Notebook

October 31, 2017

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt

        %matplotlib inline
```

Preface: Installing Devito (do not include in manuscipt)

This tutorial and the coming second part are based on Devito version 3.1.0. It requires the installation of the full software with examples, not only the code generation API. To install:

That final dot is important, don't miss it out!

0.0.1 Geophysics tutorial

1 Full-waveform inversion 1: forward modeling

Mathias Louboutin1*, Philipp Witte1, Michael Lange2, Navjot Kukreja2, Fabio Luporini2, Gerard Gorman2, and Felix J. Herrmann1,3

1 Seismic Laboratory for Imaging and Modeling (SLIM), The University of British Columbia

2 Imperial College London, London, UK

3 now at Georgia Institute of Technology, USA

Corresponding author: mloubout@eoas.ubc.ca

Since its re-introduction by Pratt (1999), full-waveform inversion (FWI) has gained a lot of attention in geophysical exploration because of its ability to build high resolution velocity models more or less automatically in areas of complex geology. While there is an extensive and growing literature on the topic, publications focus mostly on technical aspects, making this topic inaccessible for a broader audience due to the lack of simple introductory resources for newcomers to geophysics. We will accomplish this by providing a hands-on walkthrough of FWI using Devito (Lange et al. 2016), a system based on domain-specific languages that automatically generates code for time-domain finite-differences.

As usual, this tutorial is accompanied by all the code you need to reproduce the figures. Go to github.com/seg/tutorials-2017 and follow the links. In the Notebook, we describe how to simulate synthetic data for a specified source and receiver setup and how to save the corresponding wavefields and shot records. In part two of this series, we will address how to calculate model updates, i.e. gradients of the FWI objective function, via adjoint modeling. Finally, in part three we will demonstrate how to use this gradient as part of an optimization framework for inverting an unknown velocity model.

1.1 Introduction

Devito provides a concise and straightforward computational framework for discretizing wave equations, which underlie all FWI frameworks. We will show that it generates verifiable executable code at run time for wave propagators associated with forward and (in part 2) adjoint wave equations. Devito frees the user from the recurrent and time-consuming development of performant time-stepping codes and allows the user to concentrate on the geophysics of the problem rather than on low-level implementation details of wave-equation simulators. This tutorial covers the conventional adjoint-state formulation of full-waveform tomography (Tarantola 1984) that underlies most of the current methods referred to as full-waveform inversion (Virieux and Operto 2009). While other formulations have been developed to improve the convergence of FWI for poor starting models, in these tutorials we will concentrate on the standard formulation that relies on the combination of a forward/adjoint pair of propagators and a correlation-based gradient. In part one of this tutorial, we discuss how to set up wave simulations for inversion, including how to express the wave equation in Devito symbolically and how to deal with the acquisition geometry.

What is FWI?

FWI tries to iteratively minimize the difference between data that was acquired in a seismic survey and synthetic data that is generated from a wave simulator with an estimated (velocity) model of the subsurface. As such, each FWI framework essentially consists of a wave simulator for forward modeling the predicted data and an adjoint simulator for calculating a model update from the data misfit. This first part of this tutorial is dedicated to the forward modeling part and demonstrates how to discretize and implement the acoustic wave equation using Devito.

1.2 Wave simulations for inversion

The acoustic wave equation with the squared slowness m, defined as $m(x, y) = c^{-2}(x, y)$ with $c(x, y)$ being the unknown spatially varying wavespeed, is given by:

$$m \frac{\mathrm{d}^2 u(t,x,y)}{\mathrm{d}t^2} \; - \; \Delta u(t,x,y) \; + \; \eta(x,y) \frac{\mathrm{d}u(t,x,y)}{\mathrm{d}t} \; = \; q(t,x,y;x_\mathrm{s},y_\mathrm{s}), \qquad (1)$$

where Δ is the Laplace operator, $q(t, x, y; x_\mathrm{s}, y_\mathrm{s})$ is the seismic source, located at $(x_\mathrm{s}, y_\mathrm{s})$ and $\eta(x, y)$ is a space-dependent dampening parameter for the absorbing boundary layer (Cerjan et al. 1985). As shown in Figure 1, the physical model is extended in every direction by `nbpml` grid points to mimic an infinite domain. The dampening term $\eta\,\mathrm{d}u/\mathrm{d}t$ attenuates the waves in the dampening layer and prevents waves from reflecting at the model boundaries. In Devito, the discrete representations of m and η are contained in a `model` object that contains a `grid` object with all relevant information such as the origin of the coordinate system, grid spacing, size of the model and dimensions `time, x, y`:

```
In [2]: # FIGURE 1
        from IPython.display import HTML
        HTML("../Figures/Figure1_composed.svg")
```

```
Out[2]: <IPython.core.display.HTML object>
```

Figure 1: (a) Diagram showing the model domain, with the perfectly matched layer (PML) as an absorbing layer to attenuate the wavefield at the model boundary. (b) The example model used in this tutorial, with the source and receivers indicated. The grid lines show the cell boundaries.

```
In [3]: # NOT FOR MANUSCRIPT
        from examples.seismic import Model, plot_velocity

        # Define a velocity model. The velocity is in km/s
        v = np.empty((101, 101), dtype=np.float32)
        v[:, :51] = 1.5
        v[:, 51:] = 2.5
```

```
In [4]: model = Model(vp=v,                # A velocity model.
                      origin=(0, 0),       # Top left corner.
                      shape=(101, 101),    # Number of grid points.
                      spacing=(10, 10),    # Grid spacing in m.
                      nbpml=40)            # boundary layer.
```

In the `Model` instantiation, `vp` is the velocity in km/s, `origin` is the origin of the physical model in meters, `spacing` is the discrete grid spacing in meters, `shape` is the number of grid points in each dimension and `nbpml` is the number of grid points in the absorbing boundary layer. Is is important to note that `shape` is the size of the physical domain only, while the total number of grid points, including the absorbing boundary layer, will be automatically derived from `shape` and `nbpml`.

1.3 Symbolic definition of the wave propagator

To model seismic data by solving the acoustic wave equation, the first necessary step is to discretize this partial differential equation (PDE), which includes discrete representations of the velocity model and wavefields, as well as approximations of the spatial and temporal derivatives using finite-differences (FD). Unfortunately, implementing these finite-difference schemes in low-level code by hand is error prone, especially when we want performant and reliable code.

The primary design objective of Devito is to allow users to define complex matrix-free finite-difference approximations from high-level symbolic definitions, while employing automated code generation to create highly optimized low-level C code. Using the symbolic algebra package SymPy (Meurer et al. 2017) to facilitate the automatic creation of derivative expressions, Devito generates computationally efficient wave propagators.

At the core of Devito's symbolic API are symbolic types that behave like SymPy function objects, while also managing data:

- `Function` objects represent a spatially varying function discretized on a regular Cartesian grid. For example, a function symbol `f = Function(name='f', grid=model.grid, space_order=2)` is denoted symbolically as `f(x, y)`. The objects provide auto-generated symbolic expressions for finite-difference derivatives through shorthand expressions like `f.dx` and `f.dx2` for the first and second derivative in `x`.
- `TimeFunction` objects represent a time-dependent function that has time as the leading dimension, for example `g(time, x, y)`. In addition to spatial derivatives `TimeFunction` symbols also provide time derivatives `g.dt` and `g.dt2`.
- `SparseFunction` objects represent sparse components, such as sources and receivers, which are usually distributed sparsely and often located off the computational grid — these objects also therefore handle interpolation onto the model grid.

To demonstrate Devito's symbolic capabilities, let us consider a time-dependent function $\mathbf{u}(\text{time}, x, y)$ representing the discrete forward wavefield:

```
In [5]: # NOT FOR MANUSCRIPT
        from devito import TimeFunction

        t0 = 0.      # Simulation starts a t=0
        tn = 1000.   # Simulation last 1 second (1000 ms)
        dt = model.critical_dt  # Time step from model grid spacing

        nt = int(1 + (tn-t0) / dt)  # Discrete time axis length
        time = np.linspace(t0, tn, nt)  # Discrete modelling time
```

```
In [6]: u = TimeFunction(name="u", grid=model.grid,
                         time_order=2, space_order=2,
                         save=True, time_dim=nt)
```

where the `grid` object provided by the `model` defines the size of the allocated memory region, `time_order` and `space_order` define the default discretization order of the derived derivative expressions.

We can now use this symbolic representation of our wavefield to generate simple discretized expressions for finite-difference derivative approximations using shorthand expressions, such as `u.dt` and `u.dt2` to denote $\frac{\mathrm{d}u}{\mathrm{d}t}$ and $\frac{\mathrm{d}^2u}{\mathrm{d}t^2}$ respectively:

```
In [7]: # NOT FOR MANUSCRIPT
        u

Out[7]: u(time, x, y)

In [8]: u.dt

Out[8]: -u(time - dt, x, y)/(2*dt) + u(time + dt, x, y)/(2*dt)

In [9]: u.dt2

Out[9]: -2*u(time, x, y)/dt**2 + u(time - dt, x, y)/dt**2 + u(time + dt, x, y)/dt**
```

Using the automatic derivation of derivative expressions, we can now implement a discretized expression for Equation 1 without the source term $q(x, y, t; x_s, y_s)$. The `model` object, which we created earlier, already contains the squared slowness $\mathbf{m}$ and damping term η as `Function` objects:

```
In [10]: pde = model.m * u.dt2 - u.laplace + model.damp * u.dt
```

If we write out the (second order) second time derivative `u.dt2` as shown earlier and ignore the damping term for the moment, our `pde` expression translates to the following discrete the wave equation:

$$\frac{\mathbf{m}}{\mathrm{dt}^2}\Big(\mathbf{u}[\text{time}-\mathrm{dt}] - 2\mathbf{u}[\text{time}] + \mathbf{u}[\text{time}+\mathrm{dt}]\Big) - \Delta\mathbf{u}[\text{time}] = 0, \quad \text{time} = 1\cdots n_{t-1} \qquad (2)$$

with time being the current time step and dt being the time stepping interval. To propagate the wavefield, we rearrange to obtain an expression for the wavefield $\mathbf{u}(\mathrm{time}+\mathrm{dt})$ at the next time step. Ignoring the damping term once again, this yields:

$$\mathbf{u}[\mathrm{time}+\mathrm{dt}] = 2\mathbf{u}[\mathrm{time}] - \mathbf{u}[\mathrm{time}-\mathrm{dt}] + \frac{\mathrm{dt}^2}{\mathbf{m}}\Delta\mathbf{u}[\mathrm{time}] \qquad (3)$$

We can rearrange our `pde` expression automatically using the SymPy utility function `solve`, then create an expression which defines the update of the wavefield for the new time step $\mathbf{u}(\mathrm{time}+\mathrm{dt})$, with the command `u.forward`:

```
In [11]: # NOT FOR MANUSCRIPT
         from devito import Eq
         from sympy import solve
```

```
In [12]: stencil = Eq(u.forward, solve(pde, u.forward)[0])
```

`stencil` represents the finite-difference approximation derived from Equation 3, including the finite-difference approximation of the Laplacian and the damping term. Although it defines the update for a single time step only, Devito knows that we will be solving a time-dependent problem over a number of time steps because the wavefield `u` is a `TimeFunction` object.

1.4 Setting up the acquisition geometry

The expression for time stepping we derived in the previous section does not contain a seismic source function yet, so the update for the wavefield at a new time step is solely defined by the two previous wavefields. However as indicated in Equation 1, wavefields for seismic experiments are often excited by an active (impulsive) source $q(x, y, t; x_\mathrm{s})$, which is a function of space and time (just like the wavefield `u`). To include such a source term in our modeling scheme, we simply add the the source wavefield as an additional term to Equation 3:

$$\mathbf{u}[\mathrm{time}+\mathrm{dt}] = 2\mathbf{u}[\mathrm{time}] - \mathbf{u}[\mathrm{time}-\mathrm{dt}] + \frac{\mathrm{dt}^2}{\mathbf{m}}\Big(\Delta\mathbf{u}[\mathrm{time}] + \mathbf{q}[\mathrm{time}]\Big). \qquad (4)$$

Since the source appears on the right-hand side in the original equation (Equation 1), the term also needs to be multiplied with $\frac{\mathrm{dt}^2}{\mathbf{m}}$ (this follows from rearranging Equation 2, with the source on the right-hand side in place of 0). Unlike the discrete wavefield `u` however, the source `q` is typically localized in space and only a function of time, which means the time-dependent source wavelet is injected into the propagating wavefield at a specified source location. The same applies when we sample the wavefield at receiver locations to simulate a shot record, i.e. the simulated wavefield needs to be sampled at specified receiver locations only. Source and receiver both do not necessarily coincide with the modeling grid.

Here, `RickerSource` acts as a wrapper around `SparseFunction` and models a Ricker wavelet with a peak frequency `f0` and source coordinates `src_coords`:

```
In [13]: # NOT FOR MANUSCRIPT
         from examples.seismic import RickerSource

         # Src is halfway across model, at depth of 20 m.
         x_extent, _ = model.domain_size
         src_coords = [x_extent/2, 20]
```

```
In [14]: f0 = 0.010  # kHz, peak frequency.
         src = RickerSource(name='src', grid=model.grid, f0=f0,
                            time=time, coordinates=src_coords)
```

The `src.inject` function now injects the current time sample of the Ricker wavelet (weighted with $\frac{\mathrm{dt}^2}{\mathbf{m}}$ as shown in Equation 4) into the updated wavefield `u.forward` at the specified coordinates.

```
In [15]: src_term = src.inject(field=u.forward,
                               expr=src * dt**2 / model.m,
                               offset=model.nbpml)
```

To extract the wavefield at a predetermined set of receiver locations, there is a corresponding wrapper function for receivers as well, which creates a `SparseFunction` object for a given number `npoint` of receivers, number `nt` of time samples, and specified receiver coordinates `rec_coords`:

```
In [16]: # NOT FOR MANUSCRIPT
         from examples.seismic import Receiver

         # Recs are distributed across model, at depth of 20 m.
         x_locs = np.linspace(0, model.domain_size[0], 101)
         rec_coords = [(x, 20) for x in x_locs]

In [17]: rec = Receiver(name='rec', npoint=101, ntime=nt,
                        grid=model.grid, coordinates=rec_coords)
```

Rather than injecting a function into the model as we did for the source, we now simply save the wavefield at the grid points that correspond to receiver positions and interpolate the data to their exact possibly of the computatational grid location (`rec.interpolate` in Devito).

```
In [18]: rec_term = rec.interpolate(u, offset=model.nbpml)

In [19]: # NOT FOR MANUSCRIPT
         # PLOTS HALF OF FIGURE 1.
         import matplotlib.patches as patches
         from matplotlib.ticker import MultipleLocator

         fig = plt.figure(figsize=(9,9))

         extent = [model.origin[0], model.origin[0] + 1e-3 * model.shape[0] * model
                   model.origin[1] + 1e-3*model.shape[1] * model.spacing[1], model.

