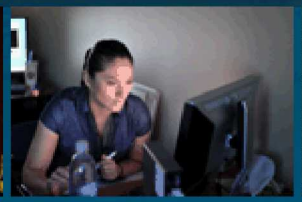
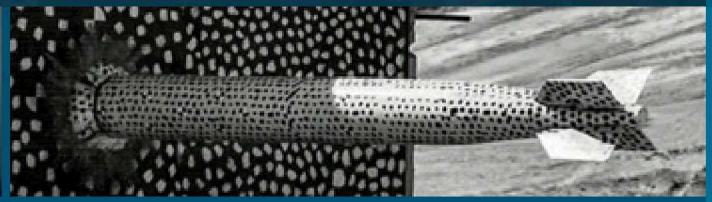
SAND2020-10234R

Waveform Modeling and Simulation for Crustal Phases

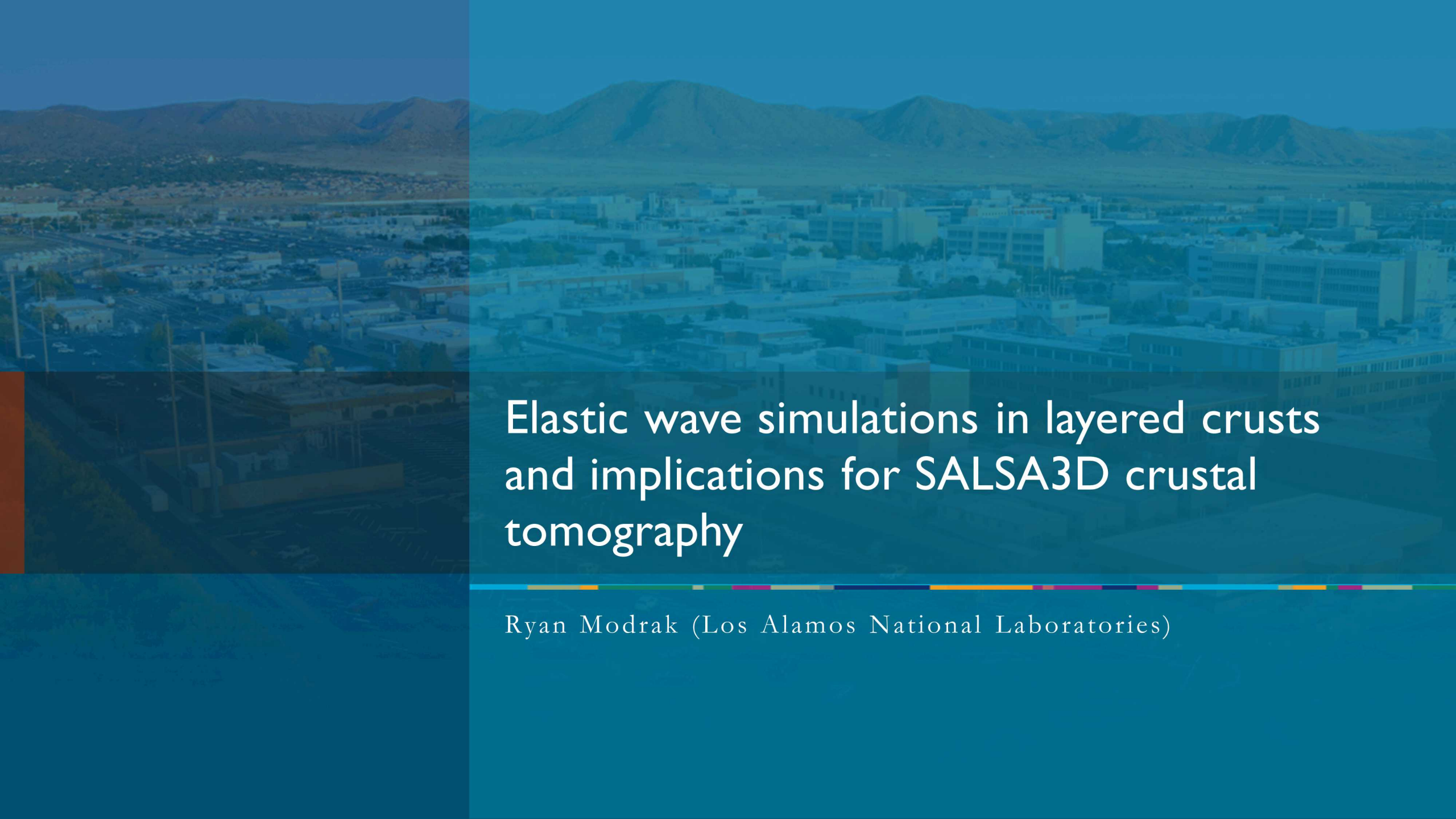




PRESENTED BY

Ryan Modrak (Los Alamos National Laboratories); Nathan Downey (Sandia National Laboratories)

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.



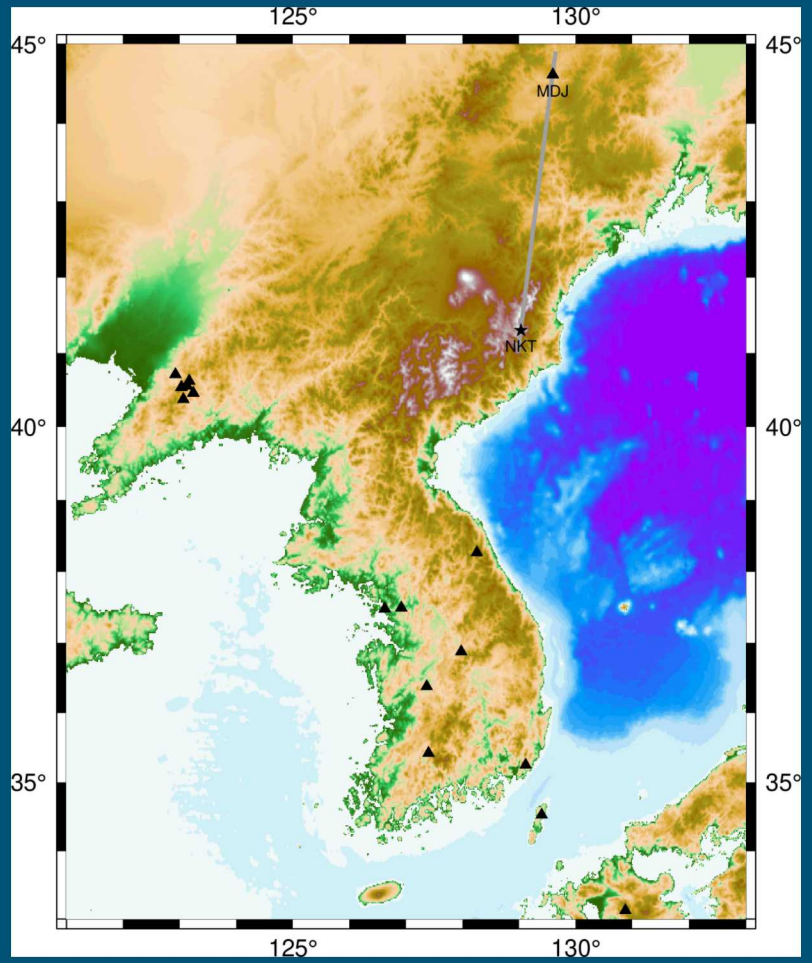
Elastic wave simulations in layered crusts and implications for SALSA3D crustal tomography

Ryan Modrak (Los Alamos National Laboratories)

