Trans-dimensional inversion of CSEM data using Bayesian RJ-MCMC

Anandaroop Ray
anray@ucsd.edu
Kerry Key
kkey@ucsd.edu
http://marineemlab.ucsd.edu

March 18, 2014

Contents

Building the MATLAB MEX file and plotting codes

0.1 From a .m file to a MATLAB executable

The first order of business is to build the MATLAB MEX file for 1D CSEM. For this, one needs to have MATLAB Coder installed. This allows you to run the anisotropic 1D CSEM forward modeling code as a compiled executable, rather than as an interpreted file.

This is simply achieved by starting MATLAB, getting to the **mex** directory and running **build_mex.m** and then **build_mex_hed_ved.m**.

If all goes well, you should see a response like this, after a few seconds

```
Code generation successful: View report
>>
```

You can test to see that the code is working by running testMex.m

```
>>testMex
Elapsed time is 0.129797 seconds.
Elapsed time is 0.023201 seconds.
```

The first time is for a normal call, the second is due to the MEX, for the 1D canonical reservoir model at 3 frequencies with 150 receivers using the lagged convolution code of [1, 3]. The results are shown in Figure 1. Feel free to modify testMex.m for different field responses.

0.2 Plotting routines

An easier way to get model responses and plot them is to first include the **mex** and **include_matlab** directories in your MATLAB path (Figure 2). Now navigate to the **exam-**

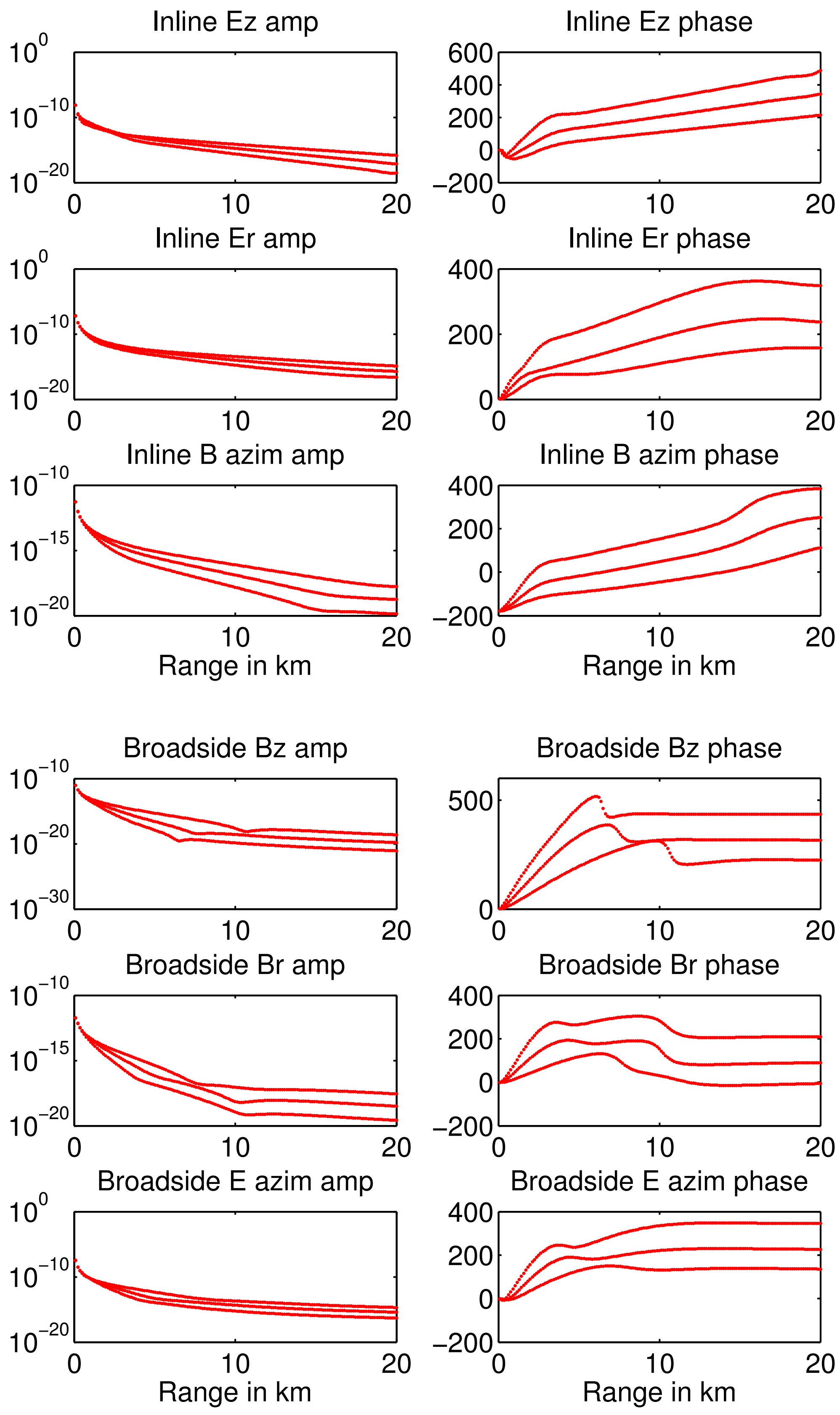
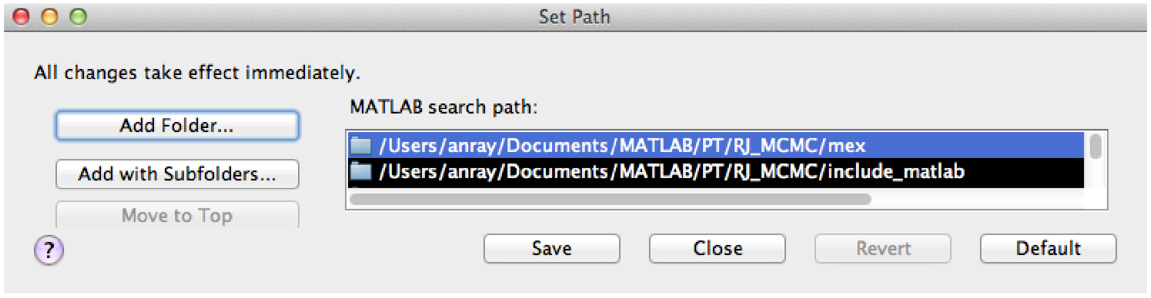


Figure 1: Inline and broadside CSEM fields due to 1D canonical model at 0.1, 0.3 and 0.7 Hz



```
/Users/anray/Documents/MATLAB/PT/RJ_MCMC/mex
/Users/anray/Documents/MATLAB/PT/RJ_MCMC/include_matlab
```

Figure 2: Setting your MATLAB path

ple_aniso directory. You can then use a structure, say **S** to include all the relevant survey information and another model structure, say **canonical** in the following manner.

```
>> ls
canonical.mat misfit_parameters_canonical_ANISO.mat

>> S = load ('misfit_parameters_canonical_ANISO')

S =

             ans: [0x1 double]
               f: [0.1000 0.3000 0.7000]
             zTx: 990
               r: [1x20 double]
             zRx: [999 999 999 999 999 999 999 999 999 999 999 999 999 999 999]
               z: [-1000000 0 1000]
            data: [20x18 double]
              sd: 0.0400
      components: [1 0 0 0 0 0 0 0]
             rho: [2x2 double]
            kMin: 1
            kMax: 30
            zMin: 1002
            zMax: 3500
           rhMin: -1
           rhMax: 3.3000
           rvMin: -1
           rvMax: 3.3000
            rSD2: 0.0200
            rSD1: 0.1000
```

```
          MoveSd: 2.2000
            hMin: 0
    chooseconfig: 1
       isotropic: 0
            RxAz: 0
           TxDip: 0

>> load canonical
>> canonical

canonical =

       z: [2000 2100]
    rhoh: [0 2 0]
    rhov: [0.6021 2 0]

>>
```

The model **canonical** has interfaces at 2000 and 2100 m, with horizontal and vertical resistivities specified as log10(resistivity). The upper parts of the resistivity structure to complete the model can be found in **S.z** and **S.rho**.

```
>> S.z

ans =

    -1000000           0        1000

>> S.rho

ans =

        1e+13       0.3125
        1e+13       0.3125

>>
```

You can plot the model response, in the following manner.

```
>> plot_model_field(S,{canonical})
```

The output is shown in Figure 3. The model response is plotted against synthetic data with 4% Gaussian noise added to it, found in **S.data**. Data is stored in **S.data** in the format

