PyFWI

Release 0.1.9

Amir Mardan

May 15, 2023

GETTING STARTED

PyFWI is an open source Python package to perform seismic modeling and full-waveform inversion (FWI) in elastic media. This package is implemented in time domain and coded using GPU programming (PyOpenCL) to accelerate the computation.

CHAPTER

ONE

INSTALLATION

PyFWI can be installed using `pip` as

```
(.venv) $ pip install PyFWI
```

on macOS or

```
(.venv) $ py -m pip install PyFWI
```

on Windows.

CHAPTER

TWO

SIMPLE GRADIENT COMPUTATION

In this section we see some applications of PyFWI. First, forward modeling is shown and then we estimate gradient of cost funtion with respect to V_P.

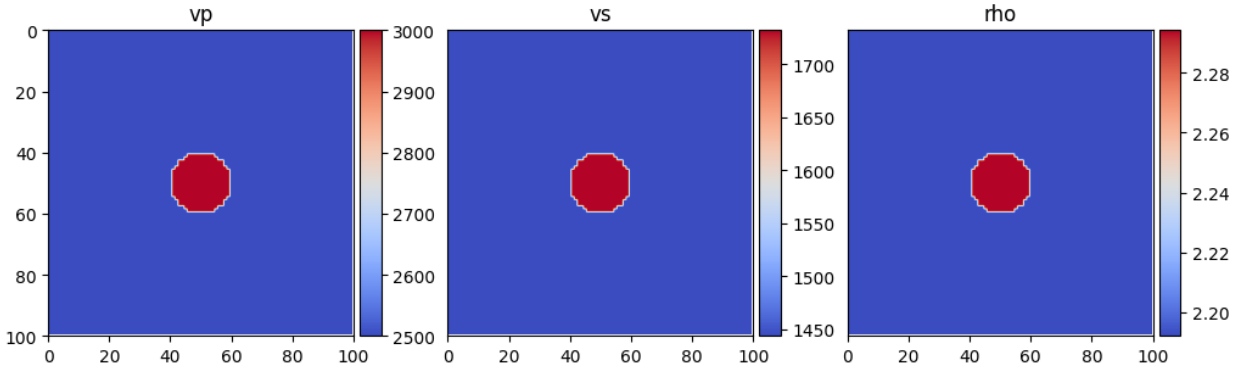
1. Forward modeling

In this simple example, we use PyFWI to do forward modeling. So, we need to first import the following packages amd modulus.

```
import matplotlib.pyplot as plt
import numpy as np

import PyFWI.wave_propagation as wave
import PyFWI.acquisition as acq
import PyFWI.seiplot as splt
import PyFWI.model_dataset as md
import PyFWI.fwi_tools as tools
import PyFWI.processing as process
```

A simple model can be created by using `model_dataset` module as

```
Model = md.ModelGenerator('louboutin')
model = Model()

im = splt.earth_model(model, cmap='coolwarm')
```



Then we need to create an input dictionary as follow

```
model_shape = model[[*model][0]].shape
```

(continues on next page)

(continued from previous page)

```
inpa = {
    'ns': 1,  # Number of sources
    'sdo': 4,  # Order of FD
    'fdom': 15,  # Central frequency of source
    'dh': 7,  # Spatial sampling rate
    'dt': 0.004,  # Temporal sampling rate
    'acq_type': 1,  # Type of acquisition (0: crosswell, 1: surface, 2: both)
    't': 0.8,  # Length of operation
    'npml': 20,  # Number of PML
    'pmlR': 1e-5,  # Coefficient for PML (No need to change)
    'pml_dir': 2,  # type of boundary layer
    'device': 1, # The device to run the program. Usually 0: CPU 1: GPU
}

seisout = 0 # Type of output 0: Pressure

inpa['rec_dis'] =  1 * inpa['dh']  # Define the receivers' distance
```

Now, we obtain the location of sources and receivers based on specified parameters.

```
offsetx = inpa['dh'] * model_shape[1]
depth = inpa['dh'] * model_shape[0]

src_loc, rec_loc = acq.surface_seismic(inpa['ns'], inpa['rec_dis'], offsetx,
                                                    inpa['dh'], inpa['sdo'])
src_loc[:, 1] -= 5 * inpa['dh']

# Create the source
src = acq.Source(src_loc, inpa['dh'], inpa['dt'])
src.Ricker(inpa['fdom'])
```

Finally, we can have the forward modelling as

```
# Create the wave object
W = wave.WavePropagator(inpa, src, rec_loc, model_shape, components=seisout, chpr=20)

# Call the forward modelling
d_obs = W.forward_modeling(model, show=False)  # show=True can show the propagation of␣
↪the wave
```