Goals

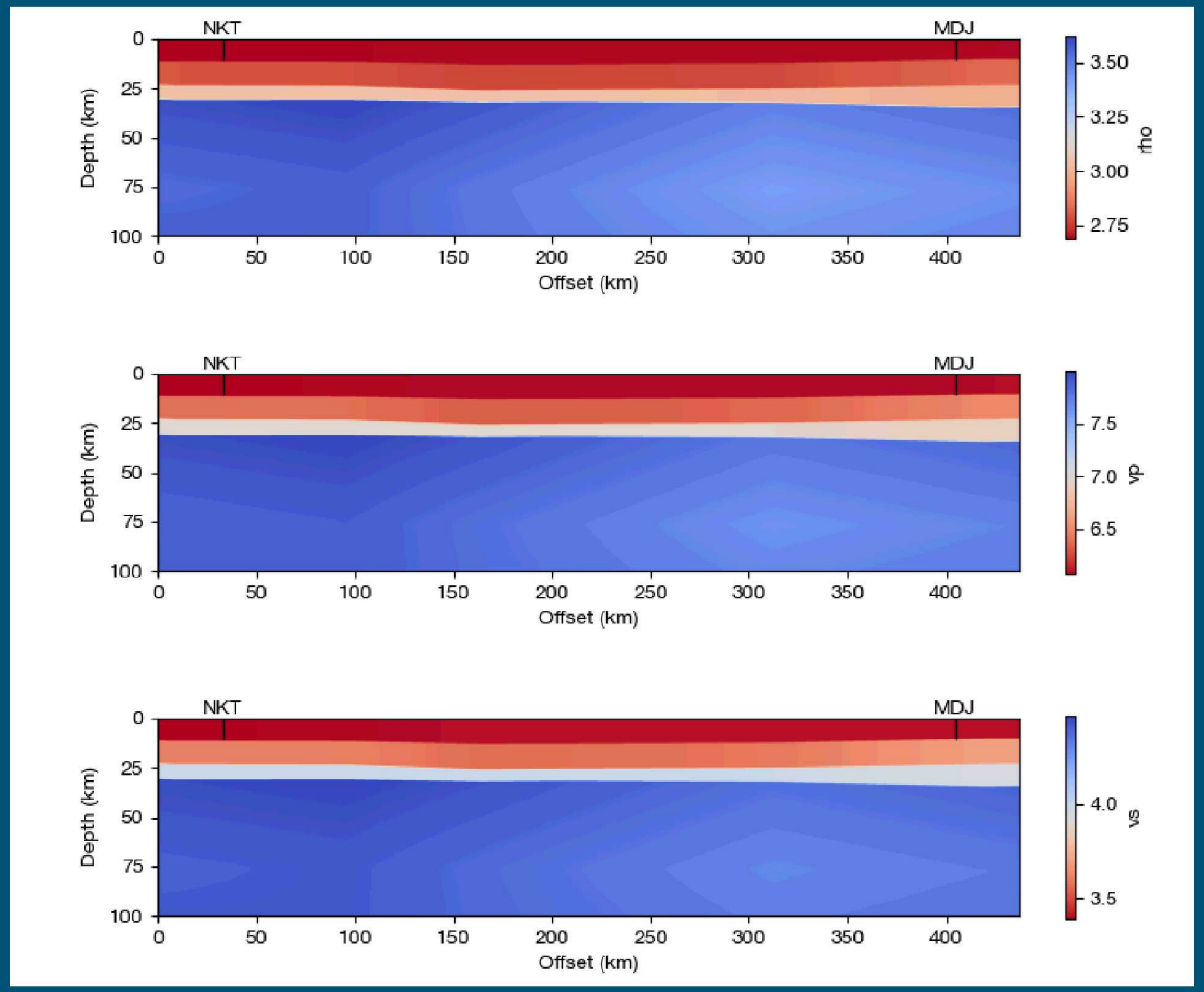
- Numerically simulate wave propagation in layered crustal models with laterally-varying structure
- Investigate seismic phases Pg and Pn; help understand the variability and complexity of these phases in recorded seismic data
- Use wavefield movies and "wavepaths" to suggest improvements for SALSA3D crustal tomography

Cross Section Used for Wavefield Simulations



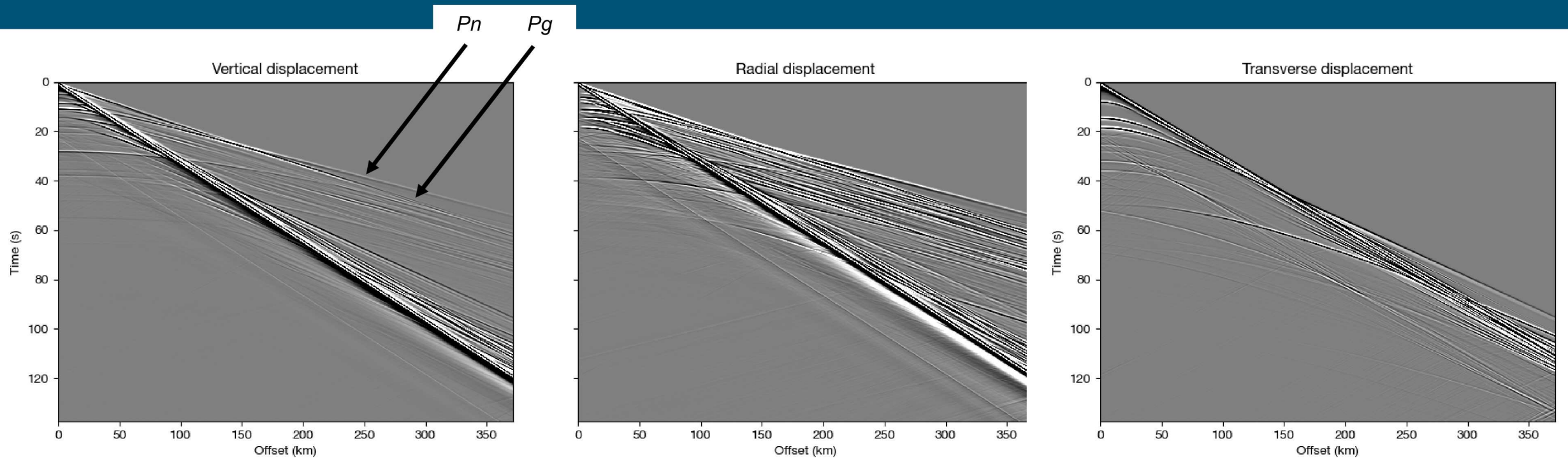
For numerical wave simulations, we chose a ~400 km cross-section from the North Korea Test Site to station MDJ. Crustal structure along this path has been well studied, and unlike many nearby stations, MDJ corresponds to an entirely continental path.

SALSA3D Velocity Structure Used for Wavefield Simulations



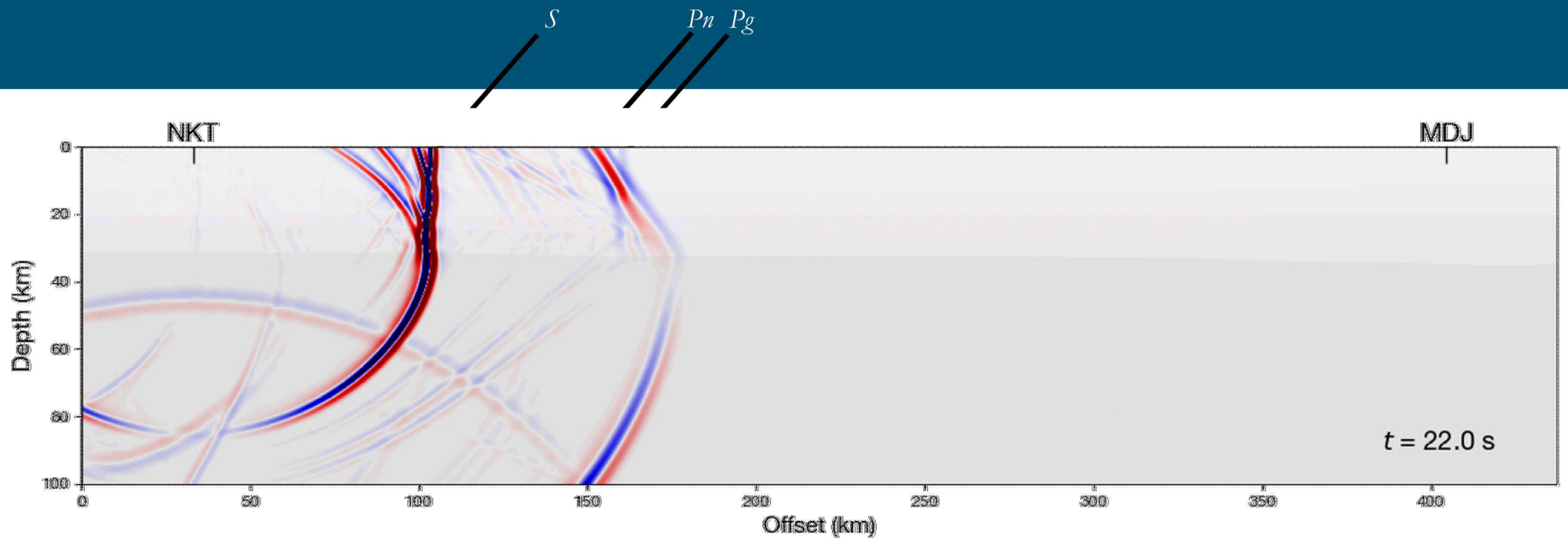
For a 2D isotropic elastic velocity model, we interpolated *P*-wavespeeds (middle) from SALSA3D. To obtain density (top) and *S*-wavespeeds (bottom), we used the depth-dependent ratios defined by AK135F.