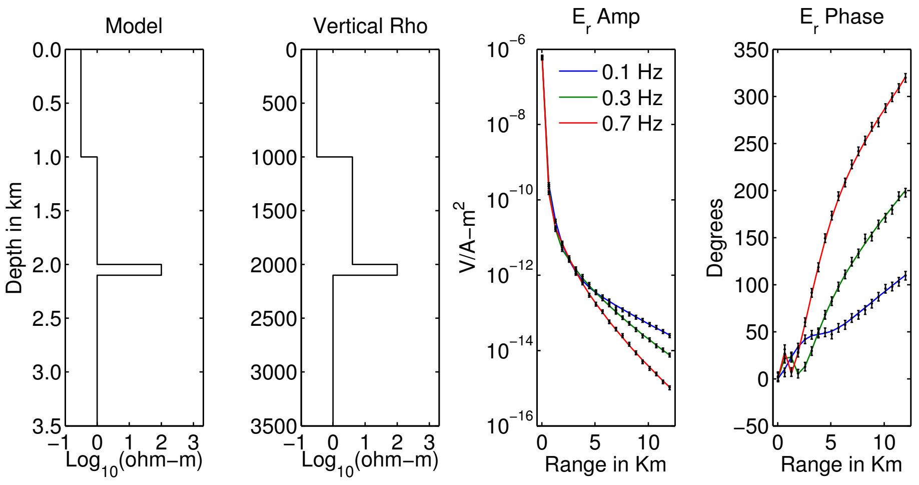


Figure 3: E_r CSEM field response due to anisotropic 1D canonical model at 0.1, 0.3 and 0.7 Hz

$[E_r, H_\beta, E_z, E_\beta, H_r, H_z]$. Each column corresponds to the frequencies found in **S.f** with the rows corresponding to **S.r**. For the 3 frequencies of E_r data, we can do the following:

```
 S.data(:,1:3)

ans =

   5.8373e-07 + 1.3582e-08i   6.0015e-07 + 2.2846e-08i   6.3097e-07 + 2.8582e-08i
   2.5347e-10 + 2.9425e-11i    2.093e-10 + 8.2025e-11i   1.3323e-10 + 8.0802e-11i
   2.5047e-11 + 1.0976e-11i   1.5379e-11 + 6.3073e-12i   1.7962e-11 + 2.2571e-12i
   5.6002e-12 + 3.6602e-12i   4.8051e-12 + 4.6526e-13i   6.3158e-12 + 3.3052e-12i
   1.9258e-12 + 1.7327e-12i   2.4273e-12 + 5.4866e-13i   1.4533e-12 + 2.5274e-12i
   7.6952e-13 + 8.1587e-13i   1.2832e-12 + 7.2558e-13i  -2.8728e-14 + 1.2274e-12i
   4.6868e-13 + 5.1802e-13i    6.082e-13 + 7.0361e-13i  -3.0039e-13 + 5.5553e-13i
   3.2423e-13 + 3.7597e-13i   2.1547e-13 + 5.3167e-13i  -2.5263e-13 + 1.5032e-13i
   2.3574e-13 + 2.5362e-13i   4.8497e-14 + 3.6217e-13i  -1.6968e-13 + 1.9726e-14i
   1.5682e-13 + 2.1546e-13i  -3.2411e-14 + 2.3678e-13i  -1.0522e-13 - 2.6358e-14i
   1.1252e-13 + 1.7515e-13i  -5.8126e-14 + 1.5254e-13i  -5.1075e-14 - 2.7825e-14i
   5.8992e-14 + 1.2464e-13i  -5.8735e-14 + 8.6374e-14i  -2.5015e-14 - 2.7412e-14i
   4.5225e-14 + 1.1094e-13i  -5.1239e-14 + 5.4757e-14i  -1.1811e-14 - 2.1835e-14i
    2.732e-14 + 8.9853e-14i  -4.5117e-14 + 2.7146e-14i  -4.2858e-15 - 1.4142e-14i
   1.3557e-14 + 7.5448e-14i  -3.1807e-14 + 1.7579e-14i  -3.6276e-16 - 8.7203e-15i
   3.2361e-15 + 6.3098e-14i  -2.3592e-14 + 6.8789e-15i   2.0026e-16 - 5.0276e-15i
  -4.9229e-15 + 4.9249e-14i  -1.9054e-14 +  3.057e-15i   1.0135e-15 - 3.2163e-15i
  -5.7127e-15 + 4.0955e-14i  -1.3402e-14 + 9.8902e-17i   1.1647e-15 - 2.0989e-15i
  -7.2462e-15 + 3.1485e-14i  -1.0119e-14 - 2.4853e-15i   9.2419e-16 - 1.1533e-15i
  -8.3547e-15 + 2.3563e-14i  -7.2441e-15 -  2.307e-15i     7.88e-16 - 6.6827e-16i
```

The data noise is given in **S.sd**. It is 4% or 0.04 in this example, but could also be a matrix of the same size as **S.data** with the data noise std. deviation on the real or imaginary component for each station. **S.data** is always of the size $nRx \times nFreq$ with a NaN indicating null values.

You will have to modify **plot_model_field.m** if you want to plot other components besides Er! You can plot a cell of models and their responses using this command within the curly braces, like so

```
plot_model_field(S,{model1;model2;...;modelN})
```

To simply plot a single model, one can do the following and the results are shown in Figure 4

```
>> figure
>> plot_model(S,canonical,'H')
>> plot_model(S,canonical,'V',2)
>> set(gca,'ydir','reverse')
>> xlabel('log10 (ohm-m)')
>> ylabel('Depth (m)')
```

0.3 Forward modeling codes

An easy way to obtain model responses in a MATLAB vector is to use **get_field.m**. It should already be in your path if you've included the **include_matlab** directory. All you have to do is:

```
>> [Er,Eb,Hr,Hb,Ez,Hz] = get_field(S,canonical);
>> Er

Er =

   6.1319e-07 + 1.2766e-09i   6.1301e-07 + 3.7407e-09i   6.1243e-07 + 8.4244e-09i
   2.4724e-10 + 4.5132e-11i   1.9548e-10 + 7.8377e-11i   1.3092e-10 + 7.0764e-11i
   2.5286e-11 + 1.0808e-11i   1.5106e-11 + 6.3288e-12i    1.813e-11 + 1.7538e-12i
   5.5662e-12 + 3.7541e-12i   4.8384e-12 + 4.3135e-13i   6.2128e-12 + 3.2539e-12i
   1.8737e-12 + 1.6687e-12i   2.4583e-12 + 5.6666e-13i    1.464e-12 +  2.409e-12i
   8.5256e-13 + 8.7387e-13i   1.2225e-12 + 7.2495e-13i   2.0419e-15 + 1.2407e-12i
```