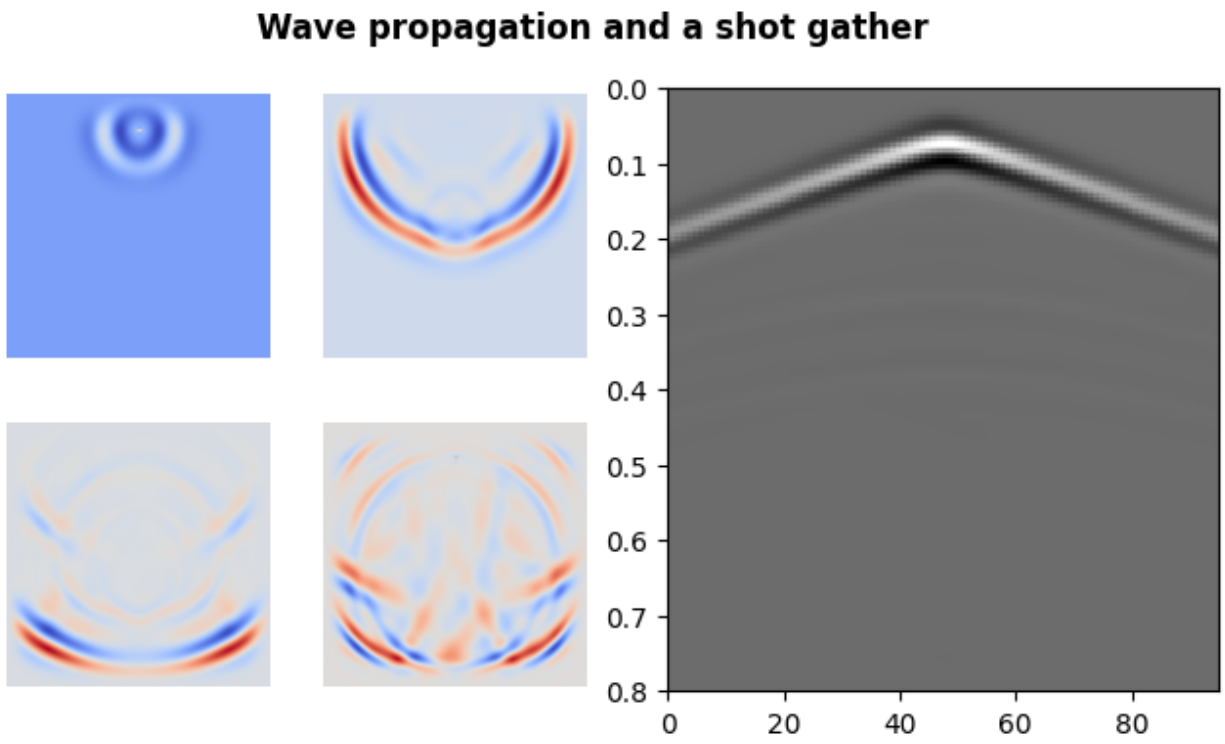
To compute the gradient using the adjoint-state method, we need to save the wavefield during the forward wave propagation. This must be done for the wavefield obtained from estimated model. For example, the wavefield at four time steps are presented here in addition to a shot gather.

```
fig = plt.figure(figsize=(8, 4))

count = 1

ax = fig.add_subplot(122)
ax = splt.seismic_section(ax, d_obs['taux'], t_axis=np.linspace(0, inpa['t'], int(1 +␣
↪inpa['t'] // inpa['dt'])))

ax_loc = [1, 2, 5, 6]
```

(continues on next page)

(continued from previous page)

```
snapshots = [40, 80, 130, 180]

for i in range(len(snapshots)):
    ax = fig.add_subplot(2, 4, ax_loc[i])
    ax.imshow(W.W['taux'][:, :, 0, snapshots[i]], cmap='coolwarm')

    ax.axis('off')
    count += 1
fig.suptitle("Wave propagation and a shot gather", fontweight='bold');
```



2. Gradient

To compute the gradient, we need the observed data and an initial model. So, first we obtain the observed data using more sources.