Synthetic Record Sections



Using a 2D spectral-element solver and an impulsive source-time function with 1 Hz dominant frequency, we computed the above displacement record sections. Pg is the first arriving phase at offsets less than 150 km, being overtaken by Pn at greater offsets.

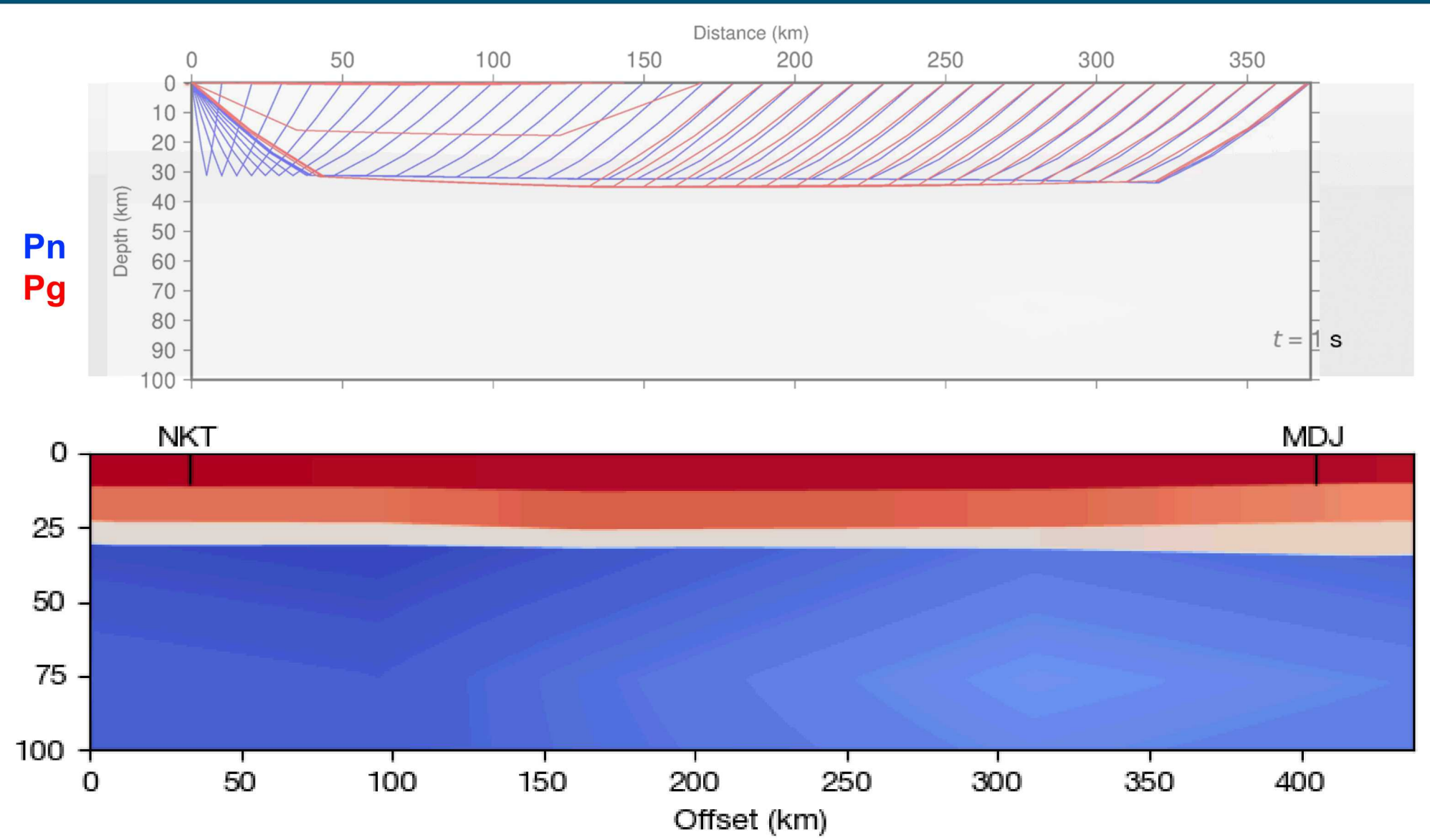
Wavefield Snapshot (Direct Body Phases Only)



P-SV wavefield snapshot just before *Pn* overtakes *Pg*. For easier visualization of direct body wave phases, this simulation was carried out with an absorbing free surface.

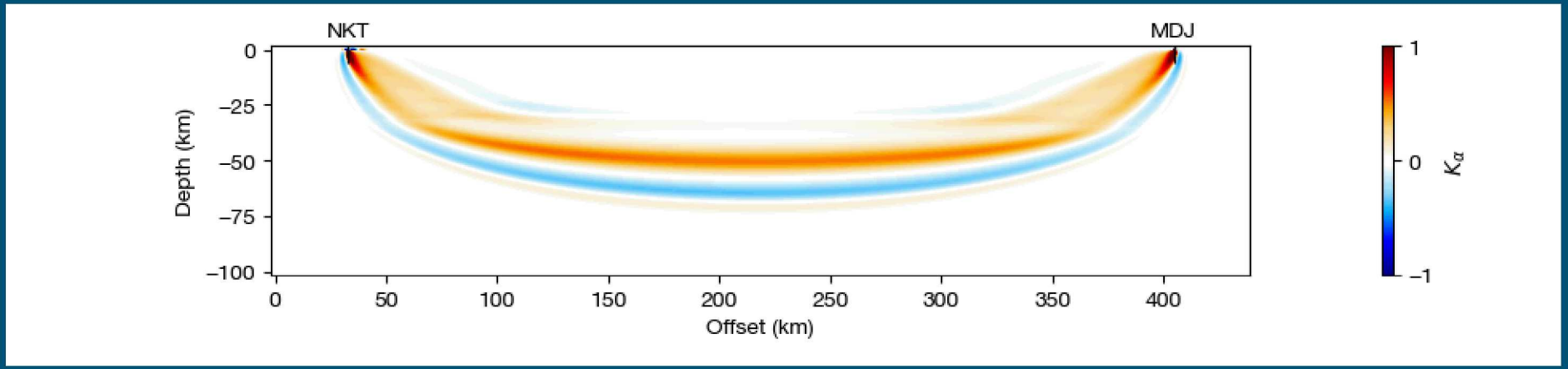
Despite spatial overlap, the two phases can be distinguished by incidence angle: *Pg* travels horizontally through the crust and has ~0° incidence, while *Pn* travels upward from the Moho and has >0° incidence.

