits variable decides the number of inner iterations per outer iteration, and is only used for truncated Newton optimization.

Regularization terms are highly dependent on the prior information available and will often need to be specifically written for the inversion problem in mind. The code takes a regularization function of the form "[rf,rg,rH] = @(model)reg_func(model)", where the model is an unscaled version of the inversion model, rf is the objective penalty term for a

given model, rg is the gradient of that term, and rH is a sparse representation of its Hessian. Use the "check_gradient" code to confirm that your regularization correctly calculates the appropriate derivatives. Problems will arise eventually if this check doesn't pass.

Main inversion code

In this section, we take the parameters we've chosen and input them into the inversion engine. If everything has worked correctly, the inversion will run, and will output an inversion model and an objective function history. Along the way, the output of the line-search in the code will be printed to the command window as the inversion runs. This allows us to keep track of the objective function and gradient during the inversion. If the inversion has run successfully and outputs a reasonable result, congratulations! Otherwise, the next section is worth consulting.

TROUBLESHOOTING

While there are many ways to make errors in the inversion input, some are much more common than others. The majority of problems I've encountered can be tracked down after starting with one of two checks:

1. Run "Elastic_def(model0)". This should take your initial model on the finite-difference grid and output c_{11}, c_{44} and density for that model. Check to see if these parameters are returned with reasonable values. Most errors not caught in the next step that we've encountered arise from a disjoint between the parameterization defined in different places. If the values recovered are not reasonable, double check that your parameterization is consistent. This step catches fewer errors than the next, but it is much less computationally expensive, so it's worth checking first.

2. In the optimization function you are using for the inversion (e.g. "TGN_optimization_general" or "Solve_LBFGS"), find the "Linesearch" line, and turn the last argument to 1 from 0. This will turn on a finite-difference check of your gradient. When on, the code will compare a numerical estimate of the 1D gradient in the direction specified: $\frac{\phi(m+\alpha*dd)-\phi(m)}{\alpha*dd}$ with the calculated 1D gradient: $\frac{\partial\phi}{\partial m} * dd$, where α is a small number. These values are labeled 'Numerical 1D derivative' and 'Expected 1D derivative', respectively in the command-line output. If they do not match to at least three or four significant digits, it is indicative that there is an error in the gradient calculation. Check this value for a few iterations. Any errors suggest that the gradient is not being calculated correctly: check your regularization and gradient functions. If there are no errors, the inversion is probably working properly and any errors are likely from unintended inputs: check that the model, acquisition, etc. match your expectations. Make sure to turn off the check when you are done troubleshooting as it incurs some extra expense at every linesearch.

EDITING THE CODE

By changing the inputs in the main script, you should be able to run an elastic inversion with respect to arbitrary elastic variables (with any parameterization and / or spatial basis functions), multicomponent receivers and explosive or z/x-oriented point-sources. The fre-

quency bands used can be chosen, and steepest-descent, truncated Newton and L-BFGS optimization are available. Modifications to the code which expand beyond these possibilities range from relatively simple to quite complex. I outline the feasibility of some of the possible edits and identify where relevant changes need to be made below.

Optimization strategy

The code is structured to provide objective functions, gradients and Hessian vector products to the optimization functions. In this sense, changes to the optimization can be relatively targeted, without requiring large-scale changes. Not all optimization strategies will be feasible to implement in terms of cost, but the code should be easily modified to treat most optimization strategies.

Inversion variables

The inversion in Elastic_inversion is designed to recover elastic properties of the subsurface, but changes to invert for source properties are relatively straightforward, and have been implemented in other versions. Re-framing to invert for almost any of the forward modeling inputs should be possible, and relatively straightforward.

Objective function

The gradient and objective function are calculated in Gradient_general, where forward and adjoint wavefields are propagated for every frequency. These wavefields will likely be significant regardless of the objective function used, and may limit the amount of work that needs to be reproduced. The code currently is not designed to change objective functions, however, so major rewrites withing Gradient_general and Hv_prod_general will be needed. These should be the only locations in the code where changes are necessary for an objective function change.

Forward modeling physics

Versions of these codes including attenuation exist, but other changes to the forward modelling engine (to include anisotropy, for instance) will probably require a complete rewriting of the codes dealing with the creation and derivatives of the Helmholtz matrix. The inversion requires the Helmholtz matrix in order to do forward modeling, and it requires the derivative of the Helmholtz matrix with respect to the inversion variables in order to calculate the gradient. These quantities are very large: for a model of N finite-difference cells, the Helmholtz matrix is N$\times$N, while its derivative will be on the order of N$\times$N$\times$N. They are also sparse, however, which allows them to be used efficiently. In this code, we use "Make_Helm_and_Derivative" to make a matrix where each vector contains either a diagonal of the Helmholtz matrix, or a vector of derivative terms. "Assemble_Helm" constructs the sparse Helmholtz matrix, and "IP_dpSA_B" calculates the Inner Product of [the derivative of S with respect to p, multiplied by A] with [B], where S is the Helmholtz matrix, p is the set of inversion variables, and A and B are vectors. This term is required for the gradient calculation, but the derivative term is difficult to construct, even sparsely, so this

function bypasses this step. Changing the forward modeling will likely require modifying or replacing each of these codes appropriately.

Forward modeling domain

These codes are designed for frequency-domain inversion, with the associated benefits and challenges. They are not easily adapted to a time-domain formulation; rewriting from scratch is probably easier. That being said, the optimization functions (Linesearch, Steepest_descent_optimization, TGN_optimization and LBFGS_solve) are not frequency-domain specific, and given gradient (and potentially Hessian-vector product) calculating functions with the same input/output structure as used in these codes, should be able to perform optimization, regardless of how the gradient and objective function are calculated.

ACKNOWLEDGEMENTS

The authors thank the sponsors of CREWES for continued support. This work was funded by CREWES industrial sponsors and NSERC (Natural Science and Engineering Research Council of Canada) through the grant CRDPJ 543578-19. Scott Keating was also supported by the Canada First Research Excellence Fund, through the Global Research Initiative at the University of Calgary.

REFERENCES

Eaid, M., Keating, S., and Innanen, K. A., 2021, Full waveform inversion of das field data from the 2018 cami vsp survey: CREWES Annual Report, **33**.

Keating, S., Innanen, K. A., and Wong, J., 2022, Cross-gradient regularization for multiparameter fwi of physically modeled data: CREWES Annual Report, **34**.