GPGN 598A: Final Project

Specster: a specfem/FWI harness

Derrick Chambers

5/10/23

1 Introduction

This report fulfills the term project requirement for GPGN 598A: Full Waveform Inversion (FWI) taken at the Colorado School of Mines in spring of 2023. In order to learn the important implementation details of FWI workflows and the intricacies of Specfem2D, which I used as the wavefield solver, I opted to create a python package for conducting this project rather than using existing workflow solutions, such as seisflows. The python implementation is called specster and it consists of nearly 2000 lines of python implementation code and 500 lines of test code (with several hundred tests in total). This report will highlight the use of the Specster library in order to interact with Specfem2D and conduct a few iterations of a simple FWI. This package may be useful to other students in the future, but currently it is only a prototype which will required additional work before it is ready for public use.

2 Installation

Specster can be installed from git using pip (provided git is installed)

```
pip install git+https://github.com/d-chambers/specster
```

Next, the environment variable "SPECFEM_BIN_PATH" and "SPECFEM2D_PATH" must be set so specster can find specfem binaries and example directories, respectively. This can be done in one of the RC scripts to avoid manually setting these each time Specster is used.

3 Specfem interactions

Specster interacts specfem 2D through a control class known as `Control2d`. `Control2d` provides several useful features to:

- Plot experiment geometries
- Run meshing and simulations
- Parse receivers and sources into tables (dataframes)
- Change simulation parameters

The following subsections provides `Control2d` use examples.

3.1 Control2D initialization

There are three ways to initialize `Control2D` objects:

```
import specster as sp

# Load the base example then copies files to temporary directory
control_default = sp.Control2d().copy()
# loads an example included in the specfem2D directory by name
# or in the specster data directory and copies to a temp directory
control_example = sp.load_2d_example("Tape2007")
# Loads a specfem directory from a path (at least includes a DATA sub-directory)
# and parfile.
control = sp.Control2d("path/to/your/file")
```

3.2 Plotting simulation geometry

Visualizing the station/source geometry and background models is useful to quickly understand the simulation.

```
import specster as sp

# loads the base example then copies files to temporary directory
# this is needed because the mesher may need to be run to generate models.
control = sp.Control2d().copy()
control.plot_geometry(kernel=['vs', 'vp'])  # plots VP and VS
```

Sources are shown as red stars, receivers as triangles, and grid quantities (in this case P/S wave speed) are plotted in separated panels.

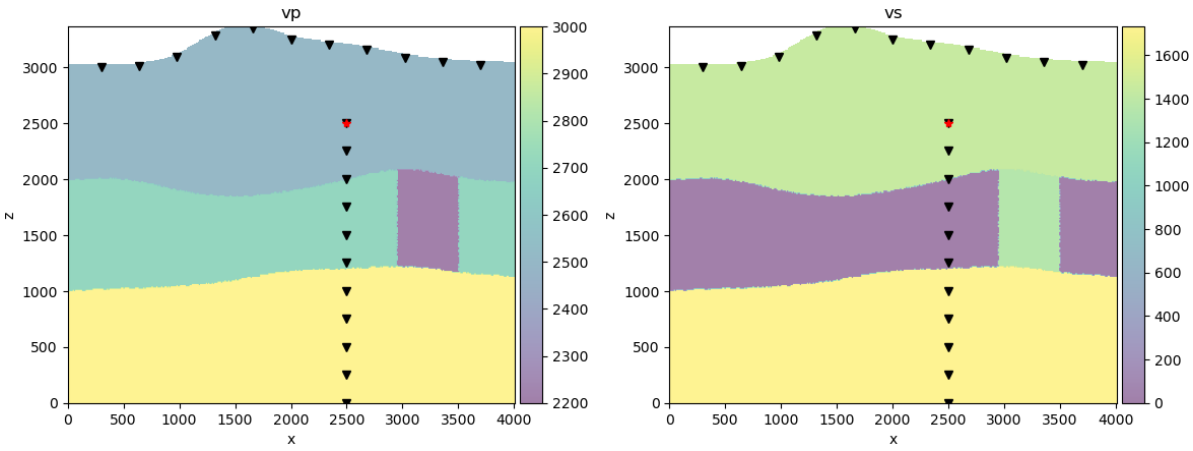


Figure 1: Basic geometry plot of default example

3.3 Running a forward simulation and output inspection

Forward simulations are run with the `Control2d.run` method. Specster pipes the Specfem2d output to the terminal upon calling this method. It also writes the output as a text file to the output directory.

`Control2d.run` is able to figure out which programs to run based on values in the par file. For example, if the par file specifies an external mesh, it will not run the xmesh2d.

```
import shutil
from pathlib import Path

import specster as sp

path = Path("outputs/run_example")

if path.is_dir():
    shutil.rmtree(path)

control = (
    sp.Control2d()  # default example
    .copy(path)  # make copy to path
    .prepare_fwi_forward()  # set params to forward mode
)

output = control.run()
```

The outputs of the runs are accessed via the `output` parameter which is a `Output2D` object. `Output2D` has many useful methods for accessing information about the run results. For example, `output.get_waveforms` loads the resulting waveforms as an obspy.Stream object.

```
st = output.get_waveforms()  # get obspy streams
st.select(component='Z')[:3].plot()  # plot first 3 streams
```