Wavefield Movie Superimposed on Bender Ray Paths

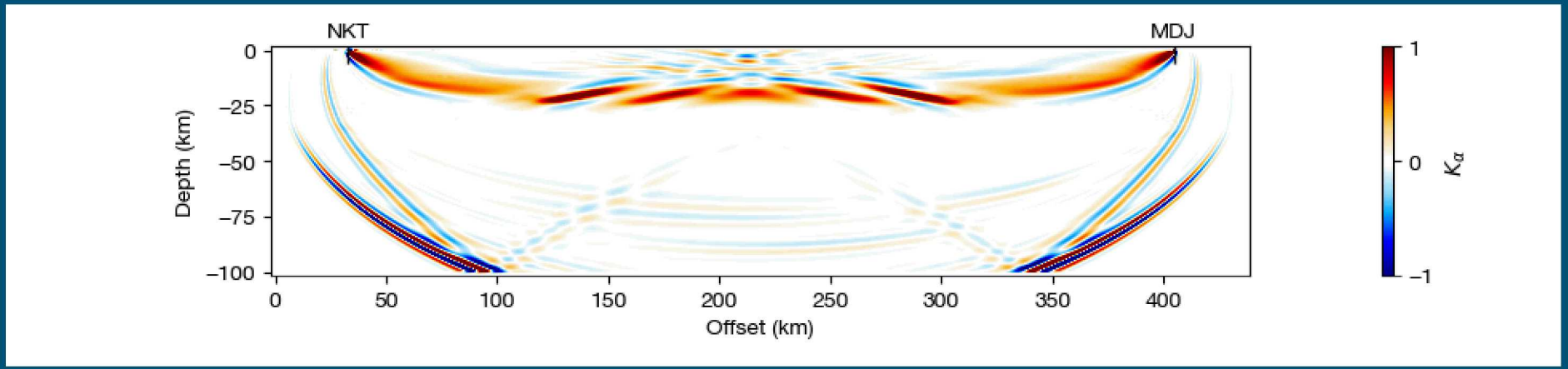


Note: Movie and raypaths may not display correctly from Sharepoint/ browser

Pn traveltime kernel



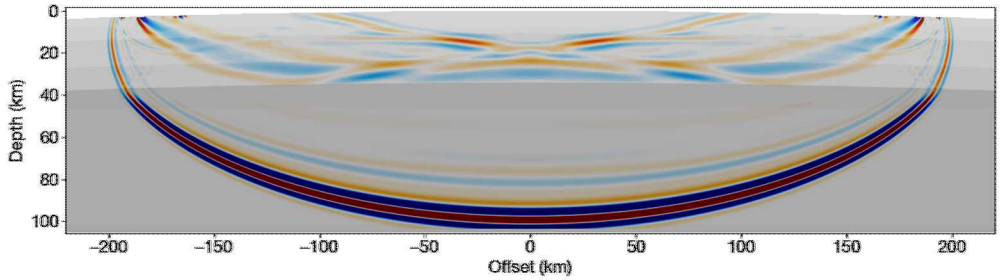
Pg traveltime kernel



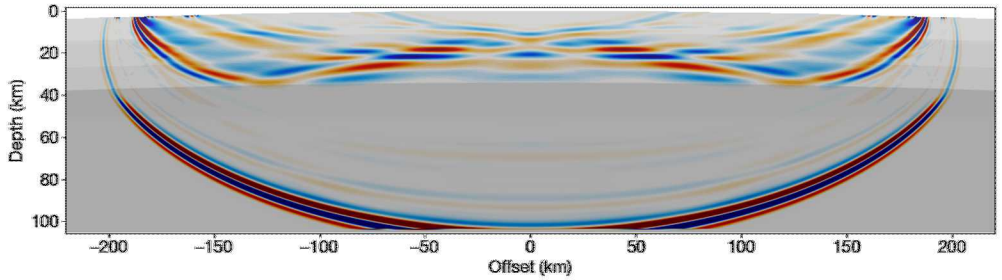
Using a cross-correlation traveltime imaging condition relevant to traveltime tomography, we backpropagated *P*-wave phases. The resulting *Pn* kernel, or "wavepath" is relatively simple. The resulting *Pg* kernel is more complicated, revealing discrete waveguide modes. These results suggest that *Pg* can be represented to some extent by ray paths within the deeper crust. Ray modeling, however, ignores diffraction, internal reflection, and mode superposition effects evident in the *Pg* kernel.

Normal modes revealed by sliding a 1-second window along the Pg wavetrain

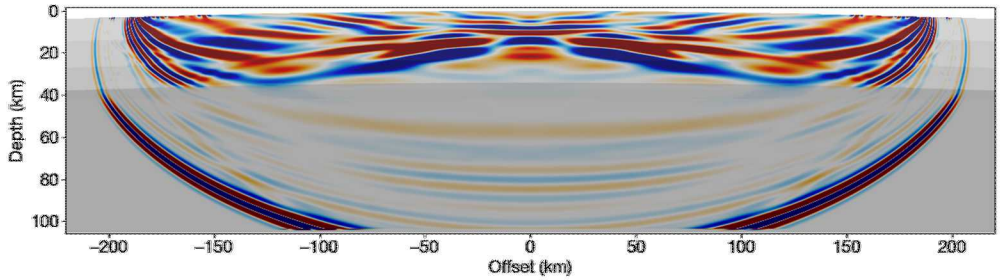
Pg kernel



Pg + 1 sec kernel



Pg + 2 sec kernel



Implications for SALSA3D tomography

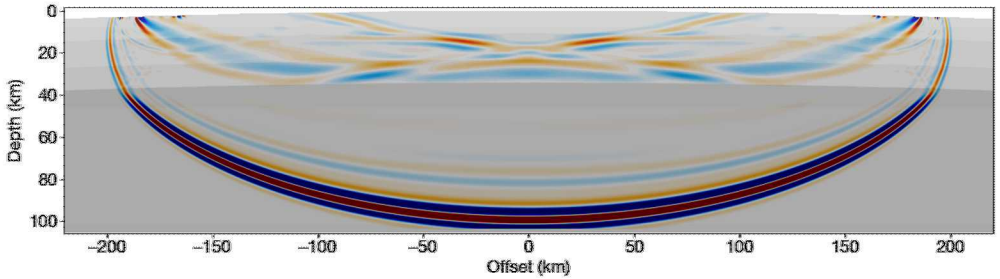
- **Wave-equation sensitivity kernels.** Kernels are important for developing intuition. However, they are expensive and might vary a lot depending on structure.

- **Asymptotic methods for estimating waveguide first arrivals.** Waveguide asymptotics (e.g., https://arxiv.org/abs/1601.00994) are mathematically rigorous, computationally inexpensive and generalizable to different velocity structures.

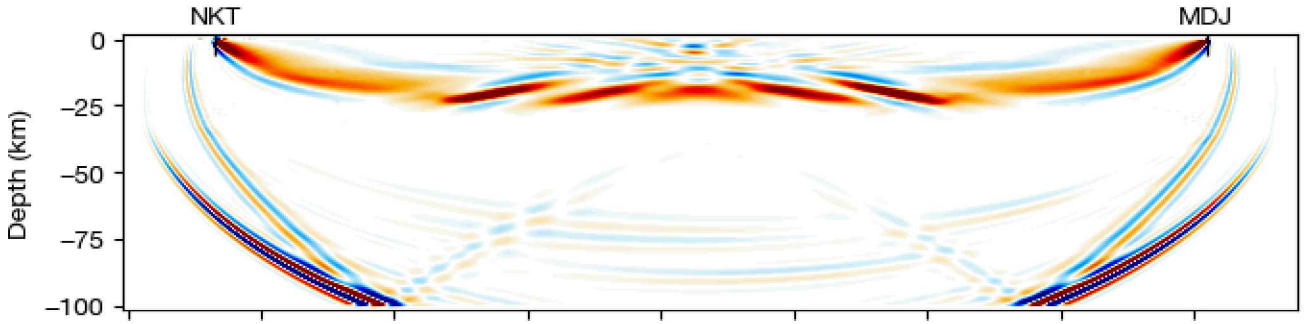
- **Ray-like approximations in quasi-layered models.** In a quasi-layered CRUST1.0 model like SALSA3D, propagation time is closely related to the speed of the fastest layer. A flat path through the fastest layer, combined with an empirical distance-based correction to account for internal reflection and diffraction, might be a useful approach.

