03 - Full-Waveform Inversion (FWI)

This notebook is the third in a series of tutorial highlighting various aspects of seismic inversion based on Devito operators. In this second example we aim to highlight the core ideas behind seismic inversion, where we create an image of the subsurface from field recorded data. This tutorial follows on the modelling tutorial and will reuse the modelling and velocity model.

Inversion requirement

Seismic inversion relies on two known parameters:

- **Field data** - or also called **recorded data**. This is a shot record corresponding to the true velocity model. In practice this data is acquired as described in the first tutorial. In order to simplify this tutorial we will fake field data by modelling it with the true velocity model.
- **Initial velocity model**. This is a velocity model that has been obtained by processing the field data. This model is a rough and very smooth estimate of the velocity as an initial estimate for the inversion. This is a necessary requirement for any optimization (method).

Inversion computational setup

In this tutorial, we will introduce the gradient operator. This operator corresponds to the imaging condition introduced in the previous tutorial with some minor modifications that are defined by the objective function (also referred to in the tutorial series as the *functional*, *f*) and its gradient, *g*. We will define these two terms in the tutorial too.

Notes on the operators

As we have already described the creation of a forward modelling operator, we will only call a wrapper function here. This wrapper already contains all the necessary operators for seismic modeling, imaging and inversion. Operators introduced for the first time in this tutorial will be properly described.

In [1]:

```
import numpy as np
%matplotlib inline

from devito import configuration
configuration['log-level'] = 'WARNING'
```

Computational considerations

As we will see, FWI is computationally extremely demanding, even more than RTM. To keep this tutorial as lightwight as possible we therefore again use a very small demonstration model. We also define here a few parameters for the final example runs that can be changed to modify the overall runtime of the tutorial.

In [2]:

```
fwi_iterations = 10  # Number of outer FWI iterations
```

True and smooth velocity models

We will use a very simple model domain, consisting of a circle within a 2D domain. We will again use the "true" model to generate our synthetic shot data and use a "smooth" model as our initial guess. In this case the smooth model is very smooth indeed - it is simply a constant background velocity without any features.

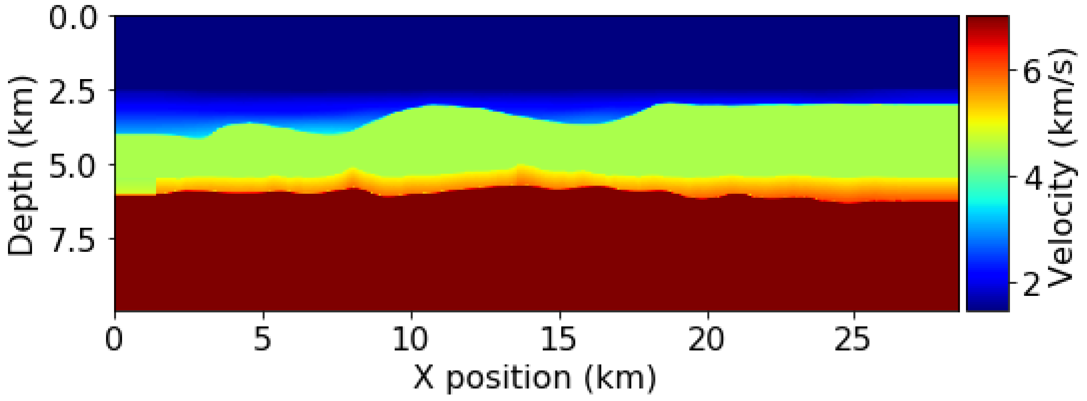
In [3]:

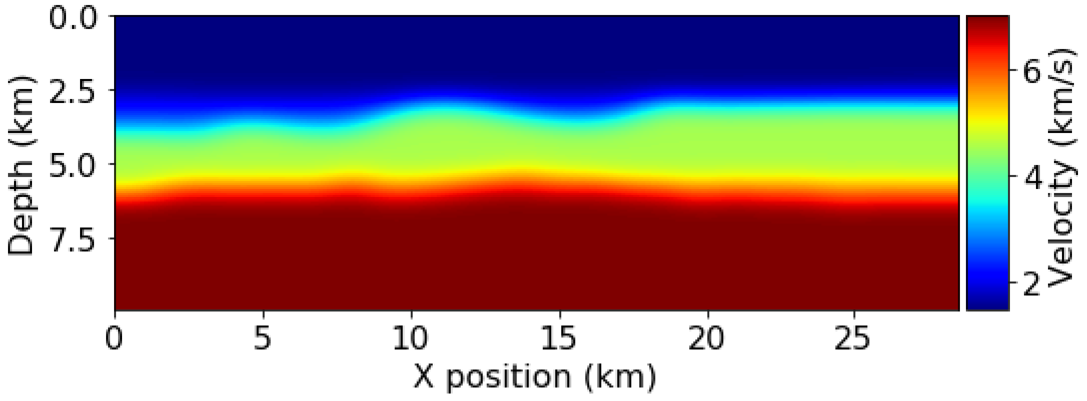
```
#IMPORT GATO DO MATO
shape = (571, 311)  # Number of lines (nx), number of columns (nz)

# Define a velocity matrix
v = np.empty(shape, dtype=np.float32)
v = np.loadtxt('GM_IL_1000.dat')

# Deleting the first column, transposing the matrix, redefining the shape,
# and passing the velocity from m/s to km/s
v = np.delete(v, 0, 1)
#v = np.transpose(v)
shape = v.shape # Number of grid point (nx, nz)
v = v/1000.
print('The new shape is ' + str(shape) + '(number of lines (nx), number of colum
ns (nz))')
```