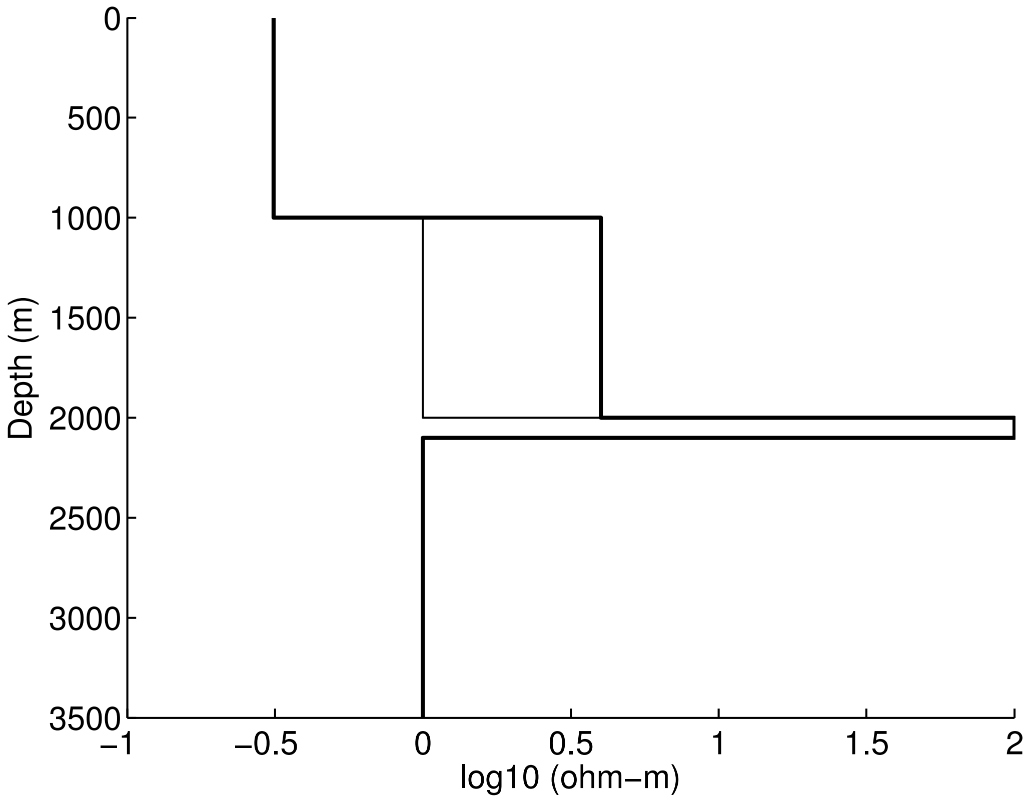


Figure 4: Plotting a given model

```
    4.842e-13 + 5.2405e-13i     5.48e-13 +  6.444e-13i  -2.9912e-13 + 5.2363e-13i
   3.1525e-13 + 3.5567e-13i   2.0309e-13 + 4.8769e-13i  -2.6611e-13 + 1.7348e-13i
    2.179e-13 + 2.6591e-13i   3.9566e-14 + 3.3992e-13i  -1.7446e-13 + 2.8444e-14i
   1.5212e-13 + 2.1094e-13i  -3.0162e-14 + 2.2557e-13i  -9.9795e-14 - 2.0086e-14i
   1.0429e-13 + 1.7206e-13i   -5.398e-14 + 1.4455e-13i  -5.2313e-14 - 2.9392e-14i
   6.8939e-14 + 1.4151e-13i  -5.6726e-14 + 8.9777e-14i  -2.5394e-14 - 2.5549e-14i
   4.3076e-14 + 1.1619e-13i  -5.0758e-14 +  5.382e-14i  -1.1188e-14 - 1.8909e-14i
   2.4557e-14 + 9.4827e-14i  -4.2027e-14 + 3.0763e-14i  -4.1182e-15 - 1.2994e-14i
   1.1666e-14 + 7.6821e-14i  -3.3239e-14 + 1.6301e-14i  -7.8998e-16 - 8.5642e-15i
   2.9735e-15 + 6.1772e-14i   -2.546e-14 + 7.4694e-15i   6.2114e-16 - 5.4734e-15i
   -2.638e-15 + 4.9323e-14i  -1.9048e-14 + 2.2668e-15i   1.1169e-15 - 3.4088e-15i
   -6.048e-15 + 3.9101e-14i  -1.3924e-14 - 5.8052e-16i   1.1408e-15 - 2.0671e-15i
  -7.9165e-15 + 3.0789e-14i  -9.9678e-15 - 1.9801e-15i   9.7605e-16 - 1.2147e-15i
  -8.7383e-15 + 2.4129e-14i  -7.0707e-15 -  2.599e-15i   8.2567e-16 - 6.7779e-16i

>>
```

Say you wished to plot Ez, simply do the following

```
figure
semilogy(S.r, abs(Ez))
xlabel ('Range in m')
ylabel('E_z amp')
legend(num2str(S.f'))
```

The results can be seen in Figure 5. To model with a given receiver azimuth (clockwise from the transmitter) or transmitter dip (+ve below horizontal), one needs to specify these values in degrees in S.RxAz and S.TxDip.

0.4 Misfit calculation

To find the misfit between the observed data and a model, say the anisotropic canonical model, simply do the following:

```
>> m = csem_misfit(canonical,S)

m =

       62.845       1.0234
```

The first value is the χ^2 misfit/2 and the second value is the RMS misfit. Given 20 Rx sites, 3 frequencies and both real and imaginary data, this equals 120 data points. Thus,

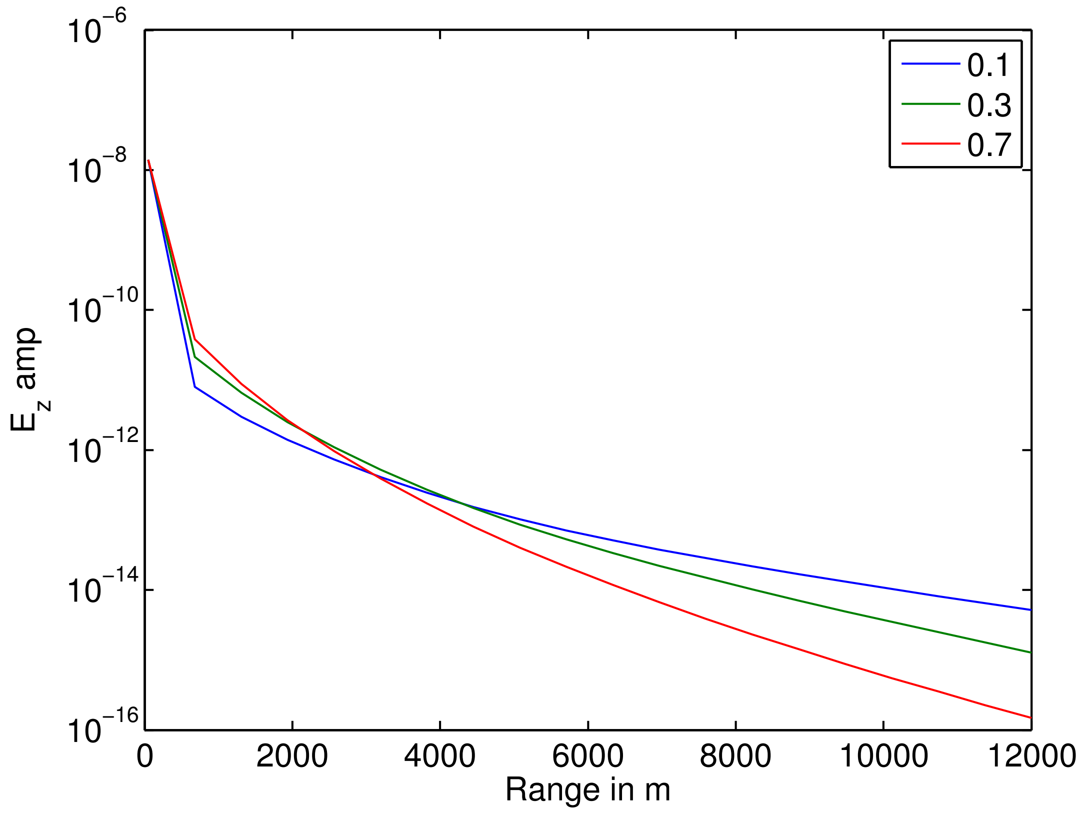


Figure 5: E_z CSEM field response due to the anisotropic 1D canonical model at 0.1, 0.3 and 0.7 Hz

the χ^2 value of 126 is in the correct ballpark. The RMS value is simply found by dividing the χ^2 value by the number of data points, then taking root.

Running the RJ-MCMC code with Parallel Tempering

0.5 Algorithm details and references

Before running the RJ-MCMC code, a few details are necessary. We are using the basic algorithm described in [3] with the parallel tempering implementation given in [2]. We use parallel tempering to be able to cut down on the number of parallel chains required to sample the model space. This allows us to run the algorithm on only 1 CPU, avoiding the use of parallel MATLAB, the licenses for which are quite expensive. However, the parallel chains are being sequentially run in my code, as MATLAB is quite slow when it comes to parallelizing an inner loop, which is what we do when using parallel tempering. The basic algorithm holds, though, and I welcome attempts to modify my parallel tempering code such that it runs using OpenMPI or OpenMP and some low level language.

0.5.1 Trans-dimensional Bayesian inversion

In the simplest terms, the objective of trans-dimensional Bayesian inversion is to sample the distribution given by $p(\mathbf{m}|\mathbf{d})$ or the probability of the model, given the observed data. This is achieved with the reversible jump algorithm by drawing candidate models from a proposal distribution. These models are then examined to see if they fall within geophysically sensible uniform bounds of resistivity and depth. If a proposed model falls outside the prior distribution, it is rejected and the Markov chain retains the previous model as the next model. If a proposed model is within the prior bounds, an acceptance probability is calculated using a ratio of the proposal probability, the prior probability and the likelihood of the candidate model with respect to the previous model. The proposed model is either accepted with the calculated probability and it becomes the next model in the chain, or it is rejected and the previous model is retained as the next in the chain. Complete details of this process are given in section 2 of [3] for the interested reader. As the algorithm proceeds, hundreds of thousands of models are sampled, with a data-driven addition or deletion of layers ('birth/death' in RJ-MCMC parlance), such that a chain of models, most of which fit the data well within the noise are retained at the end.