         model_param = dict(vmin=1.5, vmax=2.5, cmap="GnBu", aspect=1, extent=exter

         ax0 = fig.add_subplot(111)
         im = plt.imshow(np.transpose(v), **model_param)
         cb = plt.colorbar(shrink=0.8)
         ax0.set_ylabel('Depth (km)',fontsize=20)
```

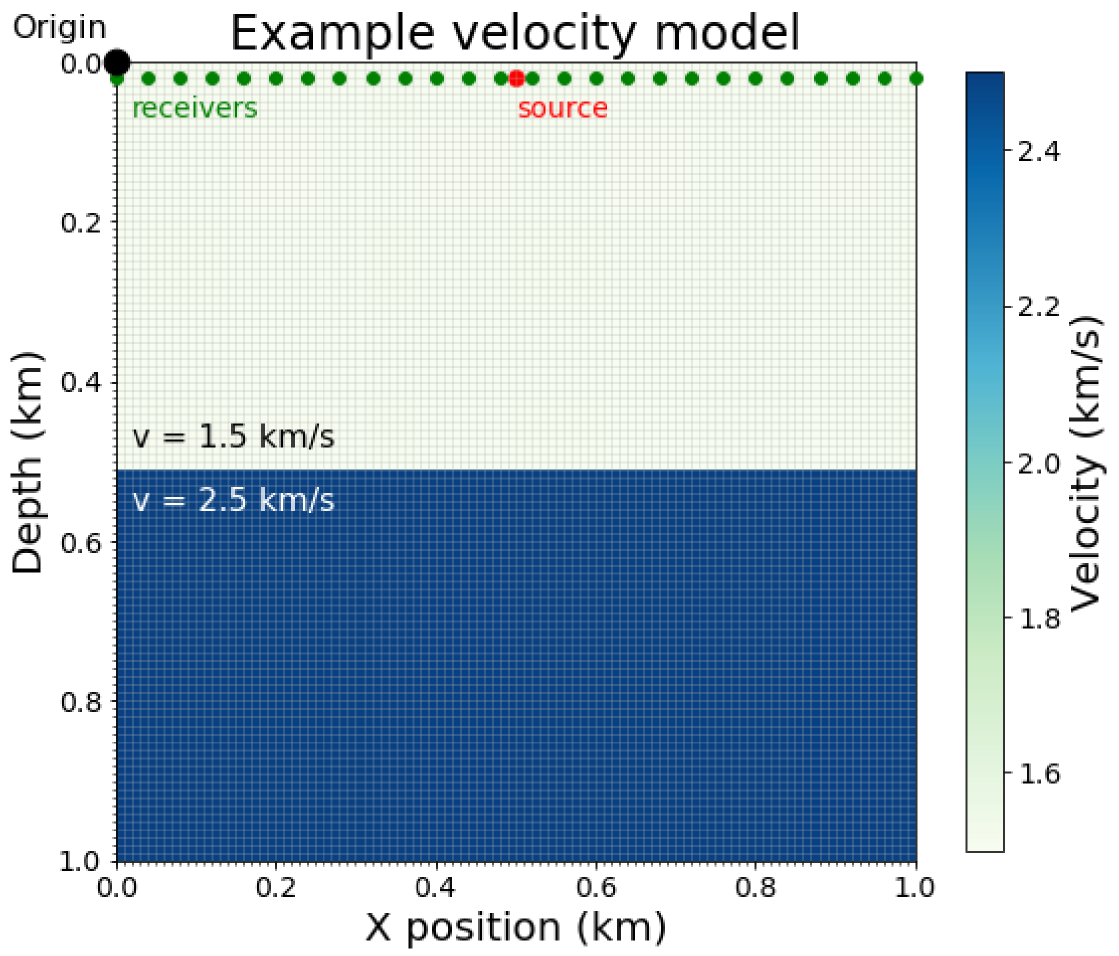
```
ax0.set_xlabel('X position (km)', fontsize=20)
cb.set_label('Velocity (km/s)', fontsize=20)
plt.xticks(fontsize=14)
plt.yticks(fontsize=14)
cb.ax.tick_params(labelsize=14)

plt.scatter(*(rec.coordinates.data[::4, :].T/1000), c='green', clip_on=Fal
plt.text(*rec.coordinates.data[0].T/1000 + [0.02, 0.05], "receivers", colo
plt.scatter(*(src.coordinates.data.squeeze()/1000), c='red', s=60)
plt.text(*src.coordinates.data[0]/1000 + [0, 0.05], "source", color='red',
plt.scatter(0, 0, c='black', s=160, clip_on=False, zorder=101)
plt.text(-0.01, -0.03, "Origin", color='k', size=16, ha="right")
plt.text(0.02, 0.5-0.03, "v = 1.5 km/s", color='k', size=16, ha="left", va
plt.text(0.02, 0.5+0.05, "v = 2.5 km/s", color='w', size=16, ha="left", va
plt.title("Example velocity model", color='k', size=24)
plt.xlim((0, 1))
plt.ylim((1, 0))

minorLocator = MultipleLocator(1/100)
ax0.xaxis.set_minor_locator(minorLocator)
ax0.yaxis.set_minor_locator(minorLocator)

plt.grid(which='minor', alpha=0.3)

plt.savefig("../Figures/model.pdf", dpi=400)
plt.savefig("../Figures/model.png")
plt.show()
```



1.5 Forward simulation

We can now define our forward propagator by adding the source and receiver terms to our stencil object:

```
In [20]: # NOT FOR MANUSCRIPT
         from devito import Operator
```

```
In [21]: op_fwd = Operator([stencil] + src_term + rec_term)
```

The symbolic expressions used to create `Operator` contain sufficient meta-information for Devito to create a fully functional computational kernel. The dimension symbols contained in the symbolic function object (`time, x, y`) define the loop structure of the created code,while allowing Devito to automatically optimize the underlying loop structure to increase execution speed.

The size of the loops and spacing between grid points is inferred from the symbolic `Function` objects and associated `model.grid` object at run-time. As a result, we can invoke the generated kernel through a simple Python function call by supplying the number of timesteps `time` and the timestep size `dt`. The user data associated with each `Function` is updated in-place during operator execution, allowing us to extract the final wavefield and shot record directly from the symbolic function objects without unwanted memory duplication:

```
In [22]: # Generate wavefield snapshots and a shot record.
         op_fwd(time=nt, dt=model.critical_dt)
```

```
CustomCompiler: compiled /var/folders/8x/2cdqc7_57plfk5txbsszysbc0000gn/T/devito-sc
===================================================================================
Section section_1<595,1> with OI=0.80 computed in 0.000 s [0.45 GFlops/s]
Section section_2<595,101> with OI=1.50 computed in 0.001 s [2.22 GFlops/s]
Section main<595,180,180> with OI=3.27 computed in 0.068 s [15.10 GFlops/s, 0.28 GF
===================================================================================
```

When this has finished running, the resulting wavefield is stored in `u.data` and the shot record is in `rec.data`. We can easily plot this 2D array as an image, as shown in Figure 2.

```
In [23]: # NOT FOR MANUSCRIPT
         # GENERATES FIGURE 2
         from matplotlib import cm

         fig1 = plt.figure(figsize=(10,10))
         l = plt.imshow(rec.data, vmin=-1, vmax=1, cmap=cm.gray, aspect=1,
                        extent=[model.origin[0], model.origin[0] + 1e-3*model.shape
                                1e-3*tn, t0])
         plt.xlabel('X position (km)',  fontsize=20)
         plt.ylabel('Time (s)',  fontsize=20)
         plt.tick_params(labelsize=20)

         plt.savefig("../Figures/shotrecord.png", dpi=400)
         plt.savefig("../Figures/shotrecord.pdf")
         plt.show()
```

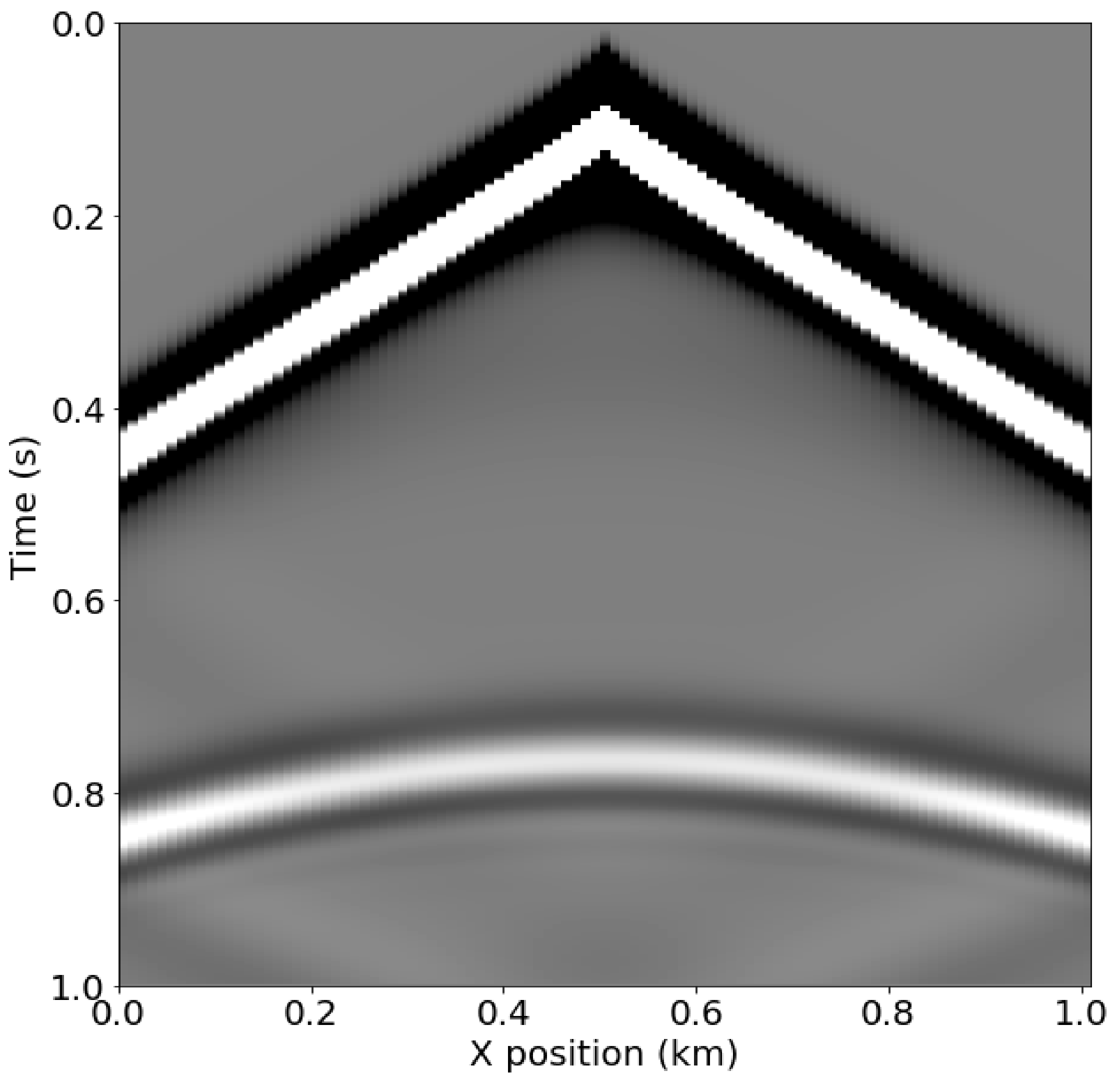


Figure 2. The shot record generated by Devito for the example velocity model.

As demonstrated in the notebook, a movie of snapshots of the forward wavefield can also be generated by capturing the wavefield at discrete time steps. Figure 3 shows three timesteps from the movie.