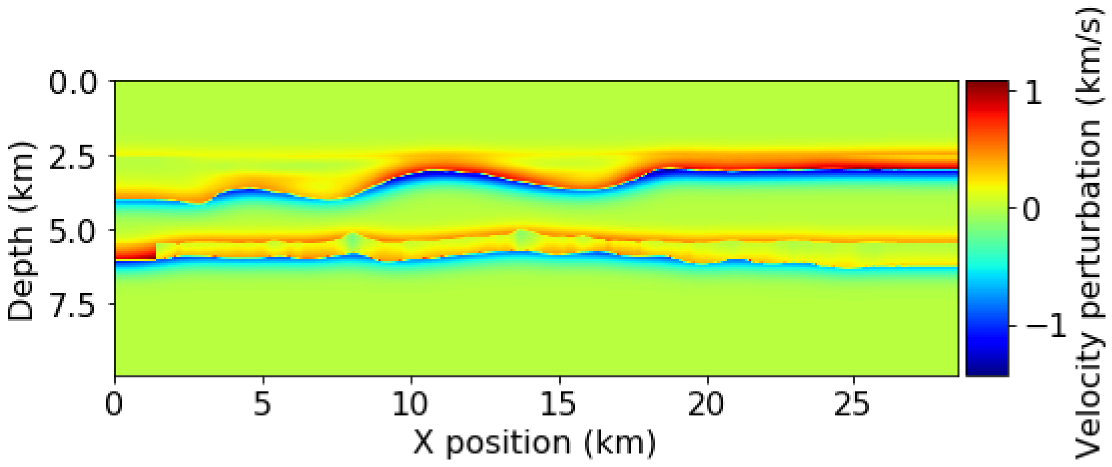
```
The new shape is (571, 310)(number of lines (nx), number of columns
(nz))
```

In [4]:

```
#NBVAL_IGNORE_OUTPUT
from examples.seismic import Model, plot_velocity, plot_perturbation, demo_model
from scipy import ndimage
from devito import *

# Define a physical size
shape = (571, 310)  # Number of lines (nx), number of columns (nz)
spacing = (50., 32.)  # Grid spacing in m.
origin = (0., 0.)  # What is the location of the top left corner. This is necess
ary to define
# the absolute location of the source and receivers

# With the velocity and model size defined, we can create the seismic model that
# encapsulates this properties. We also define the size of the absorbing layer a
s 10 grid points
model = Model(vp=v, origin=origin, shape=shape, spacing=spacing,
              space_order=2, nbl=30)


filter_sigma = (15, 15)
nshots = 285  # Need good covergae in shots, one every two grid points
nreceivers = 571  # One recevier every grid point
t0 = 0.
tn = 9000.  # Simulation last 3.5 second (3500 ms)
f0 = 0.010  # Source peak frequency is 25Hz (0.025 kHz)

# Create initial model and smooth the boundaries (for inversion)
model0 = Model(vp=v, origin=origin, shape=shape, spacing=spacing,
              space_order=2, nbl=30)
#model0 = demo_model('circle-isotropic', vp=4.0, vp_background=4.0,
#                      origin=origin, shape=shape, spacing=spacing, nbl=10,
#                     grid = model.grid)
model0.vp = ndimage.gaussian_filter(model0.vp.data, sigma=filter_sigma, order=0)

# Create initial model and smooth the boundaries (for comparison with the initia
l model)
model00 = Model(vp=v, origin=origin, shape=shape, spacing=spacing,
              space_order=2, nbl=30)
#model0 = demo_model('circle-isotropic', vp=4.0, vp_background=4.0,
#                      origin=origin, shape=shape, spacing=spacing, nbl=10,
#                     grid = model.grid)
model00.vp = ndimage.gaussian_filter(model0.vp.data, sigma=filter_sigma, order=0
)


plot_velocity(model)
plot_velocity(model0)
plot_perturbation(model0, model)
```





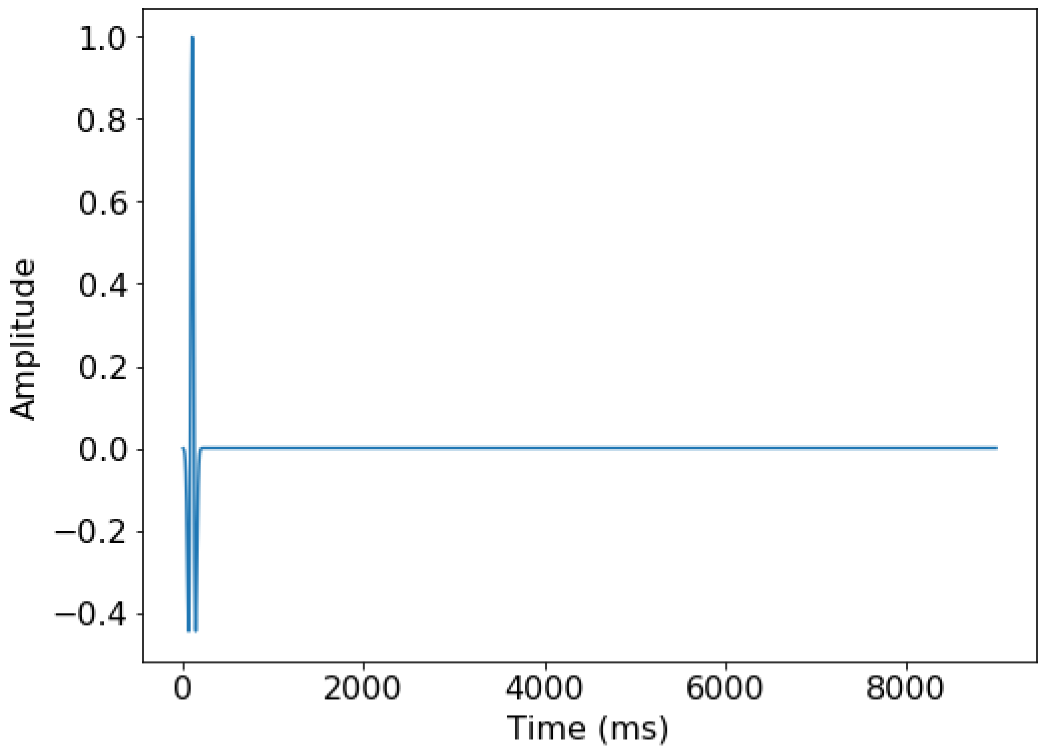


Acquisition geometry

In this tutorial, we will use the easiest case for inversion, namely a transmission experiment. The sources are located on one side of the model and the receivers on the other side. This allows to record most of the information necessary for inversion, as reflections usually lead to poor inversion results.