0.5.2 Specifying prior bounds

The prior bounds within which to sample are specified in the structure S. If we navigate to the **example_aniso** directory, and do the following, we'll see these prior bounds, printed in red.

```
>> S = load ('misfit_parameters_canonical_ANISO')

S =

             ans: [0x1 double]
               f: [0.1000 0.3000 0.7000]
             zTx: 990
               r: [1x20 double]
             zRx: [999 999 999 999 999 999 999 999 999 999 999 999 999 999 999]
               z: [-1000000 0 1000]
            data: [20x18 double]
              sd: 0.0400
      components: [1 0 0 0 0 0 0 0]
             rho: [2x2 double]
            kMin: 1
            kMax: 30
             zMin: 1002
             zMax: 3500
             rhMin: -1
             rhMax: 3.3000
             rvMin: -1
             rvMax: 3.3000
            rSD2: 0.0200
            rSD1: 0.1000
          MoveSd: 2.2000
            hMin: 0
    chooseconfig: 1
       isotropic: 0
            RxAz: 0
           TxDip: 0
```

kMin and kMax specify the minimum and maximum number of interfaces a model can have. The minimum number of interfaces is 1, which implies a model with a block above a half space. zMin and zMax specify the maximum and minimum depths at which to place an interface, rhMin and RhMax, rvMin and rvMax are the minimum and maximum log10 resistivities in the horizontal and vertical directions for each layer.

S.components tells the RJ-MCMC code which field component(s) to invert. I've only tried one component at a time, but feel free to modify the flags in this vector to invert more than one component. 1 is for a given component, 0 indicates no. The order of S.components is $[E_r, H_\beta, E_z, E_\beta, H_r, H_z]$. [0 0 1 1 0 0] would be for E_z and E_β together.

S.isotropic = 0 means the inversion will be anisotropic. A value of 1 would be for isotropic inversions.

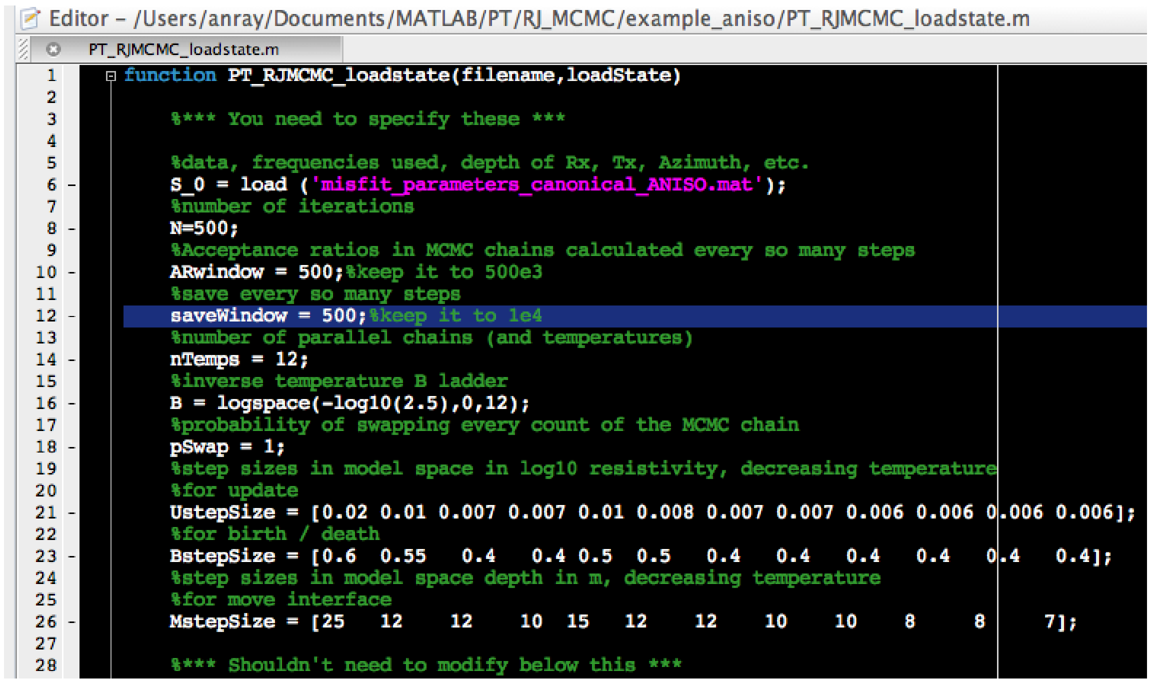
0.6 Running the RJ-MCMC code

Make sure that the **include_matlab** and **mex** directories are in your MATLAB path (Figure 2). Navigate to the **example_aniso** folder, and simply issue the following commands at the MATLAB prompt

```
>>PT_RJMCMC_loadstate('T2p5_12Temp_12Chains');
```

On my 2.7 GHz machine, this took me 115 seconds. If you've gotten till here, everything works. **T2p5_12Temp_12Chains** is the filename for the RJ-MCMC chains. You'll notice that in the directory you ran the code in, there are now a bunch of ***status** and MATLAB ***mat** files. We've basically run the code for a measly 500 iterations, and the status files show us the progress of the inversion as it runs. The .mat files contain the final results which we are interested in. Now open the **PT_RJ_MCMC_loadstate.m** file. It should look like Figure 6. These are the only lines you should modify, and only the right hand sides, till you know what you're doing. On line 6 you specify the data and inversion parameters, in a structure S_0, in this case **misfit_parameters_canonical_ANISO**. The number of iterations should be set to something like at least 500e3, which for a respectable 1D problem at 3 frequencies, takes between 20 to 24 hours to complete. The saveWindow should be set to values like 1e4 to make sure we aren't writing to disk too often. The most important parameters to set up are the number of temperatures (line 14), what they are (line 16), and the model space sampling step sizes for updating a layer resistivity (line 21, in log10 ohm-m), creating a new layer (line 22, in log10 ohm-m) and moving an interface (line 26, in m).

The highest temperature, here 2.5, anneals the likelihood (think of it as the probability of the misfit space) such that all local minima are escaped from easily. To do this, step sizes in model space must be large at large temperatures. At low temperatures, we want



Editor – /Users/anray/Documents/MATLAB/PT/RJ_MCMC/example_aniso/PT_RJMCMC_loadstate.m

PT_RJMCMC_loadstate.m

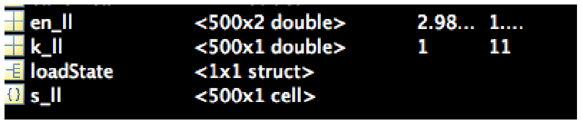
```
function PT_RJMCMC_loadstate(filename,loadState)

    %*** You need to specify these ***

    %data, frequencies used, depth of Rx, Tx, Azimuth, etc.
    S_0 = load ('misfit_parameters_canonical_ANISO.mat');
    %number of iterations
    N=500;
    %Acceptance ratios in MCMC chains calculated every so many steps
    ARwindow = 500;%keep it to 500e3
    %save every so many steps
    saveWindow = 500;%keep it to 1e4
    %number of parallel chains (and temperatures)
    nTemps = 12;
    %inverse temperature B ladder
    B = logspace(-log10(2.5),0,12);
    %probability of swapping every count of the MCMC chain
    pSwap = 1;
    %step sizes in model space in log10 resistivity, decreasing temperature
    %for update
    UstepSize = [0.02 0.01 0.007 0.007 0.01 0.008 0.007 0.007 0.006 0.006 0.006 0.006];
    %for birth / death
    BstepSize = [0.6  0.55   0.4   0.4 0.5  0.5   0.4   0.4   0.4   0.4   0.4   0.4];
    %step sizes in model space depth in m, decreasing temperature
    %for move interface
    MstepSize = [25   12    12    10  15   12    12    10    10    8     8     7];

    %*** Shouldn't need to modify below this ***
```

Figure 6: First few lines of PT_RJMCMC_loadstate



en_ll	<500x2 double>	2.98...	1....
k_ll	<500x1 double>	1	11
loadState	<1x1 struct>		
s_ll	<500x1 cell>		

Figure 7: Variables of interest in target chain **T2p5_12Temp_12Chains_12.mat**

to finely sample the model space, especially at the unbiased, target temperature T = 1, so step sizes should be smaller. 12 interacting chains at different temperatures are run, with B(1) = log10(2.5) and B(12) = log10(1) where B is the inverse temperature. In UstepSize, BstepSize and MstepSize, step sizes corresponding to hotter chains are to the left. These 12 communicating chains sample the misfit space at different scales, ensuring thorough sampling by exchanging models at every step. The number of chains, the temperatures to use and the step sizes are problem dependent, and details can be found in [2]. In this implementation, we allow any chain to exchange models with any other chain, not just adjacent ones.