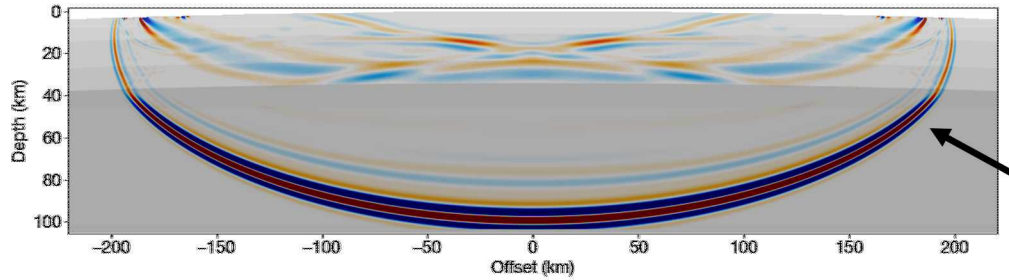
Ongoing research questions

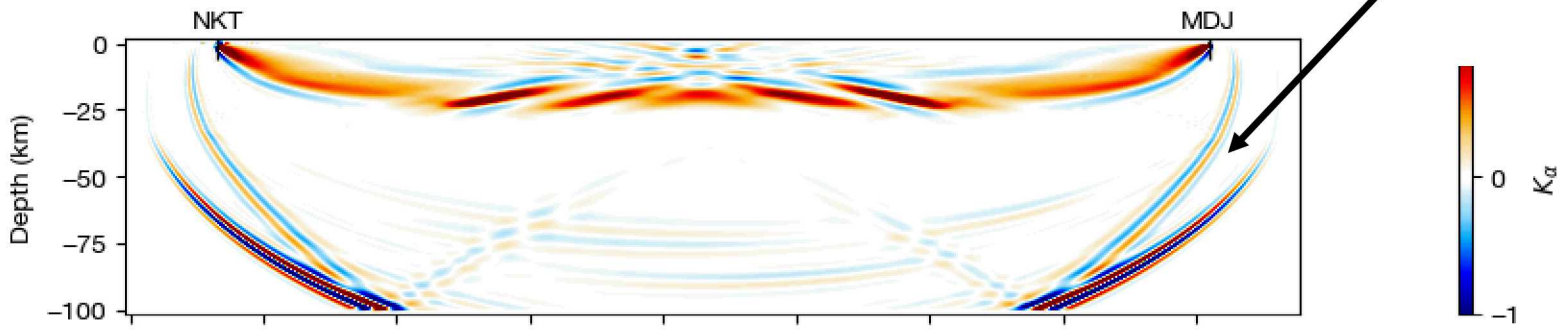
Pg kernel using 1D structure (based on SALSA3D profile beneath station MDJ)



Pg kernel using 3D structure (based on SALSA3D along NKT-MDJ cross-section)



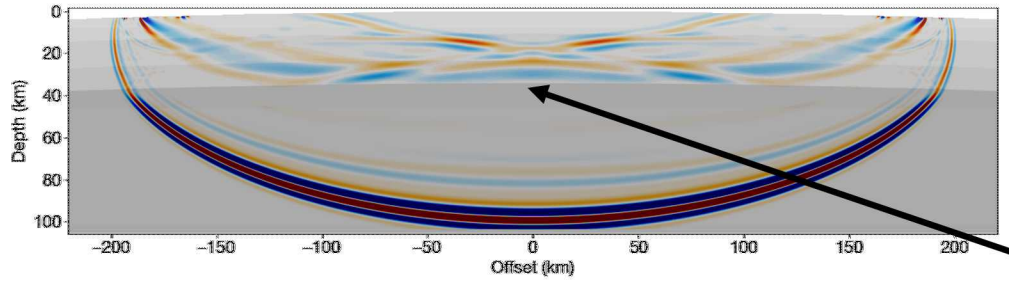


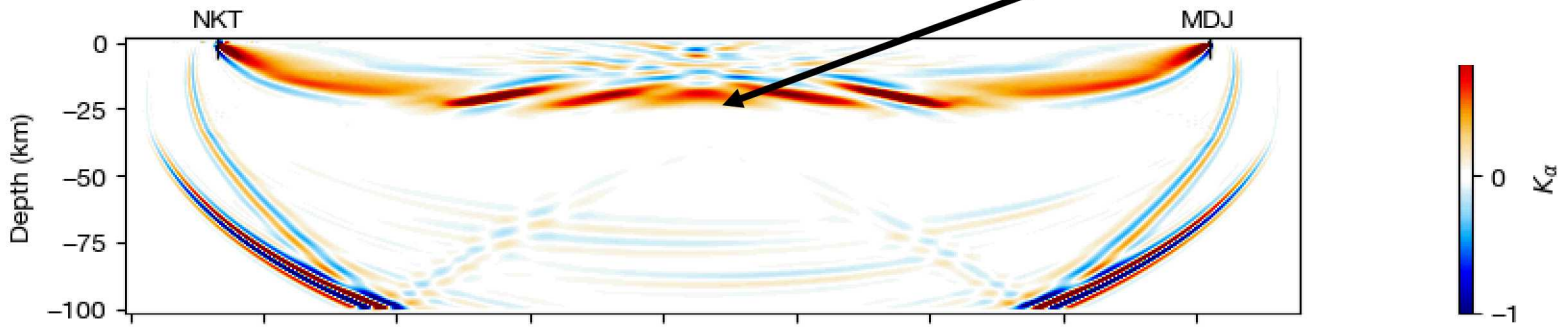


QUESTION 1

Are these pronounced fringes real or spurious?

e.g. do they disappear in 3D?