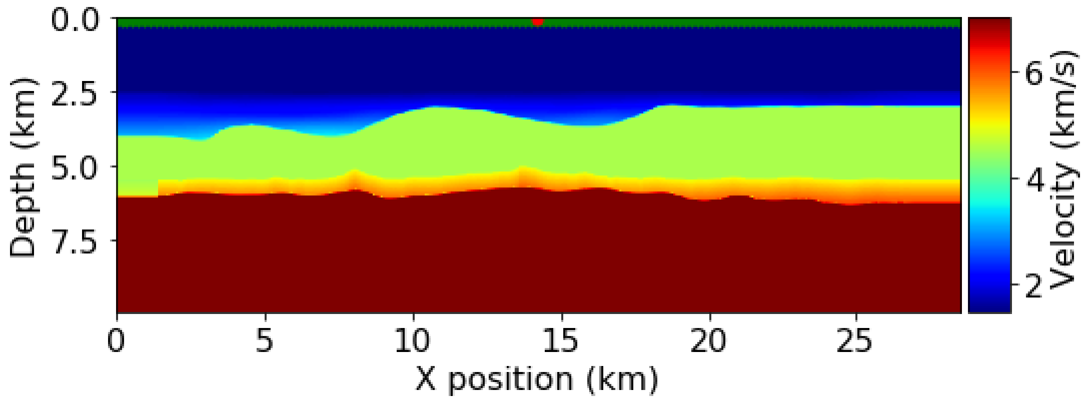
In [5]:

```
#NBVAL_IGNORE_OUTPUT
# Define acquisition geometry: source
from examples.seismic import AcquisitionGeometry

# First, position source centrally in all dimensions, then set depth
src_coordinates = np.empty((1, 2))
src_coordinates[0, :] = np.array(model.domain_size) * .5
src_coordinates[0, 1] = 20.  # Depth(m)


# Define acquisition geometry: receivers

# Initialize receivers for synthetic and imaging data
rec_coordinates = np.empty((nreceivers, 2))
rec_coordinates[:, 0] = np.linspace(0, model.domain_size[0], num=nreceivers)
rec_coordinates[:, 1] = 20. # Depth(m)

# Geometry
geometry = AcquisitionGeometry(model, rec_coordinates, src_coordinates, t0, tn,
f0=f0, src_type='Ricker')

# We can plot the time signature to see the wavelet
geometry.src.show()
```



In [6]:

```
# Plot acquisition geometry
plot_velocity(model, source=geometry.src_positions,
              receiver=geometry.rec_positions[::4, :])
```



True and smooth data

We can generate shot records for the true and smoothed initial velocity models, since the difference between them will again form the basis of our imaging procedure.

In [7]:

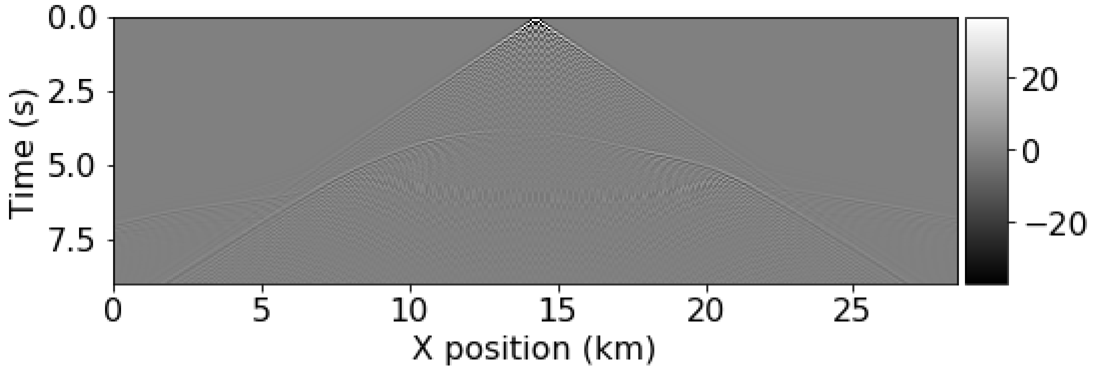
```
# Compute synthetic data with forward operator
from examples.seismic.acoustic import AcousticWaveSolver

solver = AcousticWaveSolver(model, geometry, space_order=4)
true_d, _, _ = solver.forward(vp=model.vp)
```

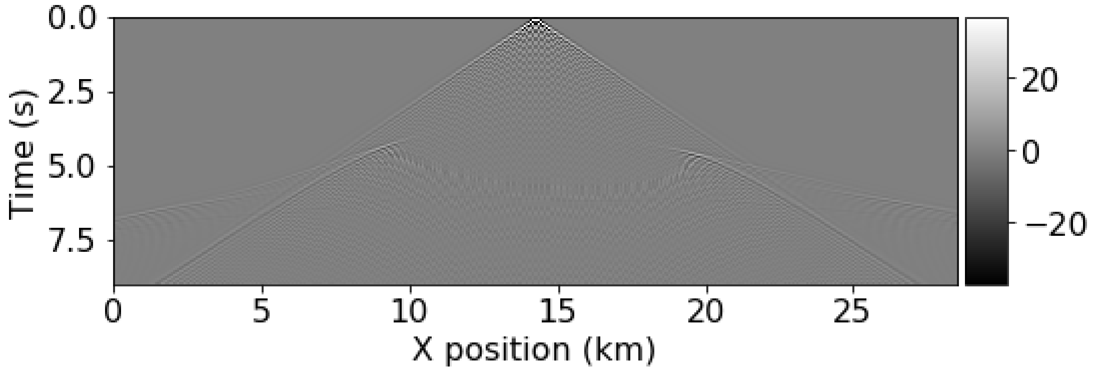
In [8]:

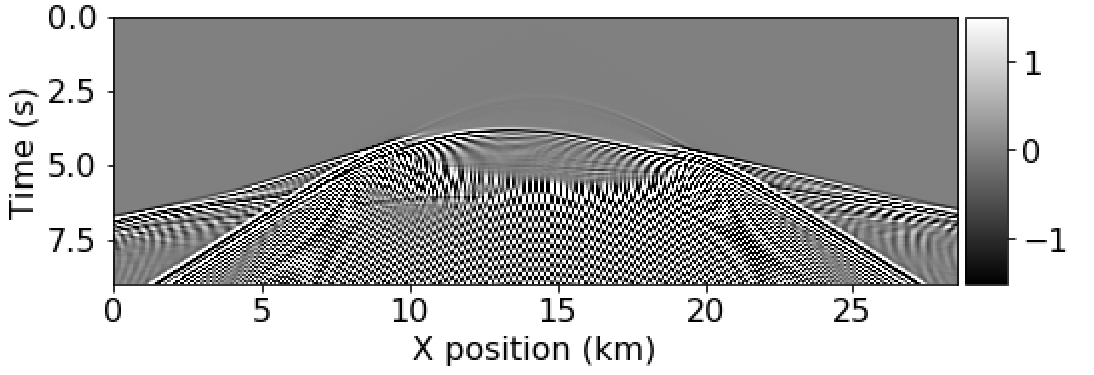
```
# Compute initial data with forward operator
smooth_d, _, _ = solver.forward(vp=model0.vp)
```

In [9]:

```
#NBVAL_IGNORE_OUTPUT
from examples.seismic import plot_shotrecord

# Plot shot record for true and smooth velocity model and the difference
plot_shotrecord(true_d.data, model, t0, tn)
plot_shotrecord(smooth_d.data, model, t0, tn)
plot_shotrecord(smooth_d.data - true_d.data, model, t0, tn)
```







Full-Waveform Inversion

Formulation

Full-waveform inversion (FWI) aims to invert an accurate model of the discrete wave velocity, $\mathbf{c}$, or equivalently the square slowness of the wave, $\mathbf{m} = \frac{1}{\mathbf{c}^2}$, from a given set of measurements of the pressure wavefield $\mathbf{u}$. This can be expressed as the following optimization problem [1, 2]:

$$\begin{aligned} \underset{\mathbf{m}}{\text{minimize}} \ \Phi_s(\mathbf{m}) &= \frac{1}{2}\left\lVert\mathbf{P}_r \mathbf{u} - \mathbf{d}\right\rVert_2^2 \\ \mathbf{u} &= \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^T \mathbf{q}_s, \end{aligned}$$

where $\mathbf{P}_r$ is the sampling operator at the receiver locations, $\mathbf{P}_s^T$ is the injection operator at the source locations, $\mathbf{A}(\mathbf{m})$ is the operator representing the discretized wave equation matrix, $\mathbf{u}$ is the discrete synthetic pressure wavefield, $\mathbf{q}_s$ is the corresponding pressure source and $\mathbf{d}$ is the measured data. It is worth noting that $\mathbf{m}$ is the unknown in this formulation and that multiple implementations of the wave equation operator $\mathbf{A}(\mathbf{m})$ are possible.

We have already defined a concrete solver scheme for $\mathbf{A}(\mathbf{m})$ in the first tutorial, including appropriate implementations of the sampling operator $\mathbf{P}_r$ and source term $\mathbf{q}_s$.

To solve this optimization problem using a gradient-based method, we use the adjoint-state method to evaluate the gradient $\nabla\Phi_s(\mathbf{m})$:

$$\nabla\Phi_s(\mathbf{m}) = \sum_{\mathbf{t}=1}^{n_t} \mathbf{u}[\mathbf{t}] \mathbf{v}_{tt}[\mathbf{t}] = \mathbf{J}^T\delta\mathbf{d}_s,$$

where n_t is the number of computational time steps, $\delta\mathbf{d}_s = \left(\mathbf{P}_r \mathbf{u} - \mathbf{d} \right)$ is the data residual (difference between the measured data and the modelled data), $\mathbf{J}$ is the Jacobian operator and $\mathbf{v}_{tt}$ is the second-order time derivative of the adjoint wavefield solving:

$$\mathbf{A}^T(\mathbf{m}) \mathbf{v} = \mathbf{P}_r^T \delta\mathbf{d}.$$

We see that the gradient of the FWI function is the previously defined imaging condition with an extra second-order time derivative. We will therefore reuse the operators defined previously inside a Devito wrapper.