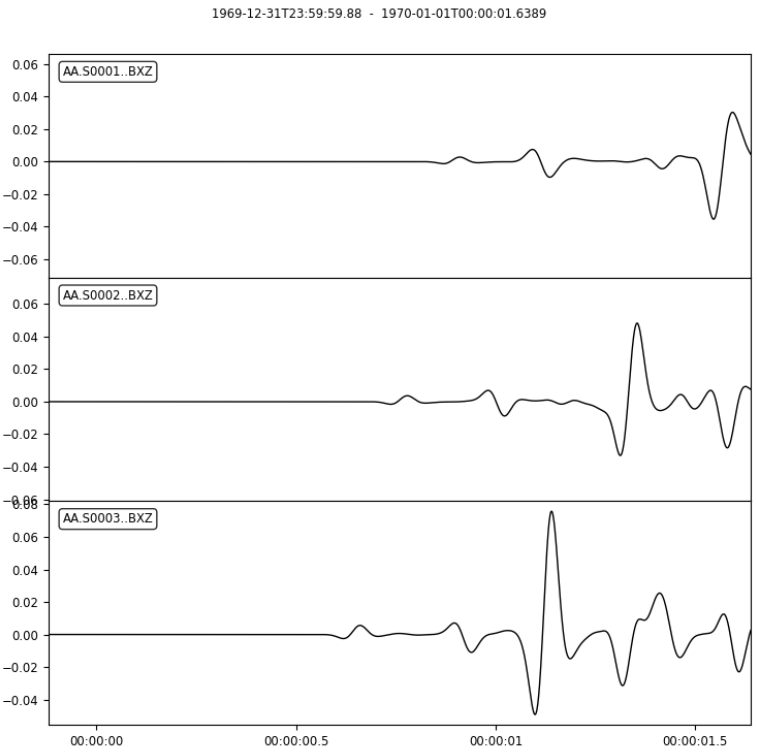


Figure 2: Plot of output streams

`Output2d` also contains important summary parameters that may help assess the health of the simulation:

```
stats = output.stats  # get obspy streams
stats.elements  # number of elements
stats.max_frequency_resolved  # the maximum freq form the source
stats.min_gll_distance  # min spacing of GLL points
stats.max_cfl  # max clf ratio. Should be less than 0.5 or so.
```

Another useful plot is the number of GLL points per shortest wavelength for different regions of the model. This can be created by `Output2d` like so:

```
output.plot_gll_per_wavelength_histogram()
```

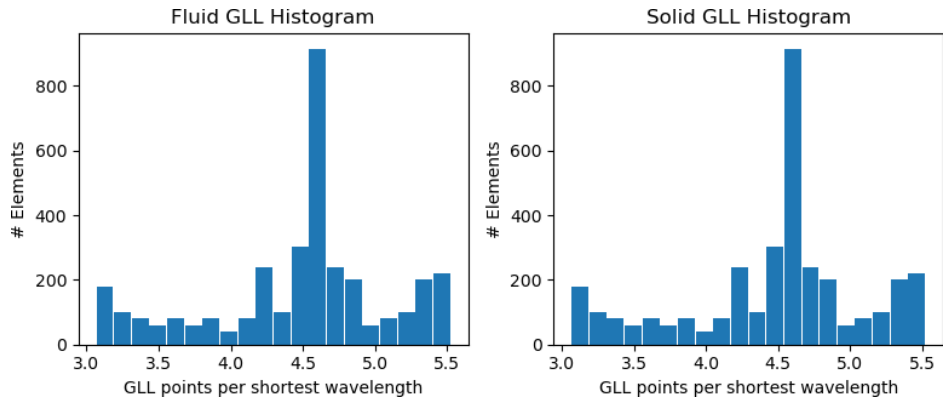


Figure 3: GLL per wavelength histograms

4 Modifying stations and sources

Stations and sources can be modified directly through the `station` and `sources` parameters, making it more repeatable and programmable to modify the experiment geometry than manually editing text files.

The following code will make a copy, remove all but one station, add another source, and then plot the experiment geometry.

```
control = control.copy()

control.stations = control.stations[:1]

new_source = control.sources[0].copy()
new_source.xs, new_source.zs = 1500, 1500
control.sources.append(new_source)

control.write(overwrite=True)

control.plot_geometry(kernel=['vs', 'vp'])
```

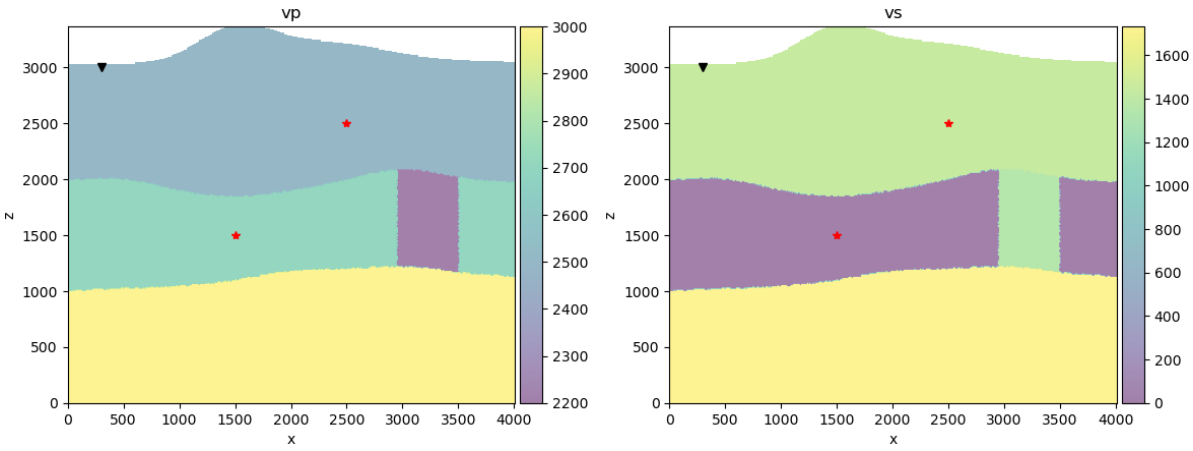


Figure 4: Modified geometry

5 Modifying the parfile

For many use cases, users shouldn't have to modify the parameters contained in the specfem par_file directory. However, when these do need to be modified, it can be done through the `par` attribute of `Control2d`. `par` is a hierarchy of pydantic models which matches the structure of the par file. They also validate inputs to help prevent the creation of an invalid par_file. For example:

```
# dt must be a float so this doesn't work
control.par.dt = "bob"
# but strings that can be converted to floats do work
control.par.dt = "0.1"
```