Note: For better visualization and avoiding crosstalk, I compute the gradient in acoustic media.

```
# Making medium acoustic
model['vs'] *= 0.0
model['rho'] = np.ones_like(model['rho'])

# Increasing number of sources
inpa['ns'] = 5

src_loc, rec_loc = acq.surface_seismic(inpa['ns'], inpa['rec_dis'], offsetx,
                                                    inpa['dh'], inpa['sdo'])
src_loc[:, 1] -= 5 * inpa['dh']

# Create the source
```

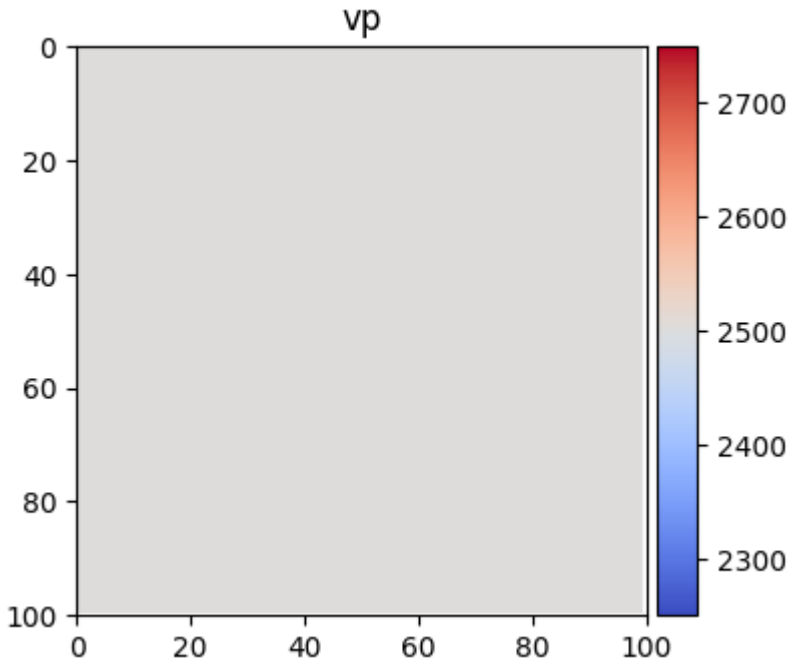
(continues on next page)

(continued from previous page)

```
src = acq.Source(src_loc, inpa['dh'], inpa['dt'])
src.Ricker(inpa['fdom'])

# Create the wave object
W = wave.WavePropagator(inpa, src, rec_loc, model_shape, components=seisout, chpr=20)

# Call the forward modelling
db_obs = W.forward_modeling(model, show=False)  # show=True can show the propagation of␣
↪the wave

# preparing data amd applying gain if required
db_obs = process.prepare_residual(db_obs, 1)
```

Then we create the initial model.

```
m0 = Model(smoothing=1)
m0['vs'] *= 0.0
m0['rho'] = np.ones_like(model['rho'])

im = splt.earth_model(m0, ['vp'], cmap='coolwarm')
```



And we simulate the wave propagation to obtain estimated data. For computing the gradient, we can smooth the gradient and scale it by defining `g_smooth` and `energy_balancing`.

```
inpa['energy_balancing'] = True
```

We save the wavefield at 20% of the time steps (`chpr = 20`) to be used for gradient calculation. The value of wavefield is accessible using the attribute `W` which is a dictionary for V_x, V_z, τ_x, τ_z, and τ_{xz} as `vx`, `vz`, `taux`, `tauz`, and `tauxz`. Each parameter is a 4D tensor. For example, we can have access to the last time step of τ_x for the first shot as `W.W['taux'][:, :, 0, -1]`.

```
Lam = wave.WavePropagator(inpa, src, rec_loc, model_shape,
                          chpr=20, components=seisout)

d_est = Lam.forward_modeling(m0, False)
d_est = process.prepare_residual(d_est, 1)
```