Inferences are to be made from only the unbiased chain, in this case the chain at temperature 12, T=1, corresponding to the file **T2p5_12Temp_12Chains_12.mat**

0.7 Examining the results

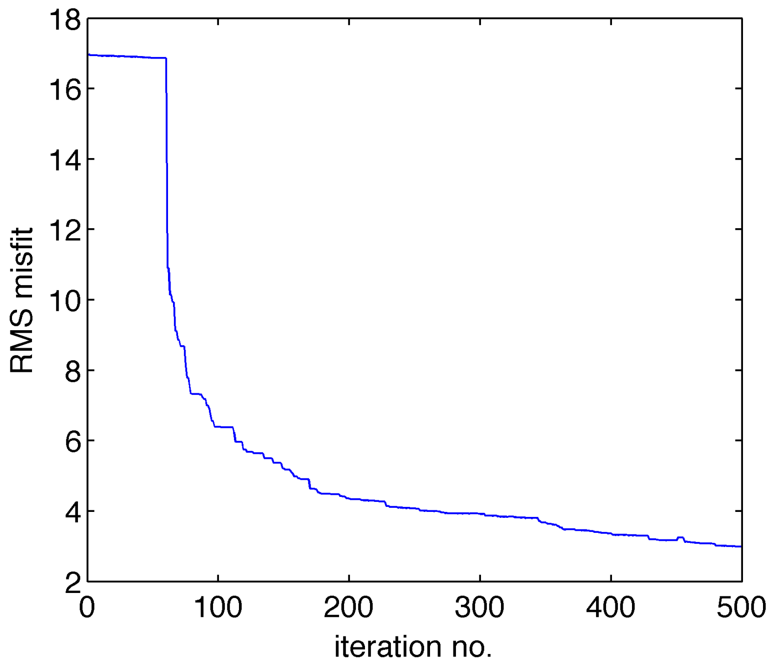
If we now run the command

```
>> load('T2p5_12Temp_12Chains_12.mat')
```

we will see our variables of interest (Figure 7). s_ll is a structure with 500 models, with s{1} the starting model, and s{500} the last one sampled. k_ll contains the number of interfaces for each of these models, and en_ll contains in the first column the χ^2 misfit/2 and the RMS misfit in the second column for each of these models.

Just to see how this works, we can issue the following commands and see the results in Figure 8

```
figure
subplot (2,1,1)
plot(en_ll(:,2))
xlabel('iteration no.')
ylabel('RMS misfit')
subplot (2,1,2)
S = load ('misfit_parameters_canonical_ANISO');
plot_model(S_ll,s_ll1,'V')
plot_model(S_ll,s_ll500,'V',2)
load canonical
plot_model(S,canonical,'V',3)
set(gca,'ydir','reverse')
xlabel('log10 (ohm-m)')
ylabel('Depth (m)')
legend('start','end','truth')
```



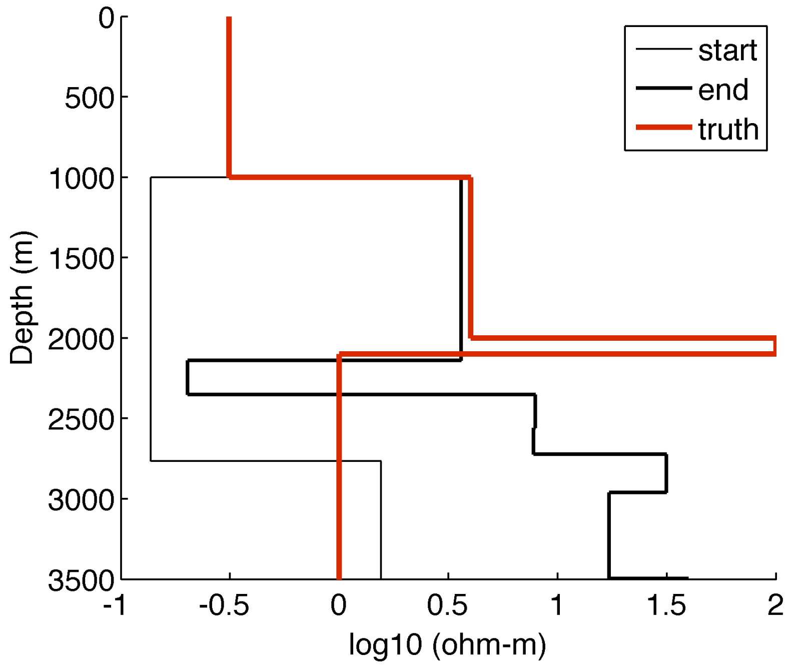


Figure 8: RMS misfit with 500 iterations (top) and the first and last models sampled (bottom)

We start from a very high misfit and climb down to lower values of misfit, and the models start looking more and more like the true model. Of course, we need to sample far longer to get useful results, and the results of such sampling are shown in the next chapter.

0.8 Modifying RJ-MCMC code for other problems

Finally, you could edit the functions to do with forward modeling and misfit calculation in **PT_RJ_MCMC_loadstate.m**. These functions are get_field() and csem_misfit().

Worked out examples and their visualization

0.9 Anisotropic canonical model example

If we navigate to the **worked_out_examples/anisotropic** directory, making sure that our included paths are set as in Figure 2 we can now run the post inversion processing codes. In this folder, for the noisy synthetic data shown in Figure 3, we have performed an inversion for 500,000 iterations. This took approximately a day to run on a 2.7 GHz machine on 1 CPU. We could load the results from the last numbered chain, the target chain at T=1 like so:

```
clear
S = load ('misfit_parameters_canonical_ANISO');
load T2p5_12Temp12Chains_AnyEx_12
```

0.9.1 Avoiding the burn in samples

The first thing to do is to avoid the 'burn in' samples, that are at a very high misfit and may bias the posterior distribution if we sample for a short time. To figure out what the burn in length is, we should plot the χ^2 values and make sure we're sampling a stationary series of misfit values through the likelihood. We can issue the following commands:

```
figure
plot(10e3:500e3,en_ll(10e3:500e3,1))
hold on
plot(10e3:100e3,en_ll(10e3:100e3,1),'r')
xlabel('iteration no.')
ylabel('\chi^2 misfit/2')
```