```
In [24]: # NOT FOR MANUSCRIPT
         # GENERATES FIGURE 3
         fig = plt.figure(figsize=(15, 5))

         times = [200, 300, 400]

         extent = [model.origin[0], model.origin[0] + 1e-3 * model.shape[0] * model
                   model.origin[1] + 1e-3*model.shape[1] * model.spacing[1], model.
```

```
data_param = dict(vmin=-1e0, vmax=1e0, cmap=cm.Greys, aspect=1, extent=ext
model_param = dict(vmin=1.5, vmax=2.5, cmap=cm.GnBu, aspect=1, extent=exte

ax0 = fig.add_subplot(131)
_ = plt.imshow(np.transpose(u.data[times[0],40:-40,40:-40]), **data_param)
_ = plt.imshow(np.transpose(v), **model_param)
ax0.set_ylabel('Depth (km)',  fontsize=20)
ax0.text(0.5, 0.08, "t = {:.0f} ms".format(time[times[0]]), ha="center", c

ax1 = fig.add_subplot(132)
_ = plt.imshow(np.transpose(u.data[times[1],40:-40,40:-40]), **data_param)
_ = plt.imshow(np.transpose(v), **model_param)
ax1.set_xlabel('X position (km)',  fontsize=20)
ax1.set_yticklabels([])
ax1.text(0.5, 0.08, "t = {:.0f} ms".format(time[times[1]]), ha="center", c

ax2 = fig.add_subplot(133)
_ = plt.imshow(np.transpose(u.data[times[2],40:-40,40:-40]), **data_param)
_ = plt.imshow(np.transpose(v), **model_param)
ax2.set_yticklabels([])
ax2.text(0.5, 0.08, "t = {:.0f} ms".format(time[times[2]]), ha="center", c

plt.savefig("../Figures/snaps.pdf")
plt.savefig("../Figures/snaps.png", dpi=400)
plt.show()
```

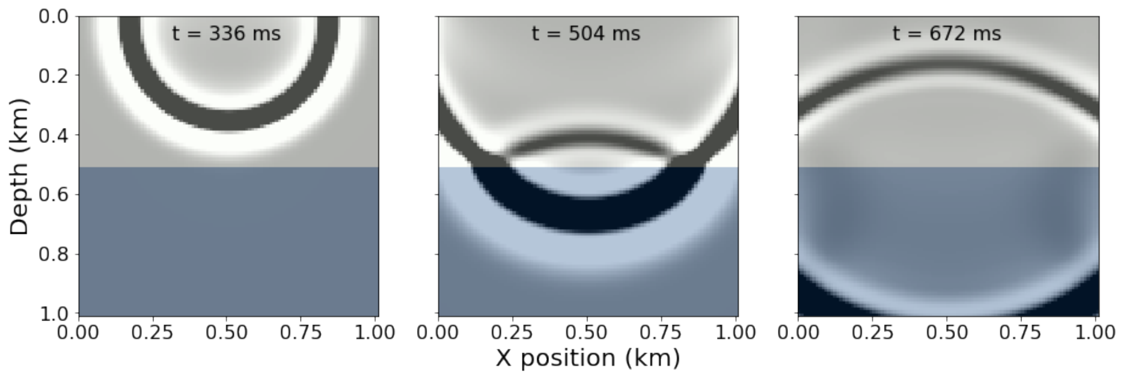


Figure 3. Three time steps from the wavefield simulation that resulted in the shot record in Figure 2. You can generate an animated version in the Notebook at github.com/seg.