The following code switches from P/SV mode to SH:

```
control.par.p_sv = False
```

This snippet tells control to not output postscript files; they are too big!

```
control.par.visualizations.postscript.output_postscript_snapshot = False
```

Once all changes have been made, the state is written back to disk using the `Control2d.write` method. This method writes all the necessary Specfem2D files to capture the changed state of the simulation. By default Specster is careful not so squash existing files, so setting the `overwrite` parameter to `True` is needed.

```
# overwrite is true to replace existing par file.
control.write(overwrite=True)
```

Of course, sometimes it may be more convenient to edit the par_file directly, then instantiate a new `Control2d` object to update the state.

6 Updating the models

6.1 Parfile updates

The material models of the simulation can be updated directly like so:

```
import specster as sp

control = sp.Control2d().copy()
# double the velocities in the inclusion material
control.par.material_models.models[3].Vp *= 2
control.par.material_models.models[3].Vs *= 2
control.write(overwrite=True)
fig, ax = control.plot_geometry(kernel=('vp', 'vs'))
```

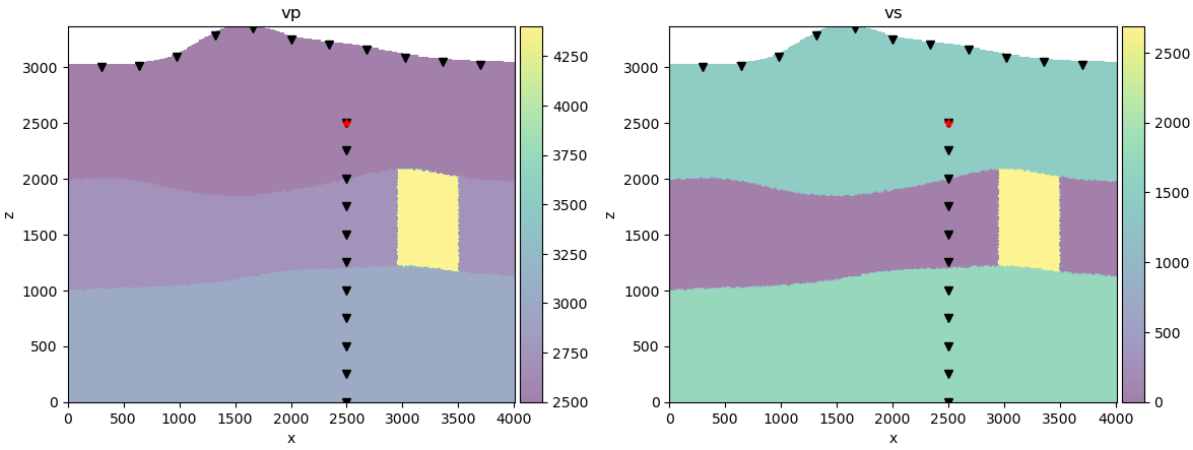


Figure 5: Model update through par file

6.2 Custom models

Also, the ability to enter custom models (which may be difficult to describe in a small number of materials in the par file) is an important feature. In the following example a circular low P

velocity zone is created in the center of the model and the s wave velocity is left unchanged.

```
import numpy as np
import specster as sp

control = sp.Control2d().copy()
# get current material properties as a dataframe.
# index is x/z coordinates and columns are material values.
df = control.get_material_model_df()
# get coords of each gll point and location of center
coords = df.reset_index()[['x', 'z']].values
center = np.mean(coords, axis=0)
# find points which are within 500m of center
distances = np.linalg.norm(coords - center, axis=1)
in_distance = np.abs(distances) < 500
# update P/S velocities by making points in distance 50% slower
new_vp = df['vp'].values
new_vp[in_distance] *= 0.5
df['vp'] = new_vp
# now set model and plot
control.set_material_model_df(df)
control.plot_geometry(kernel=('vp', 'vs'))
```

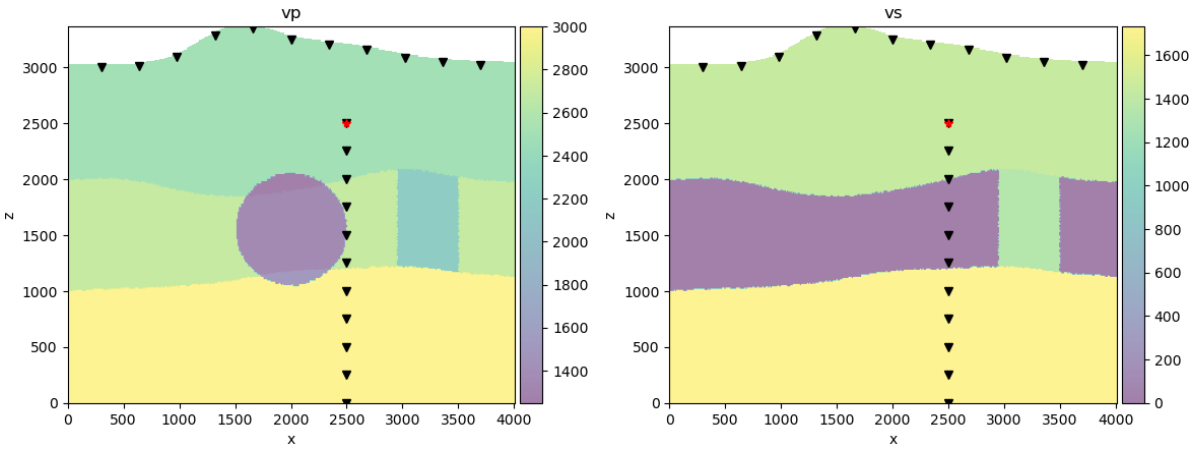


Figure 6: Model update through material dataframe

7 Full Waveform Inversion

Specster provides a few different abstraction levels for performing full waveform inversion. Using `Control2d` directly is the lowest level of abstraction, but it is helpful to understand before covering the higher levels.