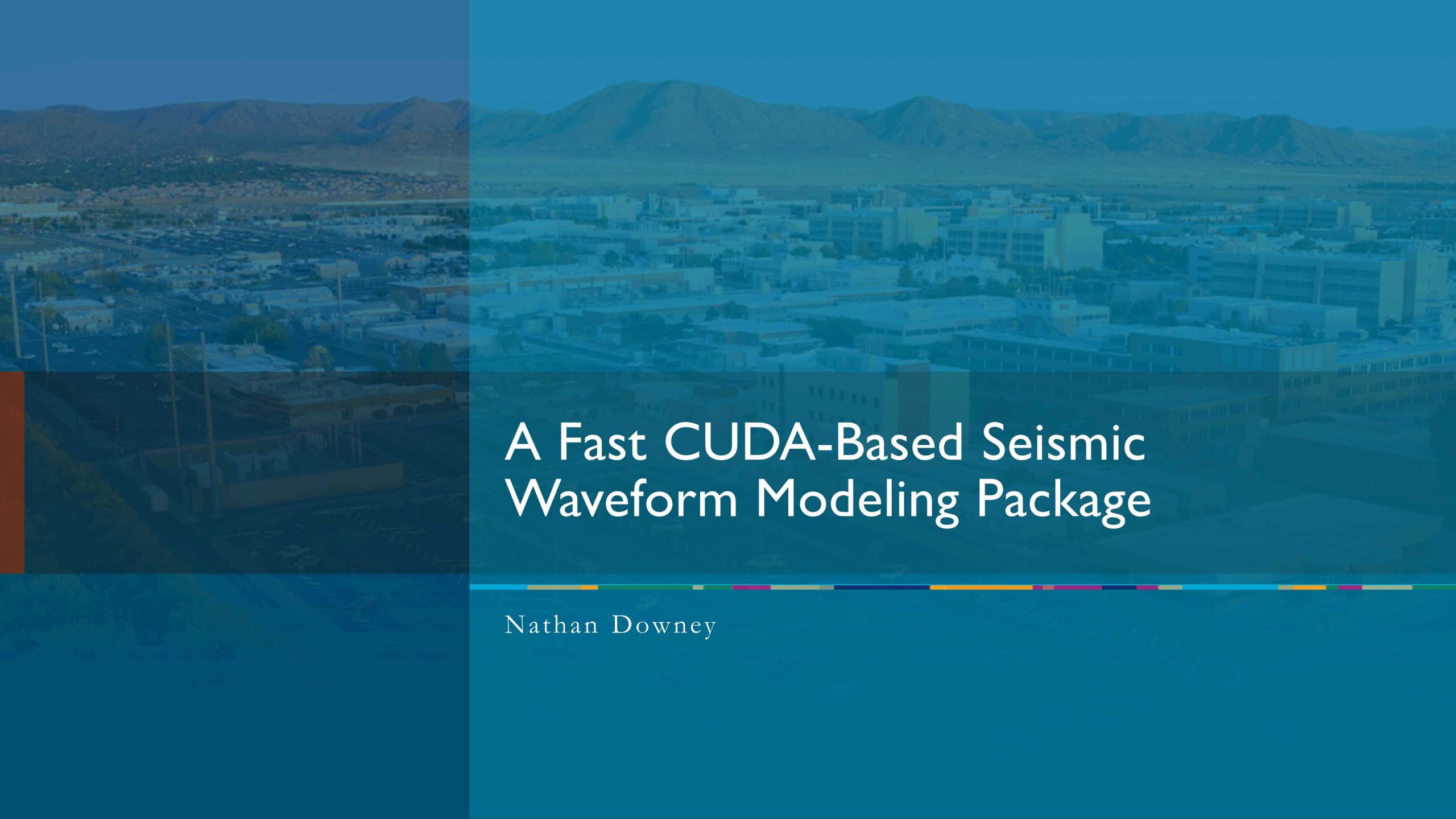




QUESTION 2

Why is the depth dependence different?

e.g. related to layer thickness?



A Fast CUDA-Based Seismic Waveform Modeling Package

Nathan Downey

Motivation

A stated goal of the GNDD Signal Propagation project is to understand the sensitivity of seismic crustal phases to changes in crustal velocity in order to better use these phases in crustal tomography.

- However, it is understood that the current technology for tomographic inversion, based on ray formulations of seismic wave propagation, are insufficient to fully capture the complexities of seismic propagation in Earth's crust.

- Therefore, new tools must be developed to help us achieve the ability to study the complex wavefields in Earth's crust, with crustal waveform modeling being the only known method by which this can be achieved.

Motivation

In this presentation I will outline progress that has been made to date in an effort to modify a community research modeling code ("simulator") into a tool which can be used by GNDD researchers to study the effect of varying crustal velocity on wavefield propagation.

1. I will review simulator technologies and why I chose the particular simulator used.
2. I will outline the structure of the python wrapper that I constructed around the simulator and the tools available to researchers investigating crustal wavefields.
3. I will give an example of a simulation of an event in Central Utah showing the utility of the wrapper and its easy integration with existing python tools.
4. I will discuss future work both in the near-term and long-term

Simulator Technologies

Three simulator technologies were investigated as possibilities for use in crustal modeling studies. The first of these, SpecFEM, is the community standard modeling code, which uses a finite element formulation to compute synthetic seismograms. The second was a Sandia-developed code similar to SpecFEM but with many more advanced features and advanced optimization. The third was a GPU accelerated finite difference code based on an old code base dating back a few decades.

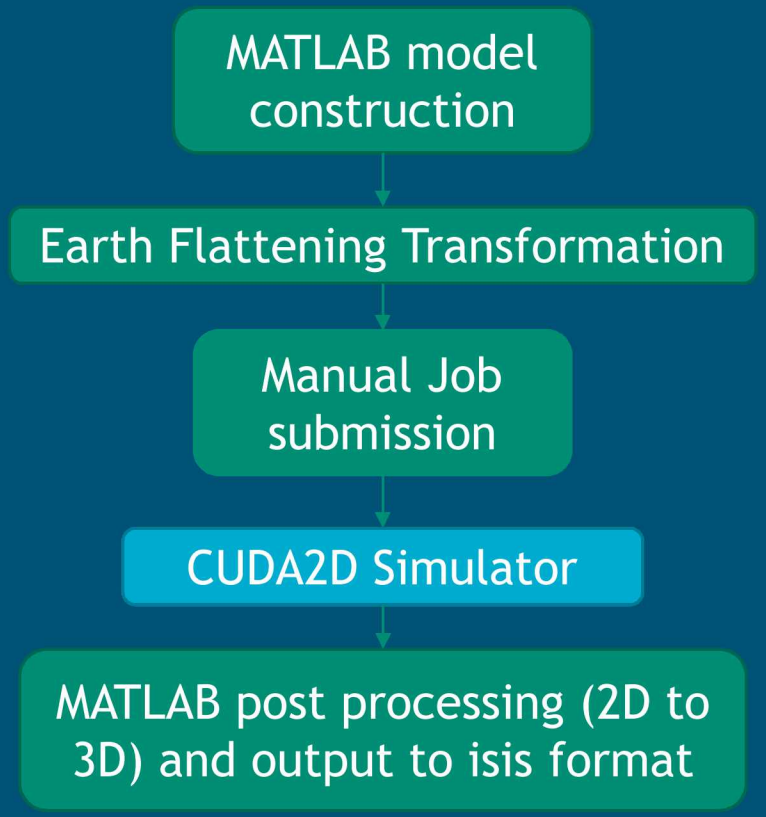
I eventually chose the GPU accelerated code because of the following features:

1. It is extremely fast and can be run on the GPU machines that we already have available in our department. The speed is achieved by: using GPU acceleration, computing in 2d and converting to a 3d response after the simulations, and the use of a finite difference formulation.
2. This code has been used in several recent studies on waveform modeling at both teleseismic and local distances. (Li et al., 2014a; Li et al., 2014b; Chu and Helmberger, 2014). One of these studies (Chu and Helmberger, 2014) examined Lg propagation at regional distances to look at sensitivity of Lg to changes in velocity deep in the crust. This type of study is very much in alignment with what we are trying to accomplish in GNEM.
3. This code is based on an old and well-tested code base that has been used in a variety of finite difference-based modeling studies in the past.

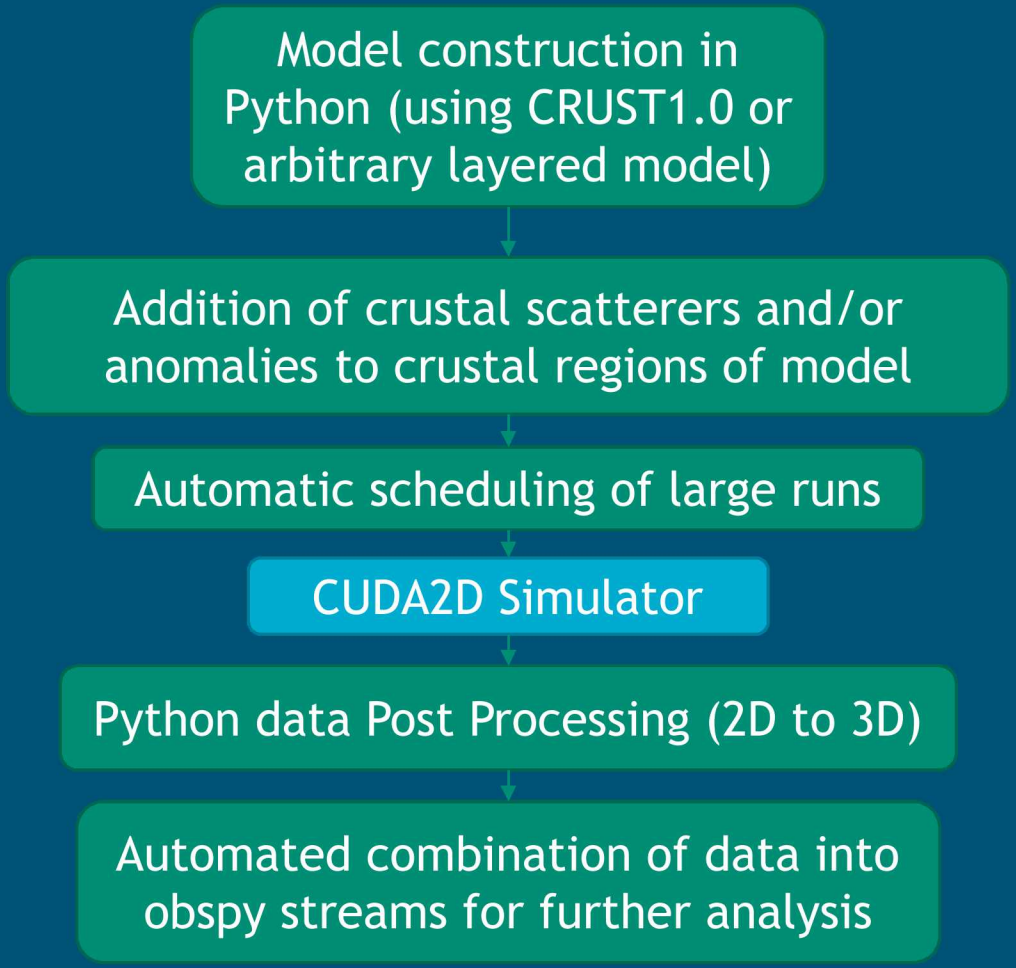
Python Wrapper

The GPU simulator as I received it used MATLAB for model preparation and data post processing. I have built a new wrapper that eliminates the need for MATLAB and provides much more powerful model construction and data processing capabilities.