```
In [26]: # NOT FOR MANUSCRIPT
         import matplotlib.animation as animation
         from IPython.display import HTML
```

```
fig = plt.figure()
im = plt.imshow(np.transpose(u.data[0,40:-40,40:-40]),
                cmap="Greys", animated=True, vmin=-1e0, vmax=1e0, aspect=1
                extent=[model.origin[0], model.origin[0] + 1e-3 * model.sh
                        model.origin[1] + 1e-3*model.shape[1] * model.spac
plt.xlabel('X position (km)',  fontsize=20)
plt.ylabel('Depth (km)',  fontsize=20)
plt.tick_params(labelsize=20)
im2 = plt.imshow(np.transpose(v), vmin=1.5, vmax=2.5, cmap=cm.GnBu, aspect
                 extent=[model.origin[0], model.origin[0] + 1e-3 * model.s
                         model.origin[1] + 1e-3*model.shape[1] * model.spa
def updatefig(i):
    im.set_array(np.transpose(u.data[i*5,40:-40,40:-40]))
    return im, im2

ani = animation.FuncAnimation(fig, updatefig, frames=np.linspace(0, nt/5-1
plt.close(ani._fig)
HTML(ani.to_html5_video())
```

```
Out[26]: <IPython.core.display.HTML object>
```

1.6 Conclusions

In this first part of the tutorial, we have demonstrated how to set up the discretized forward acoustic wave equations and its associated wave propagator with runtime code generation. While we limited our discussion to the constant density acoustic wave equation, Devito is capable of handling more general wave equations but this is a topic beyond this tutorial on simulating waves for inversion. In part two of our tutorial, we will show how to calculate a valid gradient of the FWI objective using the adjoint state method. In part three, we will demonstrate how to set up a complete matrix-free and scalable optimization framework for acoustic FWI.

1.7 Acknowledgments

This research was carried out as part of the SINBAD II project with the support of the member organizations of the SINBAD Consortium. This work was financially supported in part by EPSRC grant EP/L000407/1 and the Imperial College London Intel Parallel Computing Centre.

1.8 References

Cerjan, C., Kosloff, D., Kosloff, R., and Reshef, M., 1985, A nonreflecting boundary condition for discrete acoustic and elastic wave equations: GEOPHYSICS, 50, 705–708. doi:10.1190/1.1441945

Lange, M., Kukreja, N., Louboutin, M., Luporini, F., Zacarias, F. V., Pandolfo, V., Gorman, G., 2016, Devito: Towards a generic finite difference DSL using symbolic python: 6th workshop on python for high-performance and scientific computing. doi:10.1109/PyHPC.2016.9

Meurer A, Smith CP, Paprocki M, et al., 2017, SymPy: symbolic computing in Python. PeerJ Computer Science 3:e103 https://doi.org/10.7717/peerj-cs.103

Pratt, R. G., 1999, Seismic waveform inversion in the frequency domain, part 1: Theory and verification in a physical scale model: GEOPHYSICS, 64, 888–901. doi:10.1190/1.1444597

Tarantola, A., 1984, Inversion of seismic reflection data in the acoustic approximation: GEOPHYSICS, 49, 1259–1266. doi:10.1190/1.1441754

Virieux, J., and Operto, S., 2009, An overview of full-waveform inversion in exploration geophysics: GEOPHYSICS, 74, WCC1–WCC26. doi:10.1190/1.3238367

1.9 Supplemental material

- Devito documentation
- Devito source code and examples
- Tutorial notebooks with latest Devito/master