7.1 Single event-station kernel calculation

This section shows how to calculate and visualize a single event kernel and a single station with Specster.

The first step is to create the true model. A homogeneous model makes it easy to assess the quality of the results.

First, setup the control structure and run forward for the "true" model.

```
import specster as sp

control_true = sp.load_2d_example('homogeneous_2d').copy("outputs/kernel_one_one_true")

# remove all but one station
control_true.stations = control_true.stations[:1]
control_true.prepare_fwi_forward()
control_true.run()
```

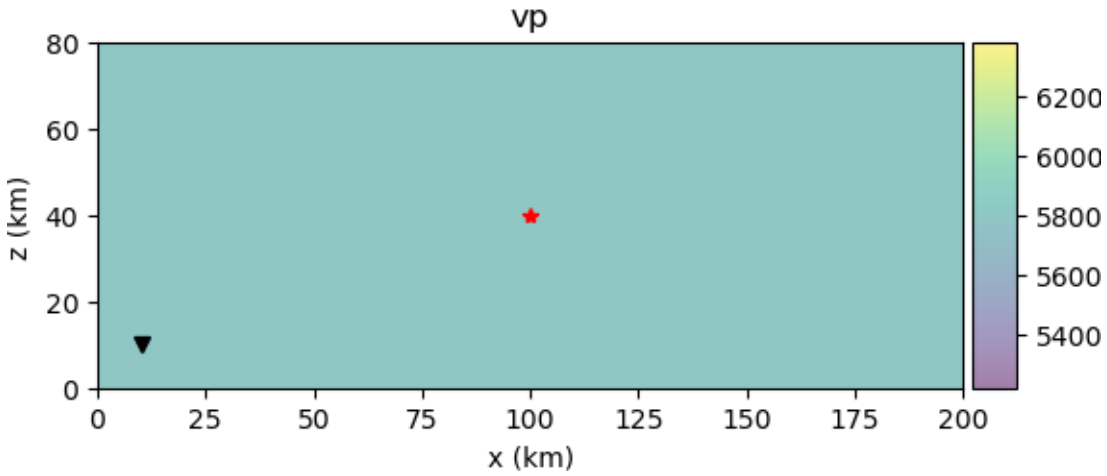


Figure 7: Homogenous model

Next, the initial model is created by modifying the velocity of the true model. Again, this is just a homogeneous background velocity model.

```
control_initial = control_true.copy("outputs/kernel_one_one_true")
# reduce s velocity by 2%.
control_initial.par.material_models.models[0].vs *= 0.98
control_initial.prepare_fwi_forward()
control_initial.run()
```

Next, the misfit classes from the `fwi` module are used to calculate the misfit and create adjoint source traces. These examples simply use the waveform misfit but travel time and amplitude misfit functions are also available.

After this, the adjoints are saved to disk and the initial control run in fwi_adjoint mode.

```
from specster.fwi.misfit import WaveformMisFit

# load seismograms as Streams
st_true = control_true.output.get_waveforms()
st_initial = control_initial.output.get_waveforms()

# calculate adjoint source
misfitter = WaveformMisFit()
adjoint_source = misfitter.get_adjoint_sources(st_true, st_initial)
adjoint_source.plot()

# write adjoints to initial control and run fwi
control_initial.write_adjoint_sources(adjoint_source)
control_initial.prepare_fwi_adjoint()
control_initial.run()
```

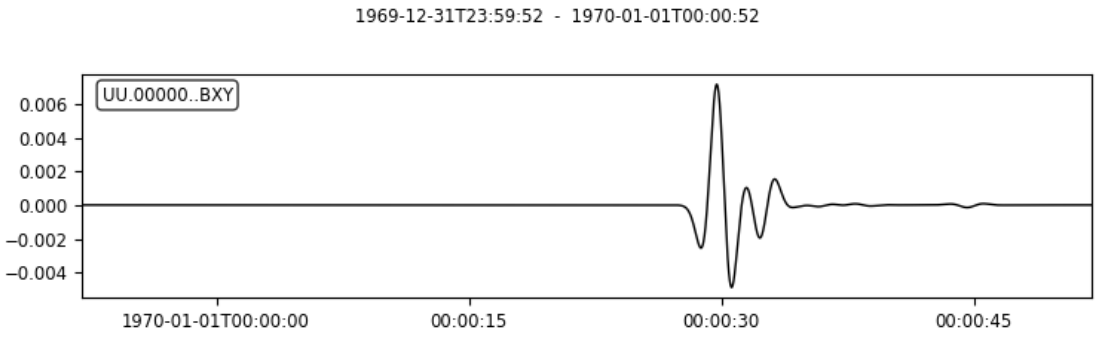


Figure 8: Adjoint source example

`plot_kernels` creates helpful visualizations of the kernels.

```
output = control_initial.output
output.plot_kernel(kernel='beta')
```