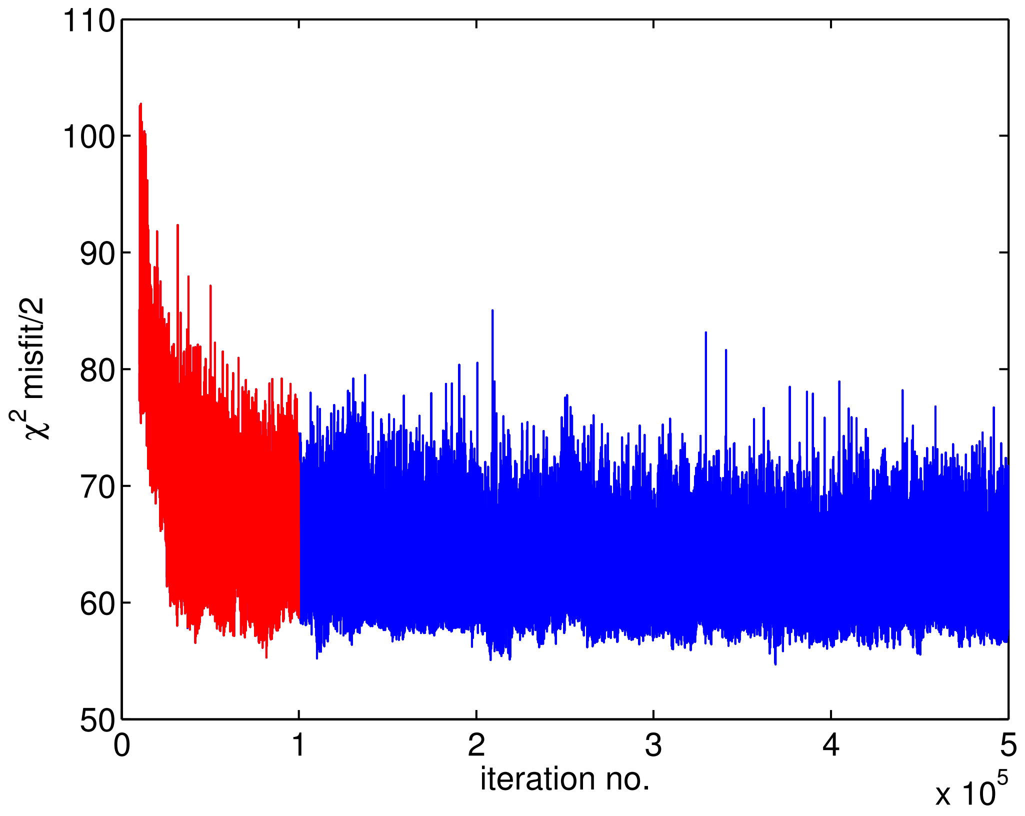
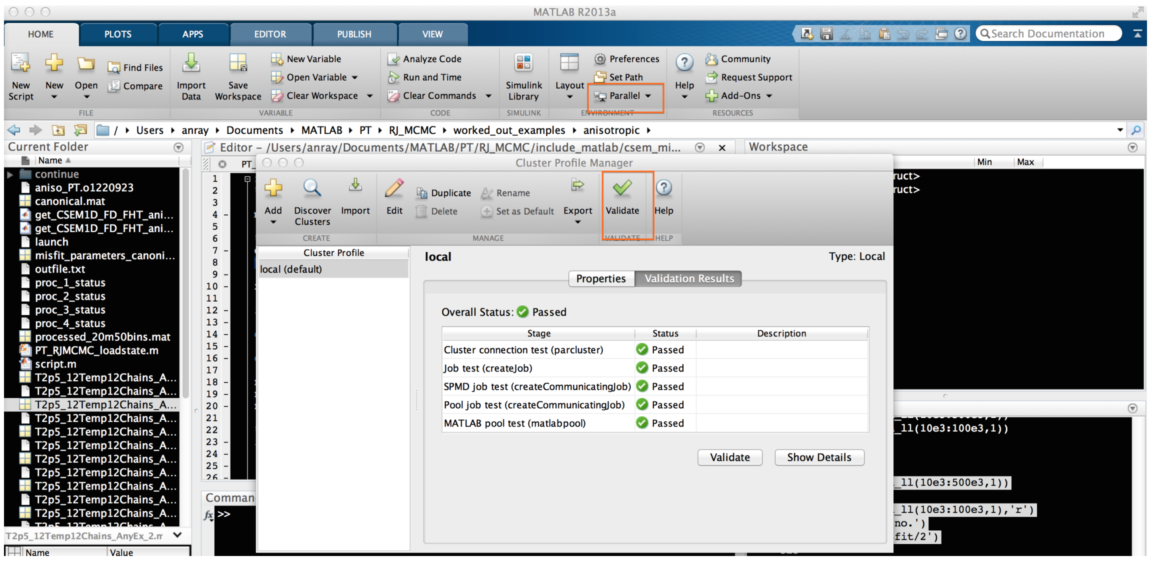


Figure 9: Avoiding the non stationary samples in red



local Type: Local

Overall Status: ✓ Passed

Stage	Status	Description
Cluster connection test (parcluster)	✓ Passed	
Job test (createJob)	✓ Passed	
SPMD job test (createCommunicatingJob)	✓ Passed	
Pool job test (createCommunicatingJob)	✓ Passed	
MATLAB pool test (matlabpool)	✓ Passed	

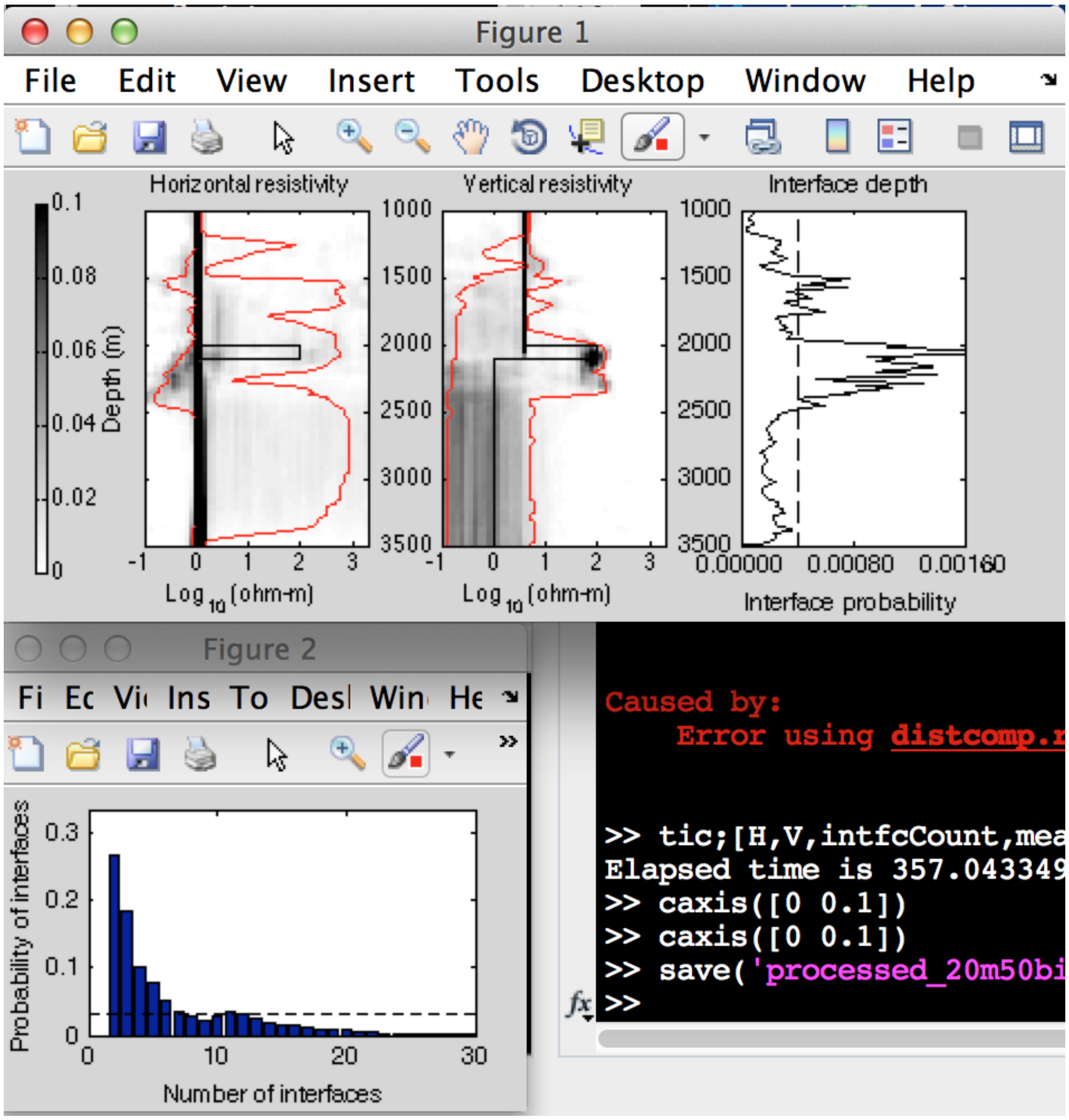
Figure 10: Validating parallel MATLAB 2013a

and obtain a plot like in Figure 9 where we avoid the non stationary part, shown in red. Thus the burn-in length is 100,000 samples. We can then run our post-inversion processing code, to look at the posterior ensemble $p(\mathbf{m}|\mathbf{d})$. To do this, we run a parallel MATLAB function **plot_rjmcmc_new_parallel**. To make sure that we have parallel MATLAB set up properly, we have to make sure that we manage cluster profiles and validate them as shown in Figure 10. If you don't have parallel MATLAB, you can of course plot results, but it will be a lot slower for you. This is what I do on my 4 core machine with MATLAB 2013a, after cluster validation.

```
matlabpool
Starting matlabpool using the 'local' profile ... connected to 4
    workers.
S = load ('misfit_parameters_canonical_ANISO');
load canonical
[H,V,intfcCount,meanH,meanV,medianH,medianV,kOut]=
    plot_rjmcmc_new_parallel(4,s_ll,k_ll,100e3,2,canonical,20,50,'
    anisotropic','NOnormalize',S);
```