Old structure of Simulator Software



New structure of Simulator Software



Python Wrapper – Code Structure

Simulation Collection Object
Data Members:
- -Collection of *Simulation Objects*

Methods:
- -**execute()** –dynamically assigns simulations to free GPUs and calls their respective execute() method

Simulation Object
Data Members:
- -*Model Object* (Next Slide)
- -*Source Object*
- -*Receiver Object*

Methods:
- -**execute()** – set up job directory, output parameter files and run simulation jobs on specified GPU
- -various parameter setting methods
- -**dataProcessing()** – post process output data, combine appropriate simulation outputs into obspy streams

Source Object
Data Members:
- -Various source parameters

Methods:
- -**setCoords**() – sets source location
- -**addDislocationComponent**() – adds a double couple component to the source
- -**addVolumetricComponent**() – adds explosive component to source
- -**setSourceTimeFunction**() – sets time dependence of source activity

Receiver Object
Data Members:
- -Coordinates of each sensor in the simulation

Methods:
- -**setCoords**() – sets receiver location

Python Wrapper – Code Structure, Model O

Inherits From

Inherits From

Model Object (abstract base class)
Data Members:
- -Velocity Model and associated parameters

Methods:
- **-setGeometry()** - uses source and receiver coordinates to set the geometry of the simulation
- **-addPerturbations()** - adds random scatterers to model
- **-imageProfile()** - generates preview image of the model
- **-addGaussianAnomaly()** - adds a Gaussian shaped velocity anomaly to the model
- **-getProfile()** (abstract)

Profile Model Object
Data Members:
- -same as Model Object

Methods:
- **-getProfile()** - creates model based on 1D profile
- **-assertMoho()** - removes any scatterers or anomalies below Moho in model

Crust1 Model Object
Data Members:
- -same as Model Object

Methods:
- **-getProfile()** - extracts profile along source-receiver line from CRUST1.0 model
- **-assertMoho()** - removes any scatterers or anomalies below Moho in model, returning velocities to pure CRUST1.0 values in the mantle.

Example Run – Circleville Event in Central Utah

```
import os
import multiprocessing
import CUDA2DSimulator as simulator
import obspy

simulationDirectory = "Simulations"
simCollectionDirectory = "Test_Simulations"

receiverCoords = [(-113.029831, 40.92083, 0.),
                  (-110.739998, 39.473, 0.),
                  (-113.362701, 37.550598, 0.),
                  (-111.750343, 40.692501, 0.),
                  (-110.741798, 37.938, 0.),
                  (-112.775002, 41.779671, 0.),
                  (-111.449951, 40.601952, 0.),
                  (-113.243896, 37.011799, 0.),
                  (-111.633331, 40.015499, 0.),
                  (-112.184303, 38.0415, 0.),
                  (-112.074997, 39.95483, 0.),
                  (-112.120331, 40.6525, 0.),
                  (-112.310997, 37.443901, 0.),
                  (-110.245697, 39.627899, 0.),
                  (-113.854698, 38.533699, 0.),
                  (-109.569504, 40.570801, 0.),
                  (-110.523827, 39.110828, 0.),
                  (-113.087502, 37.595402, 0.),
                  (-112.447197, 38.609501, 0.),
                  (-111.208168, 39.296501, 0.),
                  (-113.125397, 37.356098, 0.)]

source = simulator.Source((-112.329, 38.240, 9.6))
source.addDislocationComponent(strike=336, dip=69, rake=-129)
source.addVolumetricComponent(amplitudeRatio=0.25)
source.setSourceTimeFunction(type='Gaussian', alpha=-40)
```

Imports needed for run. The code for the wrapper is contained in the *CUDA2DSimulator* module. *multiprocessing* is used to set up the runs for each receiver in parallel.

The runfiles and output for each receiver will be in the directory *Simulations/Test_Simulations/Receiver_??/*

Receivers (stations) whose data we are attempting to model

Source parameters, these are common to all the simulations. Here we specify a source whose DC percentage is 75.

Example Run – Circleville Event in Central Utah (continued)

```
def setupJob(receiverTuple):
    receiverIndex = receiverTuple[0]
    receiverCoord = receiverTuple[1]
    thisName = os.path.join(simulationDirectory,
                            simCollectionDirectory,
                            "Receiver_{:02d}".format(receiverIndex))
    sim = simulator.CUDA2DSimulation(simDir=thisName,numGPUs=1,simTypes=['PSV'])
    rec = simulator.Receiver(receiverCoord)
    sim.setReceiver(rec)
    sim.setSource(source)

    model = simulator.crustOneModel(beginPad=50.0, endPad=50.0,
                                    h=0.5, minDepth=0.0, maxDepth=50.0)
    model.setGeometryFromSim(sim)
    model.getProfile()
    model.addPerturbations()
    model.outputProfile()
    model.imageProfile(showPlots=False)
    sim.setModel(model)
    sim.setSimulationParameters(nt=4001, dt=0.03, itrecord=1, itprint=100)

    return sim

simPool = multiprocessing.Pool()
simCollection = simulator.simulationCollection(simPool.map(setupJob, enumerate(receiverCoords)))

gpuPool=[0,1]
simCollection.execute(gpuPool)

vertStream = obspy.Stream()
for sim in simCollection:
    sim.processData()
    vertStream += sim.streamZexp
vertStream.write(os.path.join(simulationDirectory,
                              simCollectionDirectory,
                              simCollectionDirectory+"_Section.pkl"),format="PICKLE")
```

This function is used by multiprocessing to setup jobs in parallel

Setup the name and get receiver coordinates for this simulation

Instantiate simulation and add source and receiver

Create the velocity model object for the simulation. Here we set the geometry from the simulation object's source and receiver coordinates, we extract a profile from CRUST1.0, add crustal scatterers, output the model and model images, and, finally, add the model to the simulation object.

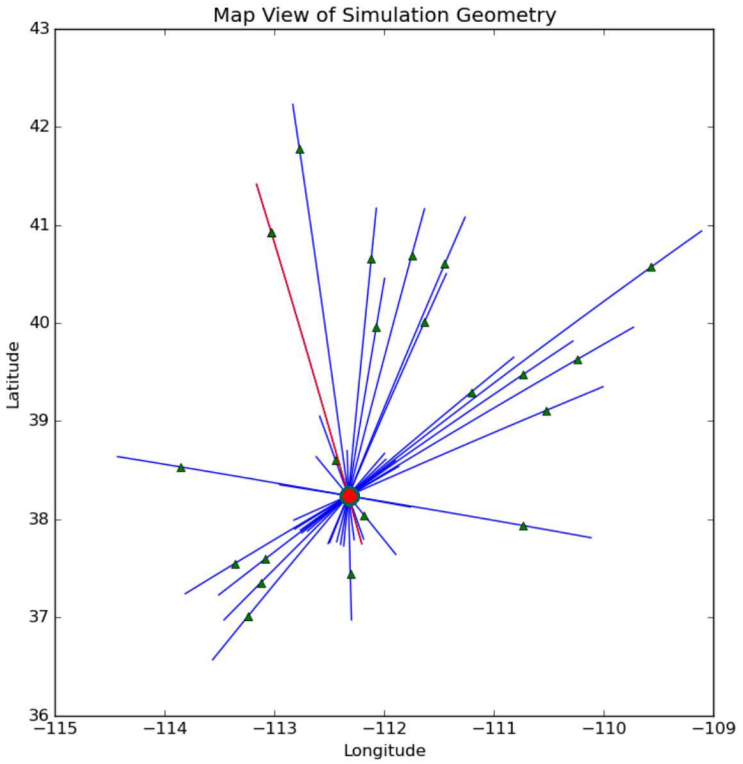
The *setupJob* function returns the simulation object with source, receiver and model members defined