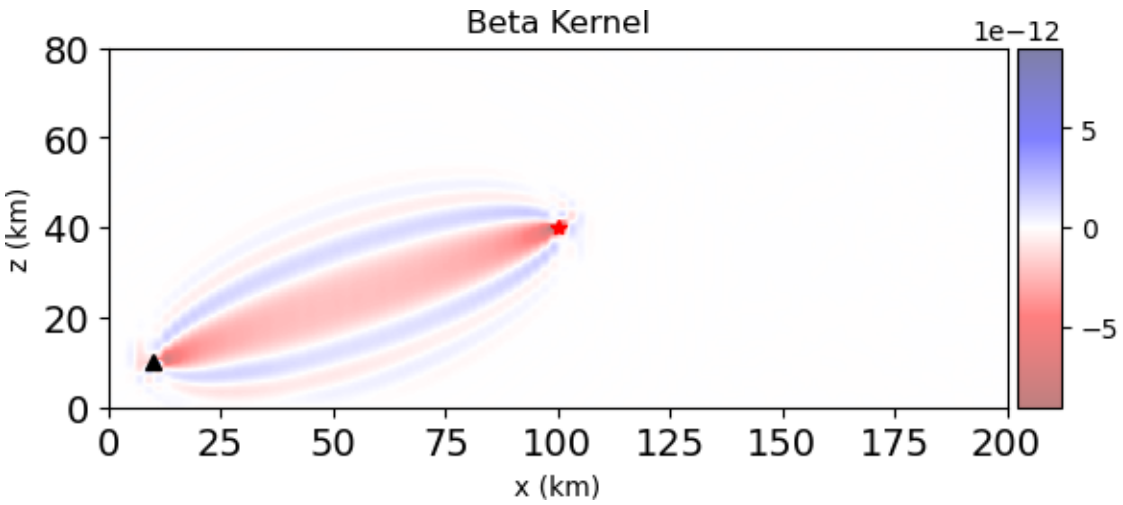


Figure 9: Single station/source kernel

This exercise could, of course, be repeated for multiple stations, but the workflow is identical.

Before moving onto multiple stations/events, it's worth taking a closer look at the misfit class and how to extend it. Here is the definition of the waveform misfit:

```
from scipy.integrate import simps

from specster.fwi.misfit import BaseMisfit


class WaveformMisfit(BaseMisfit):
    """
    Manager to calculate misfit and ajoints for waveform misfit.
    """

    def calc_misfit(self, tr_obs, tr_synth):
        """Calculate the misfit between streams."""
        dx = tr_obs.stats.delta
        misfit = simps((tr_synth.data - tr_obs.data) ** 2, dx=dx)
        return misfit
```

```
    def calc_adjoint(self, tr_obs, tr_synth):
        """Return the adjoint source trace."""
        new = tr_obs.copy()
        new.data = tr_synth.data - tr_obs.data
        return new
```

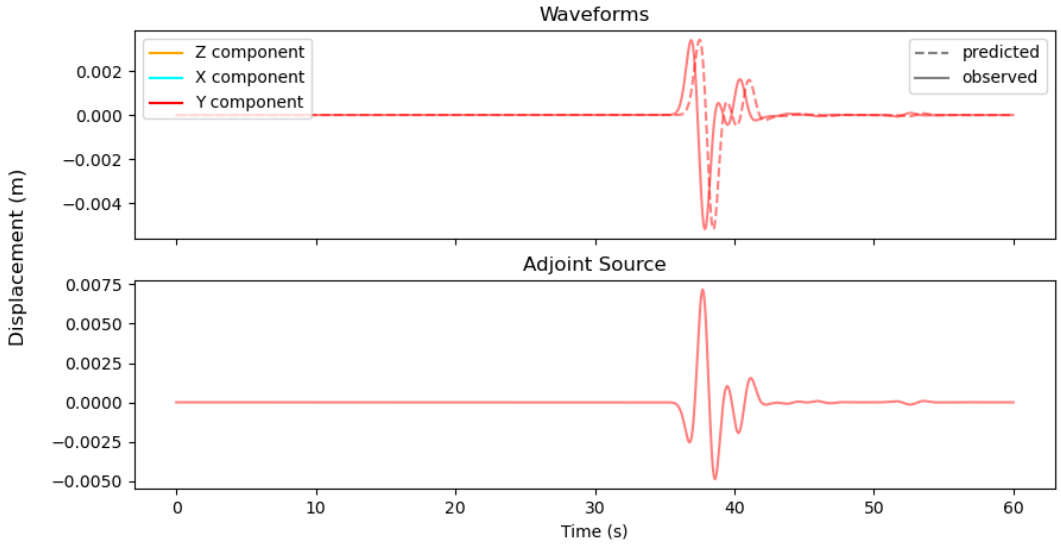
It is only required to implement two methods `clac_misfit` and `calc_adjoint`. These both take two `obspy.Trace` objects, observed and synthetic data, and return either a `float` (for the misfit) or a trace with the adjoint source.

Preprocessing can be modified in a few ways. By default, the traces are detrended and tapered. The taper percentage is controlled by the `taper_precentage` attribute which is 0.05 (5%) by default, meaning the first and last 5% of the trace has a cosine taper applied. `normalize_traces` controls whether each trace is normalized to its maximum amplitude during preprocessing. For even more control, the `preprocess_traces` method can be implemented in which case the subclass will be in complete control of preprocessing.

```
from specster.fwi.misfit import WaveformMisfit

class NewMisfit(WaveformMisfit):
    """
    A subclass of waveform misfit which does special preprocessing on traces.
    """

    def preprocess_trace(self, tr):
        """Custom preprocessing"""
        ...
```

When creating or debugging misfit functions, it can be extremely helpful to visualize the original traces and adjoint source. Here is an example:

```
st_true = control_true.output.get_waveforms()
st_initial = control_initial.output.get_waveforms()

misfit = WaveformMisfit()
fig, (ax1, ax2) = misfit.plot(st_true, st_initial)
```



Waveforms can also be windowed by creating a windowing dataframe, which will `Misfit` uses to know which parts of the waveform to use. The `Misfit` class then handles re-assembling the traces to create proper adjoint sources.