FWI gradient operator

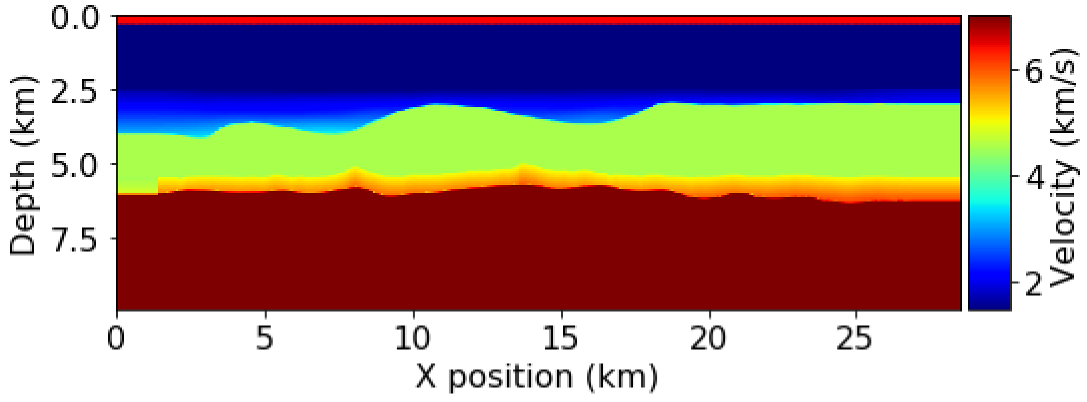
To compute a single gradient $\nabla\Phi_s(\mathbf{m})$ in our optimization workflow we again use `solver.forward` to compute the entire forward wavefield $\mathbf{u}$ and a similar pre-defined gradient operator to compute the adjoint wavefield `v`. The gradient operator provided by our `solver` utility also computes the correlation between the wavefields, allowing us to encode a similar procedure to the previous imaging tutorial as our gradient calculation:

- Simulate the forward wavefield with the background velocity model to get the synthetic data and save the full wavefield $\mathbf{u}$
- Compute the data residual
- Back-propagate the data residual and compute on the fly the gradient contribution at each time step.

This procedure is applied to multiple source positions and summed to obtain a gradient image of the subsurface. We again prepare the source locations for each shot and visualize them, before defining a single gradient computation over a number of shots as a single function.

In [10]:

```
#NBVAL_IGNORE_OUTPUT

# Prepare the varying source locations sources
source_locations = np.empty((nshots, 2), dtype=np.float32)
source_locations[:, 0] = np.linspace(0., 28550, num=nshots)
source_locations[:, 1] = 1.

plot_velocity(model, source=source_locations)
```



In [11]:

```
# Create FWI gradient kernel
from devito import Function, clear_cache, TimeFunction
from examples.seismic import Receiver

import scipy
def fwi_gradient(vp_in):
    # Create symbols to hold the gradient and residual
    grad = Function(name="grad", grid=model.grid)
    residual = Receiver(name='rec', grid=model.grid,
                        time_range=geometry.time_axis,
                        coordinates=geometry.rec_positions)
    objective = 0.

    # Creat forward wavefield to reuse to avoid memory overload
    u0 = TimeFunction(name='u', grid=model.grid, time_order=2, space_order=4,
                      save=geometry.nt)
    for i in range(nshots):
        # Important: We force previous wavefields to be destroyed,
        # so that we may reuse the memory.
        clear_cache()

        # Update source location
        geometry.src_positions[0, :] = source_locations[i, :]

        # Generate synthetic data from true model
        true_d, _, _ = solver.forward(vp=model.vp)

        # Compute smooth data and full forward wavefield u0
        u0.data.fill(0.)
        smooth_d, _, _ = solver.forward(vp=vp_in, save=True, u=u0)

        # Compute gradient from data residual and update objective function
        residual.data[:] = smooth_d.data[:] - true_d.data[:]

        objective += .5*np.linalg.norm(residual.data.flatten())**2
        solver.gradient(rec=residual, u=u0, vp=vp_in, grad=grad)

    return objective, -grad.data
```

Having defined our FWI gradient procedure we can compute the initial iteration from our starting model. This allows us to visualize the gradient alongside the model perturbation and the effect of the gradient update on the model.

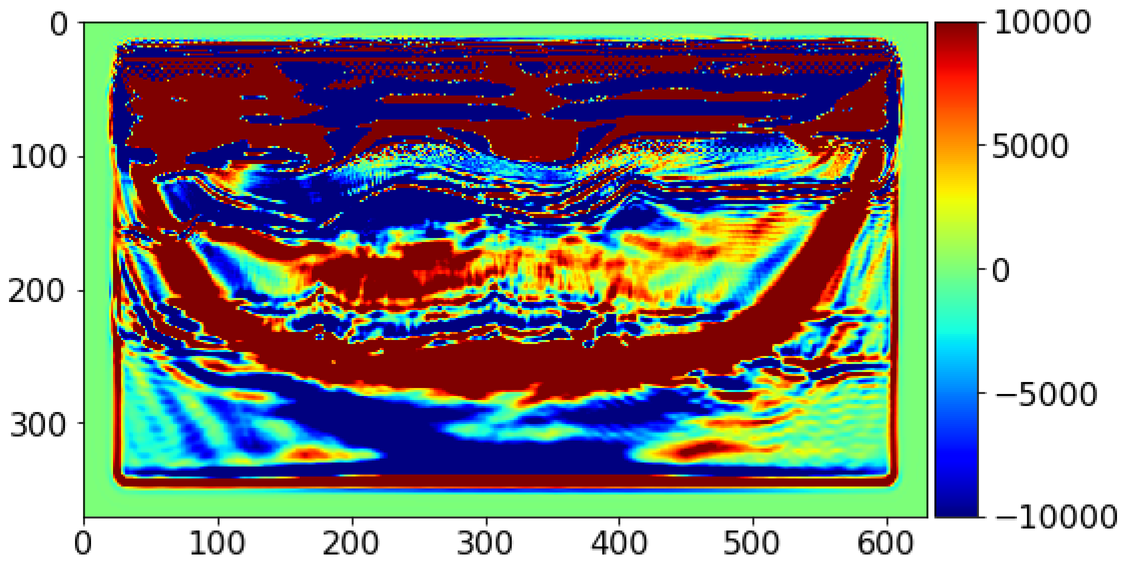
In [12]:

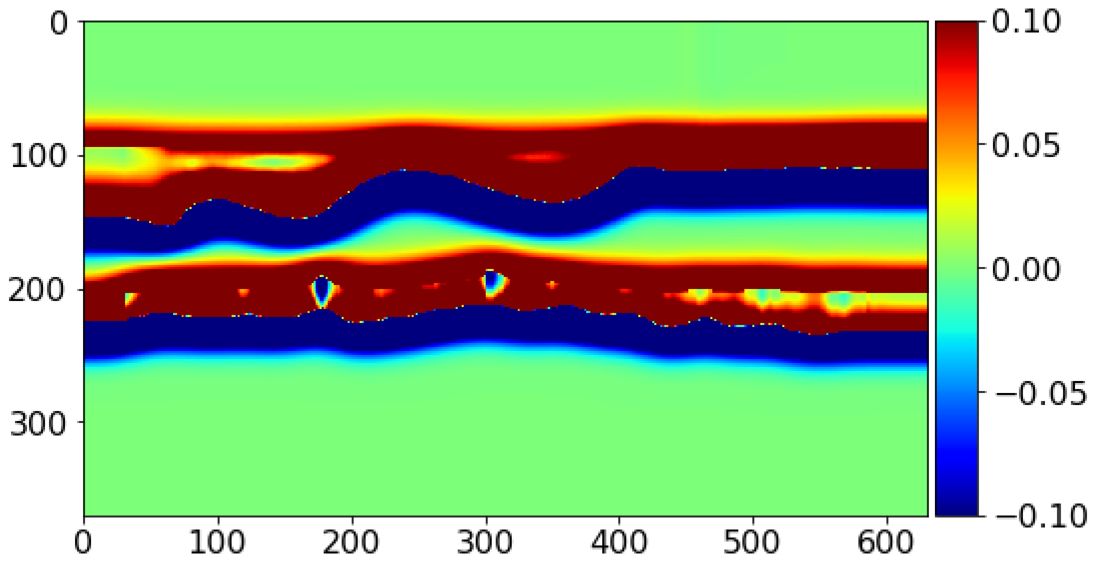
```
#NBVAL_IGNORE_OUTPUT

# Compute gradient of initial model
ff, update = fwi_gradient(model0.vp)
print('Objective value is %f ' % ff)
```

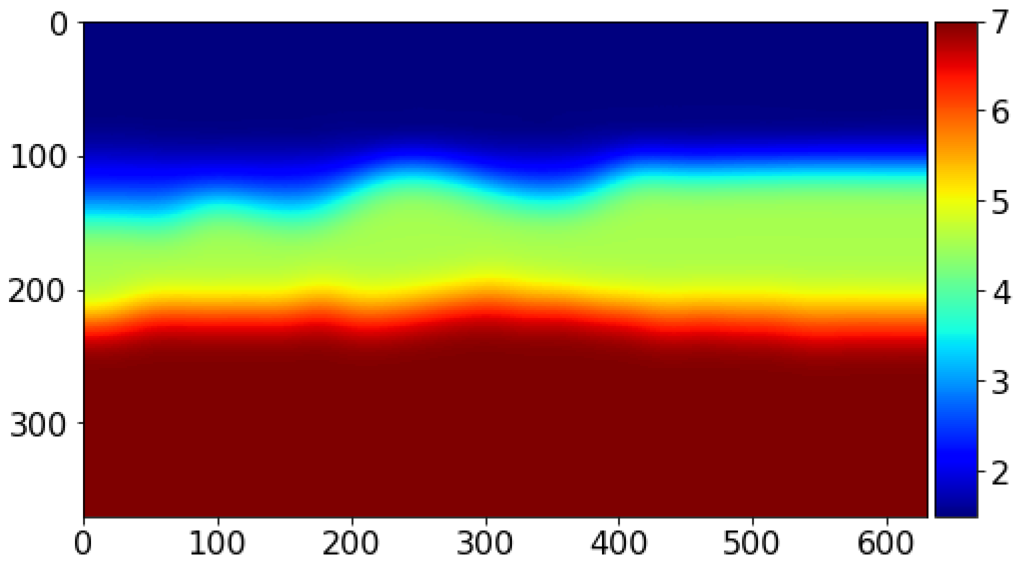
```
Objective value is 393144843.709052
```

In [13]:

```
#NBVAL_IGNORE_OUTPUT
from examples.seismic import plot_image

# Plot the FWI gradient
plot_image(update, vmin=-1e4, vmax=1e4, cmap="jet")

# Plot the difference between the true and initial model.
# This is not known in practice as only the initial model is provided.
plot_image(model0.vp.data - model.vp.data, vmin=-1e-1, vmax=1e-1, cmap="jet")

# Show what the update does to the model
alpha = 0.1 / np.abs(update).max()
plot_image(model0.vp.data - alpha*update, vmin=1.5, vmax=7.0, cmap="jet")


# We see that the gradient and the true perturbation have the same sign, therefo
re, with an appropriate scaling factor, we will update the model in the correct
 direction.
```


0
100
200
300
0
100
200
300
400
500
600
10000
5000
0
−5000
−10000


0
100
200
300
0
100
200
300
400
500
600
0.10
0.05
0.00
−0.05
−0.10



We see that the gradient and the true perturbation have the same sign, therefore, with an appropriate scaling factor, we will update the model in the correct direction.