The simulations (one per receiver) are setup in parallel and returned as a *simulationCollection* object

The simulations are run, being dynamically assigned to either GPU0 or GPU1 as they become available

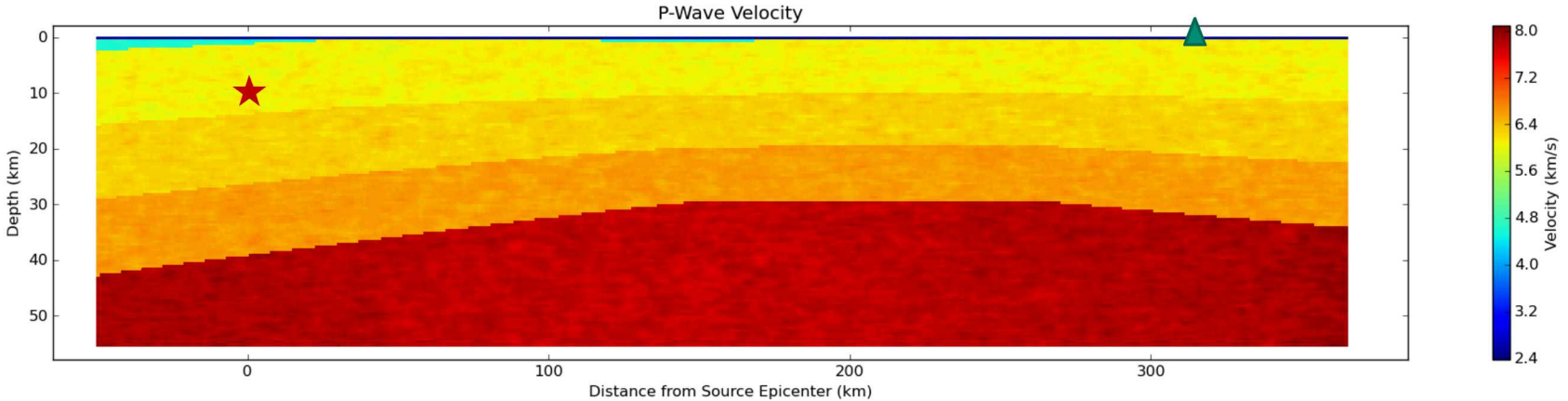
Finally, the computed synthetic data are collected from across the simulations into an obspy stream object and output to file. The data from each simulation are also stored in obspy streams in each simulation directory.

Example Run – Circleville Event in Central Utah - Output

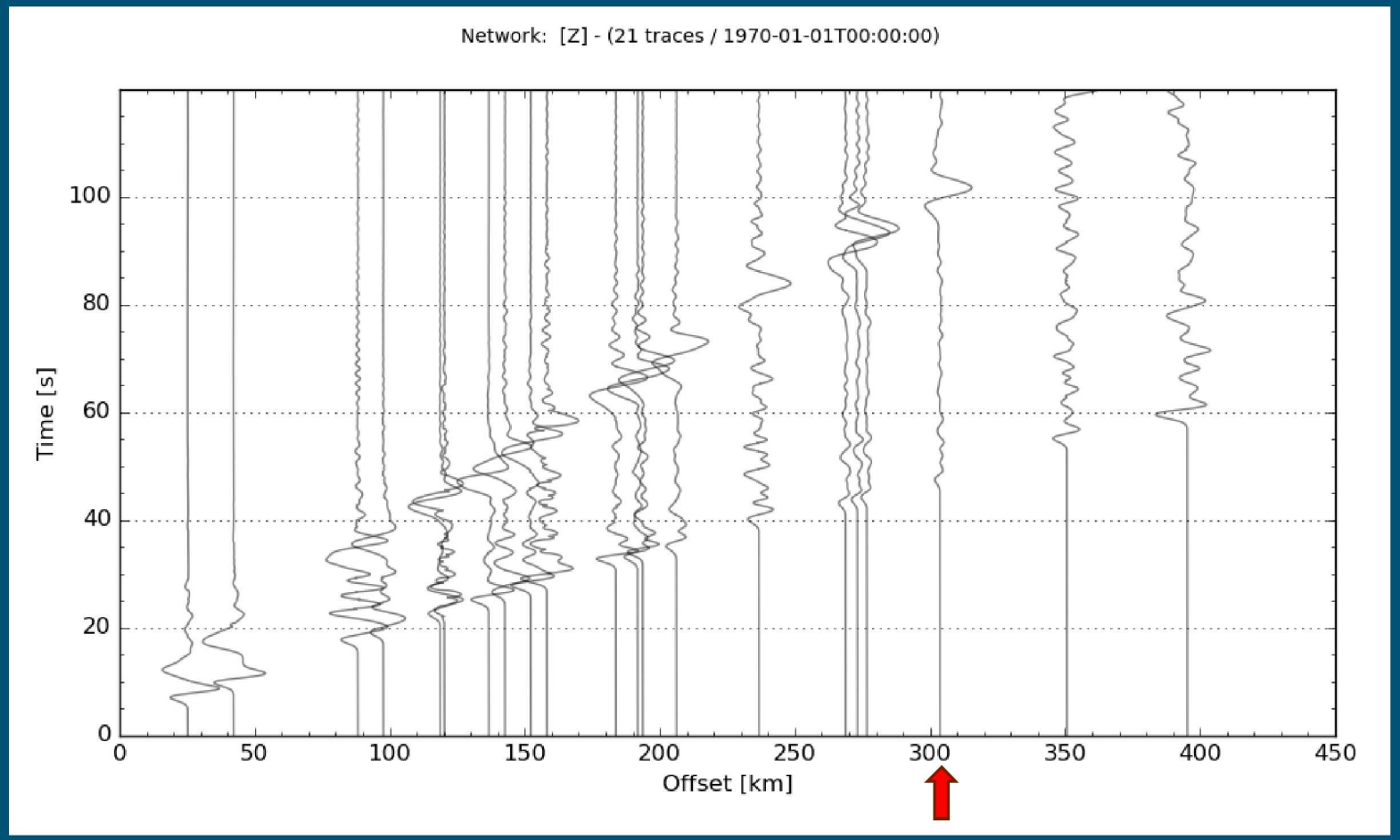


Map of the simulation showing all station locations (green triangles) and the source location (red dot). Each simulation profile is shown with a blue or red line.

P-Wave velocities extracted from CRUST1.0 along the red profile shown on the map. The source location is shown with a red star and the receiver with a green triangle.



Example Run – Circleville Event in Central Utah - Output



Record section of the vertical component of the output from the Circleville event simulation. This figure was generated directly from the obspy stream generated by the *dataProcessing()* method of the simulation object. The seismogram modeled using the velocity profile shown on the previous slide is marked with a red arrow.

Future Work

Scientific:

- Begin to study the effect of different types of crustal velocity anomalies on Pg, Pn, Lg and Sg wave propagation at local and regional distances. This will require setting up simulations to highlight these phases and compare seismograms from a base model to those simulated using a model with various crustal anomalies in Vp, Vs and density. The simulator's current state allows this work to start immediately.

Code Development:

- Add in support for the Earth flattening transformation needed for teleseismic waveform modeling. (Near Term).
- Add in a ray tracer to allow wavefield propagation to be compared with an infinite frequency response (Medium Term)
- Add alternative methods by which seismograms can be computed, using, for example, SpecFEM2D, reflectivity and integral transform methods (Long Term)
- Upgrades and bug fixes, add user-requested features (Forever and Always)