7.2 Multiple Events and Multiple Stations

A nearly homogeneous velocity model with an inclusion in the center and very good coverage of events and stations is useful for learning Specster's FWI workflow. The "inclusion_2d" dataset was created for this purpose.

```
import specster as sp

control_true = sp.load_2d_example("inclusion_2d").copy("out/inclusion_2d_path")
control_true.plot_geometry(kernel='vs')
```

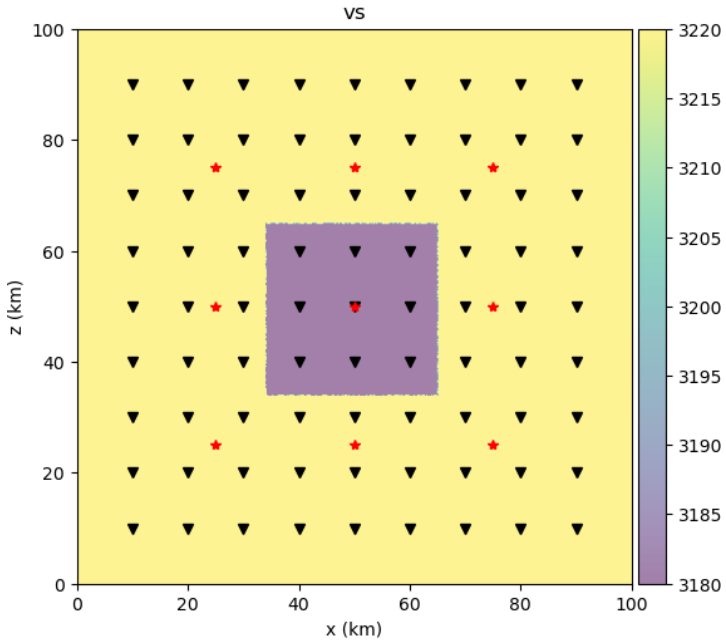


Figure 10: Inclusion model geometry

Next, each source is split into a separate run (Specster doesn't do source encoding). Fortunately, all of these runs are done in parallel using python's multiprocessing functionality.

```
control_true.run_each_source()
```

This will create a directory called "EACH_SOURCE" and populate it with copies of the specfem DATA directories although each source file has been modified to only include a single source. The source-specific directories are simply numbered sequentially starting with "000000".

Now that the "True" waveforms are created for each model, an initial mode should be created. For this, a homogenous background model makes sense.

```
control_initial = control_true.copy("out/inclusion_2d_initial")

# modify the second material property to match first. Homogenizes the model.
models = control_initial.par.material_models.models
models[1].Vs = models[0].Vs
models[1].Vp = models[0].Vp
models[1].rho = models[0].rho

control_initial.write(overwrite=True)
```

```
fig, *_ = control_initial.plot_geometry(kernel='vs')
```

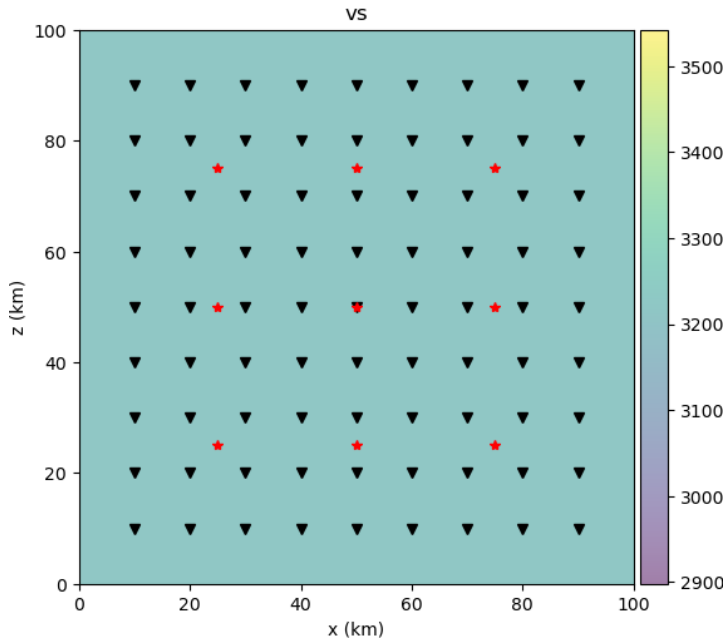


Figure 11: Inclusion starting model

It is tedious to work with multiple `Control2d` objects for each source, so Specster provides a higher level class for conducting the FWI workflow called `Inverter`.

```
import specster as sp
from specster.fwi.misfit import WaveformMisfit

inverter = sp.Inverter(
    # Specifies where true data are found
    observed_data_path=control_true.each_source_path,
    # The initial control is used to setup the inversion
    control=control_initial,
    # A "true" control object is needed to compare model misfit
    true_control=control_true,
    # The working_path optionally specifies where the inverter does its work
    working_path="outputs/fwi_inclusion_2d",
    # specifies the misfit function.
    misfit=WaveformMisfit(),
)
```

The `Inverter.run_iteration` method performs a single iteration like so:

```
inverter.run_iteration()
```

Similar to the `Misfit` class, `Inverter` is mildly extensible through initialization parameters (see its documentation) and completely extensible through inheritance. The basic (default) workflow conducted with `run_iteration` does the following:

1. Ensure each source has its own specfem files
2. Run the forward problem for each source
3. Calculate misfit and adjoint sources for each source then save adjoints
4. Run all sources in adjoint mode
5. Sum the gradients after preconditioning with the approximated Hessian provided by Specfem2D and applying a median filter to the station locations in the gradient (can be disabled based on various input parameters)
6. Conduct a line search to find the optimal step size, with a maximum update of ~2% (by default). Specster uses a simple golden section search
7. Apply the scaled gradient to the velocity model to calculate the starting model for the next iteration (steepest descent) and update all model files

For each iteration, the inversion state is saved so it can be restarted from any point.