In [14]:

```
# Define bounding box constraints on the solution.
def apply_box_constraint(vp):
    # Maximum possible 'realistic'
    # Minimum possible 'realistic'
    return np.clip(vp, 1.5, 7.0)
```

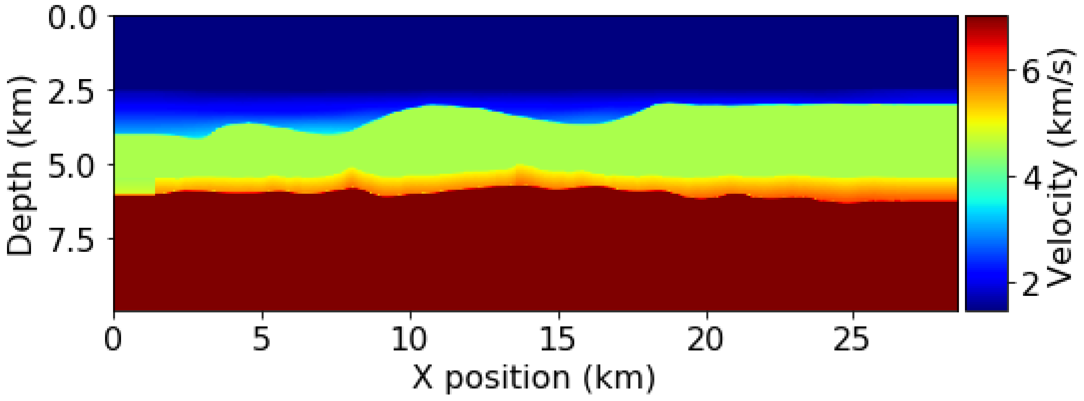
In [15]:

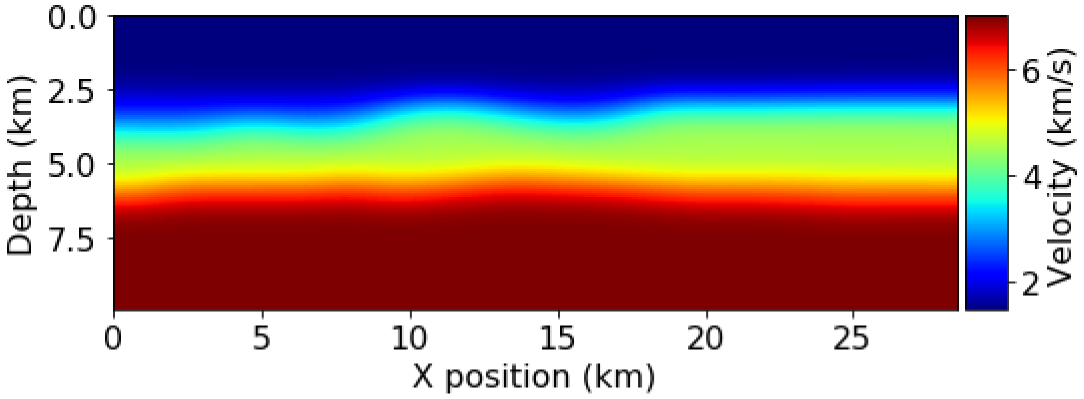
```
#NBVAL_SKIP

# Run FWI with gradient descent
history = np.zeros((fwi_iterations, 1))
for i in range(0, fwi_iterations):
    # Compute the functional value and gradient for the current
    # model estimate
    phi, direction = fwi_gradient(model0.vp)

    # Store the history of the functional values
    history[i] = phi

    # Artificial Step length for gradient descent
    # In practice this would be replaced by a Linesearch (Wolfe, ...)
    # that would guarantee functional decrease Phi(m-alpha g) <= epsilon Phi(m)
    # where epsilon is a minimum decrease constant
    alpha = 0.1 / np.abs(direction).max()

    # Update the model estimate and enforce minimum/maximum values
    model0.vp = apply_box_constraint(model0.vp.data - alpha * direction)

    # Log the progress made
    print('Objective value is %f at iteration %d' % (phi, i+1))
```

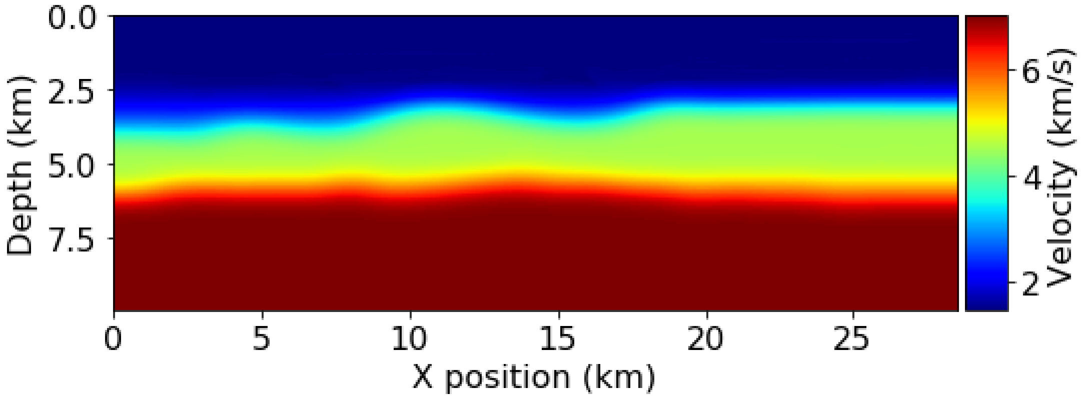
```
Objective value is 393144843.709052 at iteration 1
Objective value is 435498292.450234 at iteration 2
Objective value is 416899762.445659 at iteration 3
Objective value is 413769443.942276 at iteration 4
Objective value is 443010583.029289 at iteration 5
Objective value is 413864678.724855 at iteration 6
Objective value is 456822807.964417 at iteration 7
Objective value is 409588459.507076 at iteration 8
Objective value is 446786835.304688 at iteration 9
Objective value is 411947134.672552 at iteration 10
```