I warn you, this step is a little long - it took me 6 minutes on 4 cores @2.7GHz, and here the results are in Figure 11 after modifying the caxis to [0 0.1] for the images. I've saved the results in the file **processed_20m50bins.mat** so you don't have to do it, like so:

```
save('processed_20m50bins.mat','H','V','intfcCount','meanH','meanV
    ','medianH','medianV','kOut')
```



```
Caused by:
    Error using distcomp.r

>> tic;[H,V,intfcCount,mea
Elapsed time is 357.043349
>> caxis([0 0.1])
>> caxis([0 0.1])
>> save('processed_20m50bi
>>
```

Figure 11: Rough and ready posterior plot

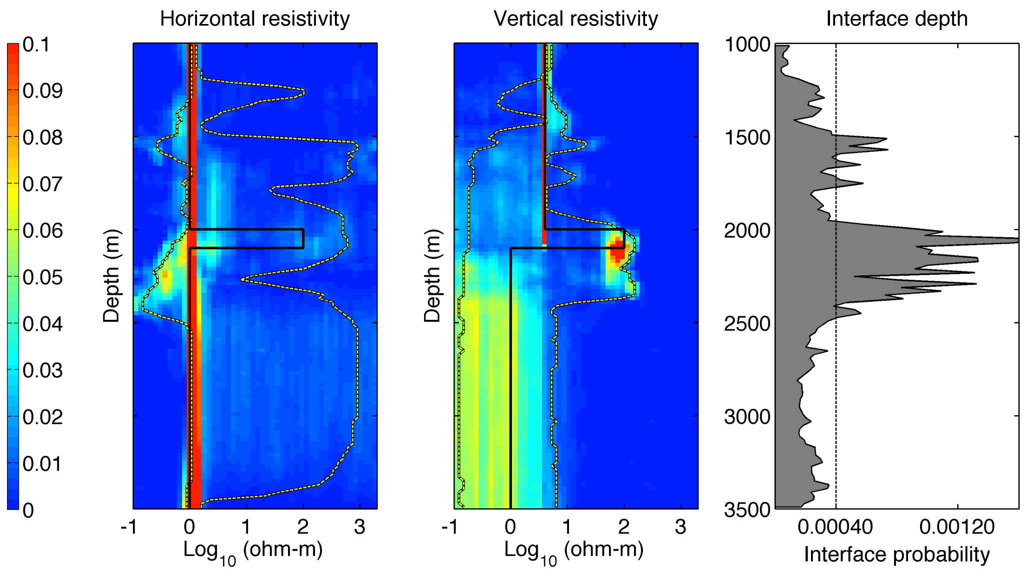


Figure 12: Pretty posterior plot after 500e3 iterations

In the function above, H and V are the horizontal and vertical resistivity PDFs with depth. The value of 4 on the right hand side corresponds to the number of cores used, 100e3 is the burn in value, 2 is the decimation of the chain ('thinning value'), 20 corresponds to the size of the depth bins in meters, 50 is the number of resistivity bins in every 20 m depth depth interval. canonical plots the model **canonical** in memory. 'anisotropic' plots are required for anisotropic inversions. It would be changed to 'isotropic', for isotropic inversion plots. 'NOnormalize' does not normalize the PDFs at every depth to the max value, remove the NO if you'd like to normalize to the max value at every depth, which I think could be misleading.

0.10 Looking at pretty plots later

Since we've saved the results, we can load these post processed results at a moment's notice and make our managers happy with function **plot_rjmcmc_HVk_processed** :

```
clear
S = load ('misfit_parameters_canonical_ANISO');
load canonical
load processed_20m50bins
plot_rjmcmc_HVk_processed(H,V,intfcCount,meanH,meanV,medianH,medianV
    ,kOut,S,canonical,'anisotropic','NOnormalize')
caxis([0 0.1])
```

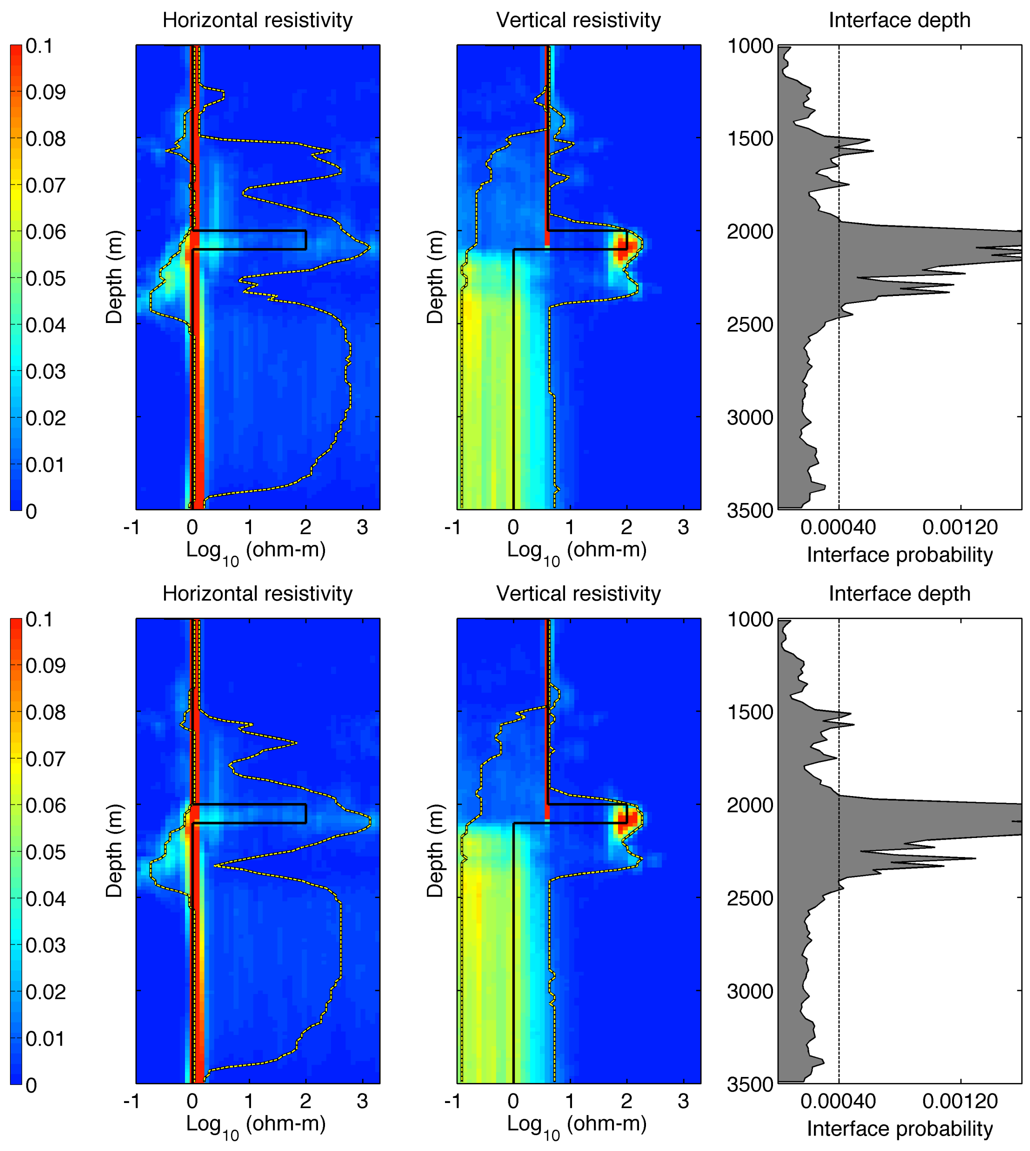


Figure 13: Pretty posterior plot after 750e3 iterations (top) and 1000e3 iterations (bottom)