To run 3 iterations:

```
for _ in range(3):
    inverter.run_iteration()
```

A list of results are stored for each iteration

```
print(inverter.results)
```

Stopping criteria can be manually specified and the loop exited when those are met.

An existing inverter can be loaded via the `load_inverter` class method:

```
import specster as sp

inverter = sp.Inverter.load_inverter("inverter/working/directory")
```

If the simplistic line search algorithm can't find a stepsize to reduce the misfit, then a `FailedLineSearch` Error will be raised. Once the optimal step size is found, the model is updated and results for the current iteration are saved in the subdirectory "ITERATIONS".

Running the previous example for 15 iterations yields the following model (top) compared to the true model (bottom). Each iteration took about 10 minutes to run.

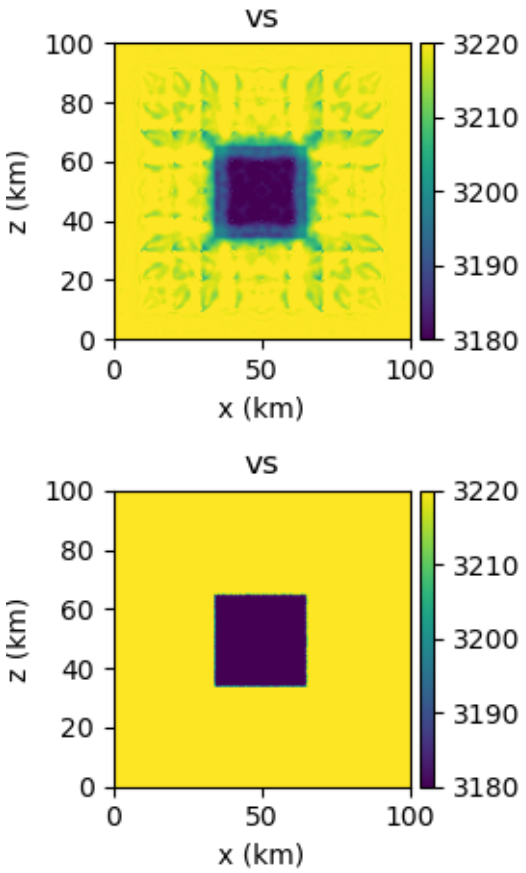


Figure 12: Final vs true model

Data stored in the inverter can be used to plot the convergence as a function of iteration for data misfit, and, if the true model is known, model misfit (L2 norm of the difference between VS models).

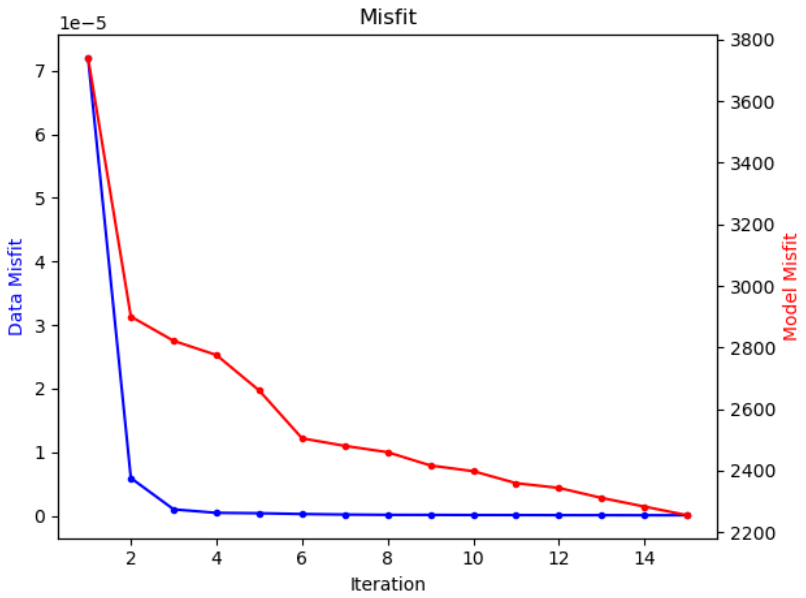


Figure 13: Model convergence

A few things to note about this example. First, although the outline of the inclusion is visible, there is still quite a bit of smearing. Also note that the station imprints are not completely removed as there are dashes of lower velocity outside the inclusion. A few options that might improve the results include: First, rather than waveform misfit, travel_time might be more robust. Second, there was no smoothing applied here, but if the `smoothing_sigma` parameter is defined when creating an `Inverter` instance, it may help remove the artifacts outside the inclusion.

8 Conclusions

Creating Specster helped me to better understand a variety of topics including Specfem2d and the general (simplest) FWI workflow. Although it still lacks some major features, like L-BFGS optimization for model updates and various misfit functions, it may be a useful tool for future FWI research.

9 Appendix A: Imaging an Acoustic Reflector

This appendix provides an example of using specster to image an acoustic reflector, similar to an reverse time migration (RTM) workflow common in exploration seismology. In this case,

only one iteration is conducted and the kernel is used as the image to illuminate the location an irregularly shaped interface between two different materials.