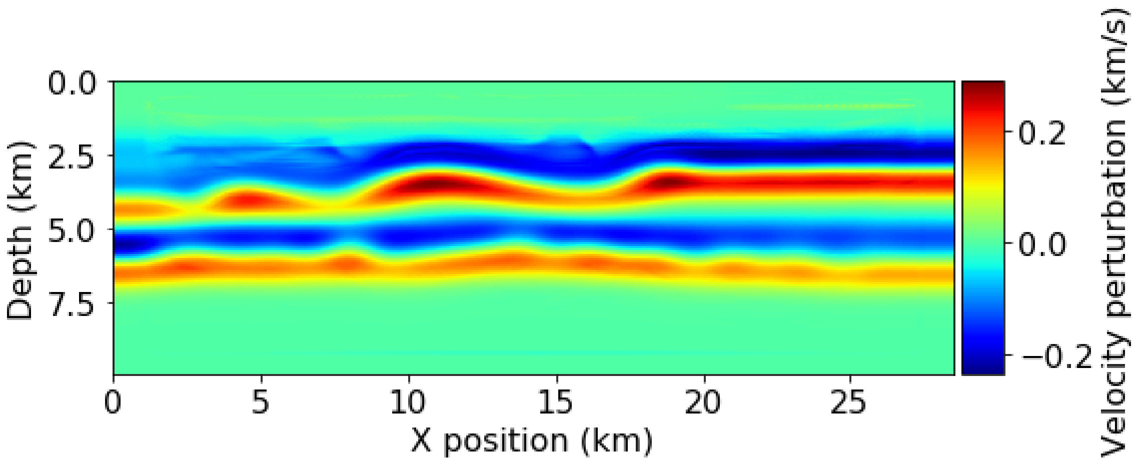
In [16]:

```
#NBVAL_IGNORE_OUTPUT

# Plot inverted velocity model
plot_velocity(model)
plot_velocity(model00)
plot_velocity(model0)
plot_perturbation(model0, model00)
```


Depth (km)
X position (km)
Velocity (km/s)

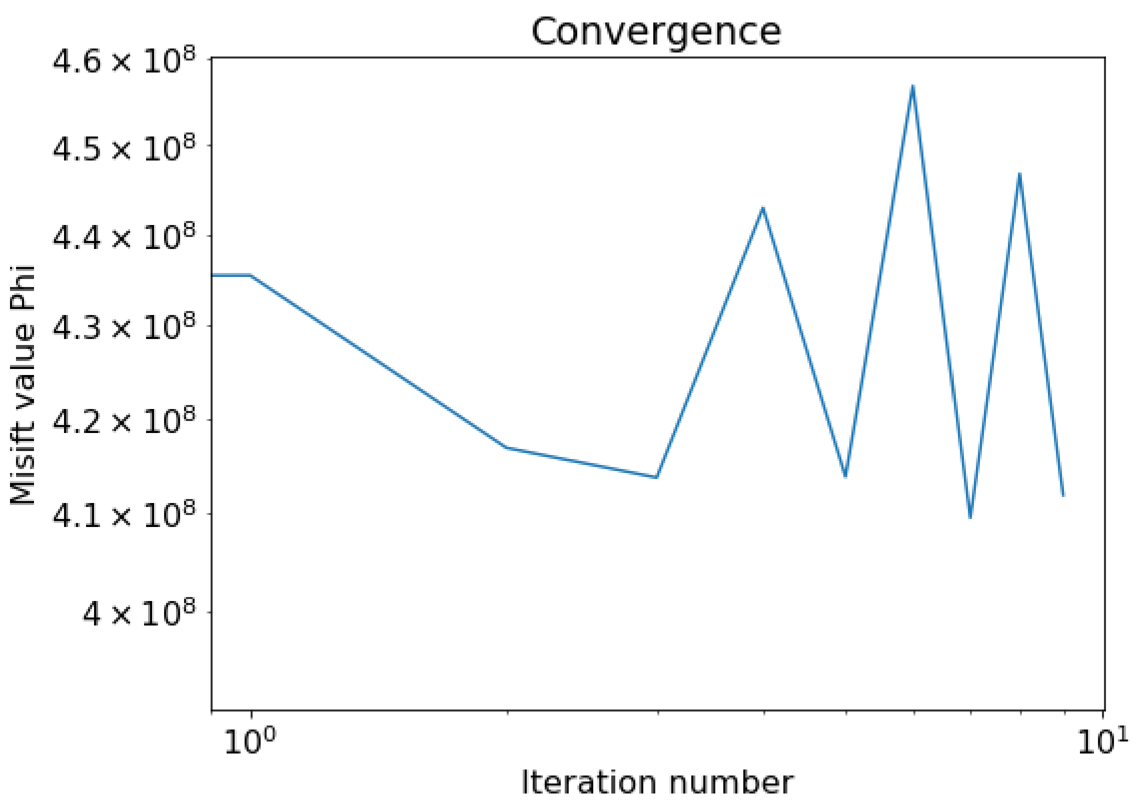

Depth (km)
X position (km)
Velocity (km/s)


Depth (km)
X position (km)
Velocity (km/s)


Depth (km)
X position (km)
Velocity perturbation (km/s)

In [17]:

```
#NBVAL_SKIP
import matplotlib.pyplot as plt

# Plot objective function decrease
plt.figure()
plt.loglog(history)
plt.xlabel('Iteration number')
plt.ylabel('Misift value Phi')
plt.title('Convergence')
plt.show()
```



References

[1] *Virieux, J. and Operto, S.: An overview of full-waveform inversion in exploration geophysics, GEOPHYSICS, 74, WCC1–WCC26, doi:10.1190/1.3238367, http://library.seg.org/doi/abs/10.1190/1.3238367 (http://library.seg.org/doi/abs/10.1190/1.3238367), 2009.*

[2] *Haber, E., Chung, M., and Herrmann, F. J.: An effective method for parameter estimation with PDE constraints with multiple right hand sides, SIAM Journal on Optimization, 22, http://dx.doi.org/10.1137/11081126X (http://dx.doi.org/10.1137/11081126X), 2012.*

This notebook is part of the tutorial "Optimised Symbolic Finite Difference Computation with Devito" presented at the Intel® HPC Developer Conference 2017.