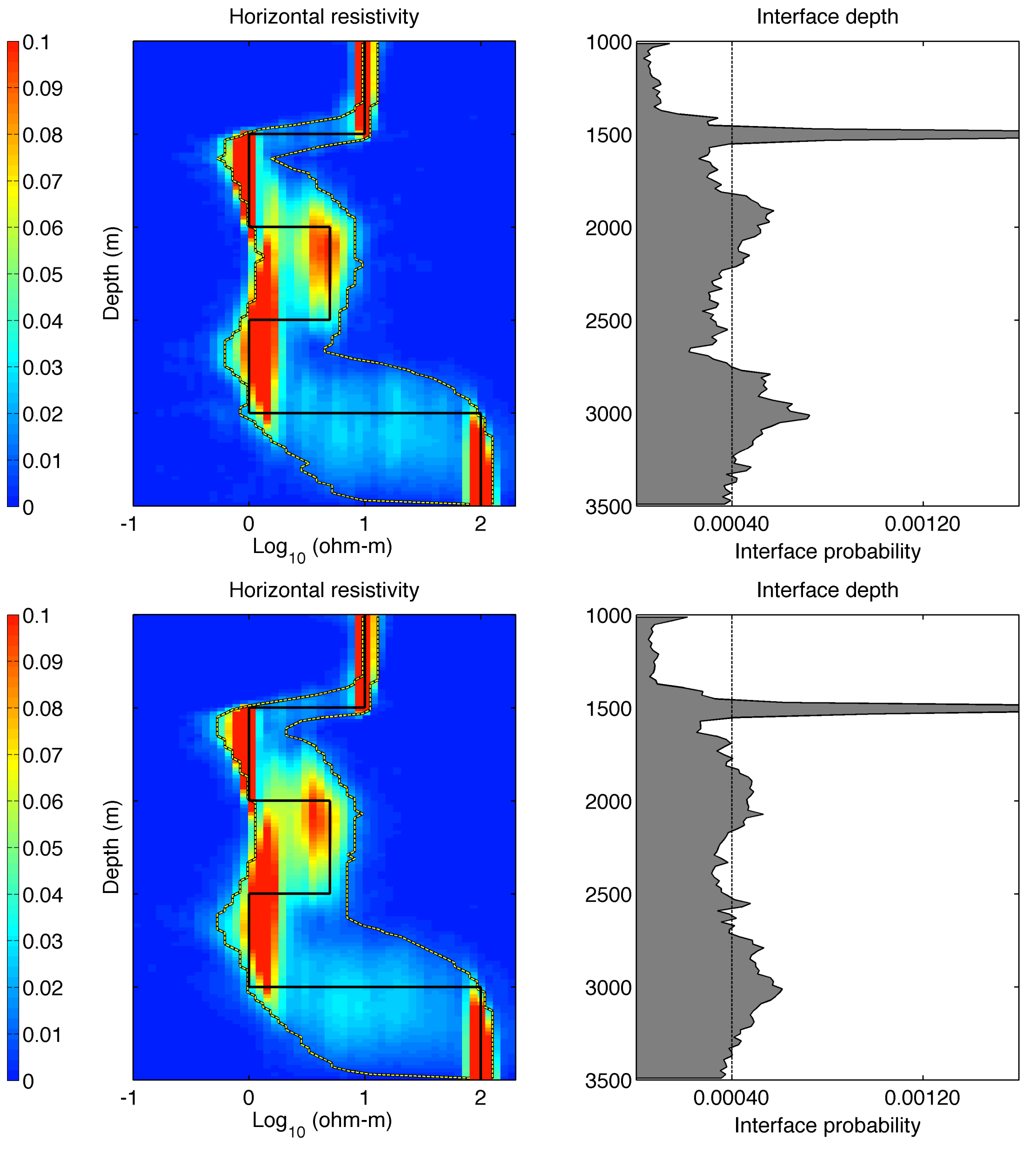


Figure 14: Posterior distributions for Mike Hoversten's model, shown in black, after 500e3 samples (top) and 1000e3 samples (bottom)

The results at the end of of 500e3 iterations are shown in Figure 12

0.11 How do we know if we've obtained enough samples?

The only way to tell is to run more iterations and see if there is a change in the posterior distribution. The code can continue where it left off. So as to not overwrite anything, I created a new directory called **continue** and copied **T2p5_12Temp12Chains_AnyEx_12.mat** and **misfit_parameters_canonical_ANISO.mat** files, along with **PT_RJMCMC_loadstate.m** to this directory. I ran the code in this **continue** directory again for 500e3 iterations using the following command, giving the program the previous RJ-MCMC state using the saved structure **loadState**. Beware, this step took another 20 hours!

```
load ('T2p5_12Temp12Chains_AnyEx_12','loadState')
PT_LikeFixed_RJMCMC_loadstate_AnyEx('T2p5_12Temp12Chains_AnyEx',
    loadState);
```

The results for the continued target chain are in **T2p5_12Temp12Chains_AnyEx_12.mat**. We need to concatenate the variables from the first and continued chains of 500e3 samples each. We do this in the following manner, getting back to the parent directory **worked_out_examples/anisotropic**

```
clear
S = load ('misfit_parameters_canonical_ANISO');
load canonical
s = []; k = []; en = [];
load T2p5_12Temp12Chains_AnyEx_12
s = [s;s_ll]; k = [k;k_ll]; en = [en;en_ll];
```

Now load the continued chains

```
cd continue
load T2p5_12Temp12Chains_AnyEx_12
s = [s;s_ll]; k = [k;k_ll]; en = [en;en_ll];
```

I processed only the first 750e3 samples with a burn in of 100e3 in the following manner, saving the results in **750Kconcat_processed_20m50bins.mat**

```
[H,V,intfcCount,meanH,meanV,medianH,medianV,kOut]=
    plot_rjmcmc_new_parallel(4,s(1:750e3),k(1:750e3),100e3,2,
    canonical,20,50,'anisotropic','NOnormalize',S);
save('750Kconcat_processed_20m50bins.mat','H','V','intfcCount','
    meanH','meanV','medianH','medianV','kOut')
```

And then I processed the entire concatenated chain, saving the results in **1000Kconcat_processed_20m50bins.mat**

```
[H,V,intfcCount,meanH,meanV,medianH,medianV,kOut]=
   plot_rjmcmc_new_parallel(4,s,k,100e3,2,canonical,20,50,'
   anisotropic','NOnormalize',S);
save('1000Kconcat_processed_20m50bins.mat','H','V','intfcCount','
   meanH','meanV','medianH','medianV','kOut')
```

Plotting these two results, using the following commands,

```
load 750Kconcat_processed_20m50bins
plot_rjmcmc_HVk_processed(H,V,intfcCount,meanH,meanV,medianH,medianV
   ,kOut,S,canonical,'anisotropic','NOnormalize')
caxis([0 0.1])
```

And

```
load 1000Kconcat_processed_20m50bins
plot_rjmcmc_HVk_processed(H,V,intfcCount,meanH,meanV,medianH,medianV
   ,kOut,S,canonical,'anisotropic','NOnormalize')
caxis([0 0.1])
```

we then obtain Figure 13. Since the posterior PDFs at 750e3 and 1000e3 samples are very nearly the same, we have converged.

0.12 Mike Hoversten's isotropic nightmare model example

First seen in [4], this model is quite hard to invert, as there are two modes that keep cropping up. Both deterministic and probabilistic inversion methods have trouble sampling the model space. This model has been tackled in some detail in [2]. In the folder **worked_out_examples/isotropic**, in exactly the same manner as in the previous section, I have worked out the solution and carried out the probabilistic inversion. Use the right **misfit_parameters*mat** file and ***processed*mat** files, you should be able to plot the results without carrying out the whole inversion yourself (which you are also welcome to do!). I leave you now with the dreaded words - 'it is left as an exercise to the reader to reproduce Figure 14'

Bibliography

[1] Kerry Key. Is the fast Hankel transform faster than quadrature? *Geophysics*, 77(3):F21–F30, 2012.

[2] Anandaroop Ray, David L Alumbaugh, G Michael Hoversten, and Kerry Key. Robust and accelerated Bayesian inversion of marine controlled-source electromagnetic data using parallel tempering. *Geophysics*, 78(6):E271–E280, 2013.

[3] Anandaroop Ray and Kerry Key. Bayesian inversion of marine CSEM data with a trans-dimensional self parametrizing algorithm. *Geophysical Journal International*, 191:1135–1151, October 2012.

[4] W Trainor and G M Hoversten. Practical challenges of stochastic inversion implementation for geophysical problems. In *2009 SEG Annual Meeting*, 2009.