9.1 Setup true and initial models

First, the true and initial model are created.

```
# true model creation
from pathlib import Path

import specster as sp

true_path = Path("true_model")

true_cont = sp.load_2d_example("acoustic_reflector").copy(true_path)
true_cont.par.visualizations.postscript.output_postscript_snapshot = False
true_cont.par.nstep = 2200
true_cont.write(overwrite=True)
true_cont.run_each_source()
true_cont.prepare_fwi_forward().run()

true_cont.plot_geometry(kernel=('vp',))
```

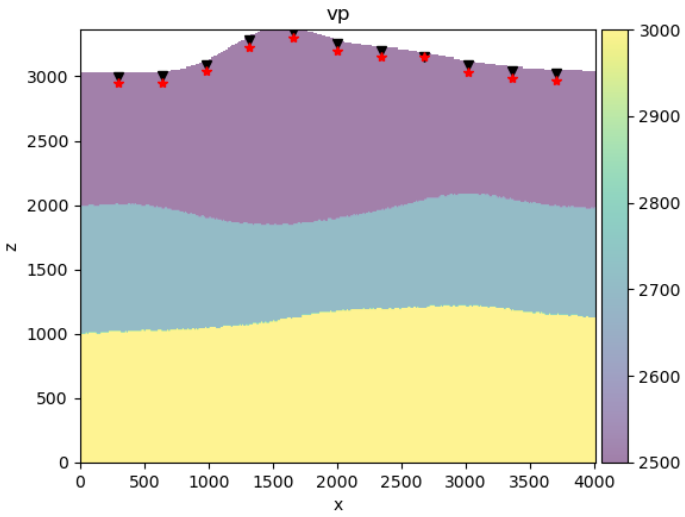


Figure 14: Acoustic Reflector Model: True

```
# initial model creation
from pathlib import Path

import specster as sp

initial_path = Path("initial_model")

init_cont = sp.load_2d_example("acoustic_reflector").copy(initial_path)
init_cont.par.visualizations.postscript.output_postscript_snapshot = False
init_cont.par.nstep = 2200
init_cont.write(overwrite=True)
init_cont.run_each_source()
init_cont.prepare_fwi_forward().run()

init_cont.plot_geometry(kernel=('vp',))
```

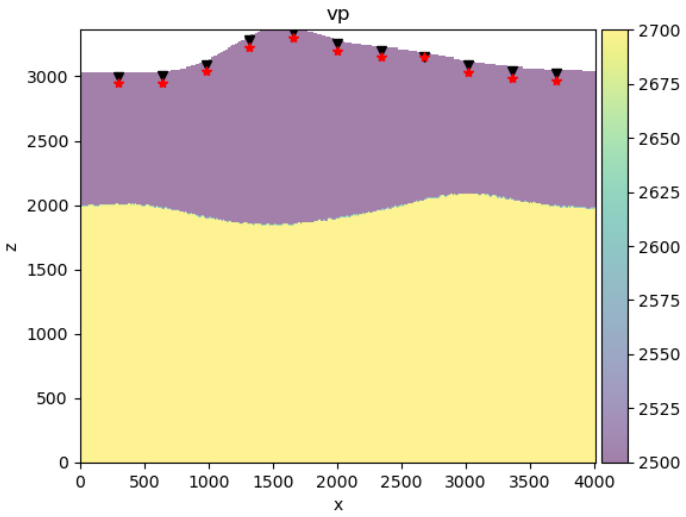


Figure 15: Acoustic Reflector Model: Initial

Then the Inverter is created and one iteration run without preconditioning or updating the model.

```
import specster as sp
from specster.fwi.misfit import WaveformMisfit

inverter = sp.Inverter(
    # Specifies where true data are found
    observed_data_path=true_cont.each_source_path,
    # The initial control is used to setup the inversion
    control=control_initial,
    # A "true" control object is needed to compare model misfit
    true_control=init_cont,
    # The working_path optionally specifies where the inverter does its work
    working_path="acoustic_reflector_fwi_path",
    hessian_preconditioning=False,
    misfit=WaveformMisfit(),
    kernels=("c",),
)
```

```
inverter.run_iteration(no_update=True)
```

The resulting c kernel shows the material interface is illuminated in the center of the model. However, there isn't sufficient data coverage to illuminate the reflector near the edges of the model. An imprint of the source frequency is also visible as the thickness of the interface line. However, the image still provides useful information about the surface of the third layer and is certainly not bad for only 11 stations and sources. It would be impossible to image the third layer without receivers at depth or additional reflections which send seismic energy back to the surface.

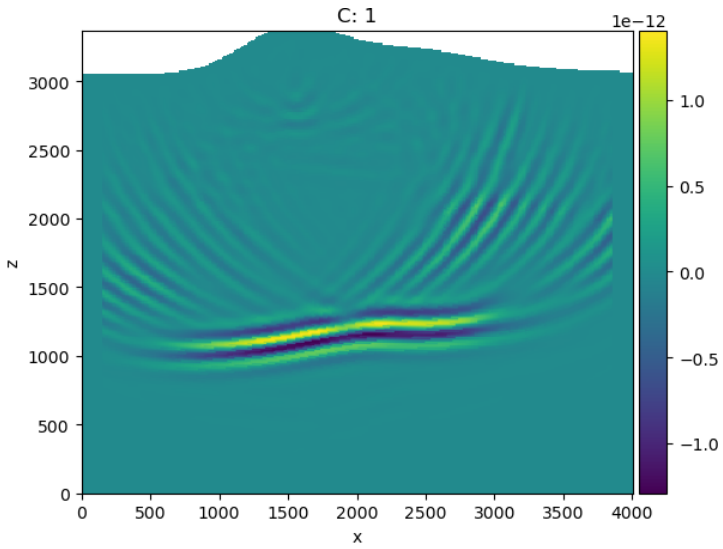


Figure 16: Acoustic Reflector Model: Image