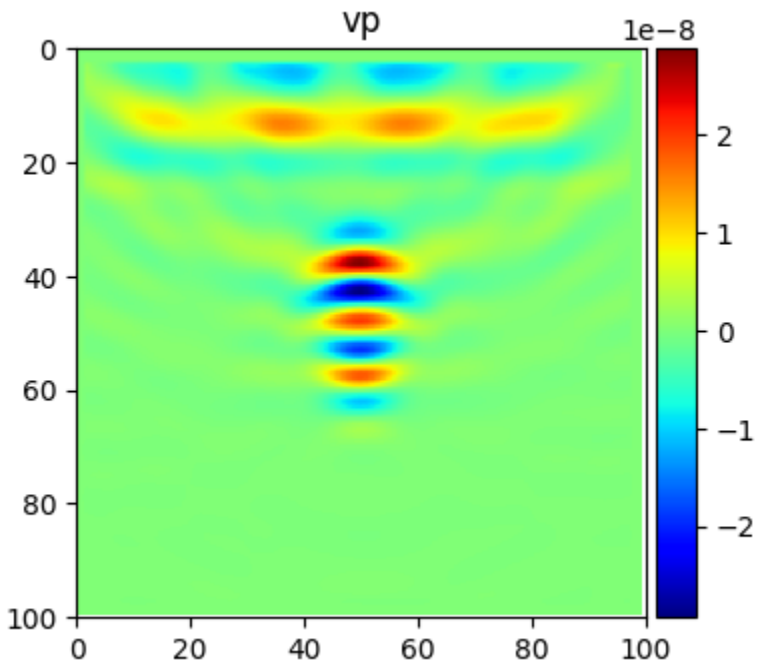
Now, we define the cost function and obtaine the residuals for adjoint-state method.

```
CF = tools.CostFunction('l2')
rms, adj_src = tools.cost_seismic(d_est, db_obs, fun=CF)
# print(rms)
```

Using the adjoint source, we can estimate the gradient as

```
grad = Lam.gradient(adj_src, show=False)
```

```
# Time to plot the results
splt.earth_model(grad, ['vp'], cmap='jet');
```



CHAPTER

THREE

SIMPLE FWI EXAMPLE

In this section we see application of PyFWI for performin FWI. First, forward modeling is shown and then we estimate a model of subsurface using FWI.

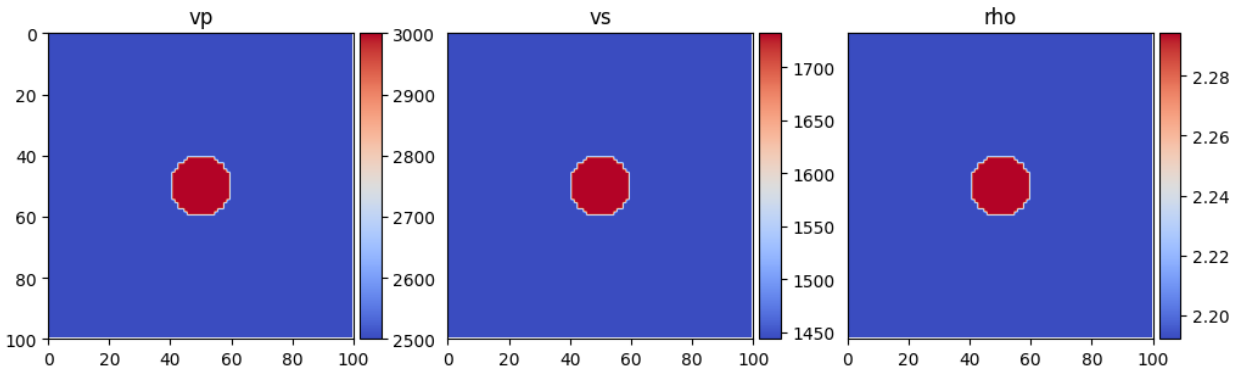
1. Forward modeling

In this simple example, we use PyFWI to do forward modeling. So, we need to first import the following packages amd modulus.

```
import matplotlib.pyplot as plt
import numpy as np

import PyFWI.wave_propagation as wave
import PyFWI.acquisition as acq
import PyFWI.seiplot as splt
import PyFWI.model_dataset as md
import PyFWI.fwi_tools as tools
import PyFWI.processing as process
from PyFWI.fwi import FWI
```

A simple model can be created by using `model_dataset` module as

```
Model = md.ModelGenerator('louboutin')
model = Model()

im = splt.earth_model(model, cmap='coolwarm')
```



Then we need to create an input dictionary as follow

```
model_shape = model[[*model][0]].shape

inpa = {
    'ns': 4,  # Number of sources
    'sdo': 4,  # Order of FD
    'fdom': 15,  # Central frequency of source
    'dh': 7,  # Spatial sampling rate
    'dt': 0.004,  # Temporal sampling rate
    'acq_type': 0,  # Type of acquisition (0: crosswell, 1: surface, 2: both)
    't': 0.6,  # Length of operation
    'npml': 20,  # Number of PML
    'pmlR': 1e-5,  # Coefficient for PML (No need to change)
    'pml_dir': 2,  # type of boundary layer
    'device': 1, # The device to run the program. Usually 0: CPU 1: GPU
}

seisout = 0 # Type of output 0: Pressure

inpa['rec_dis'] =  1 * inpa['dh']  # Define the receivers' distance
```

Now, we obtain the location of sources and receivers based on specified parameters.

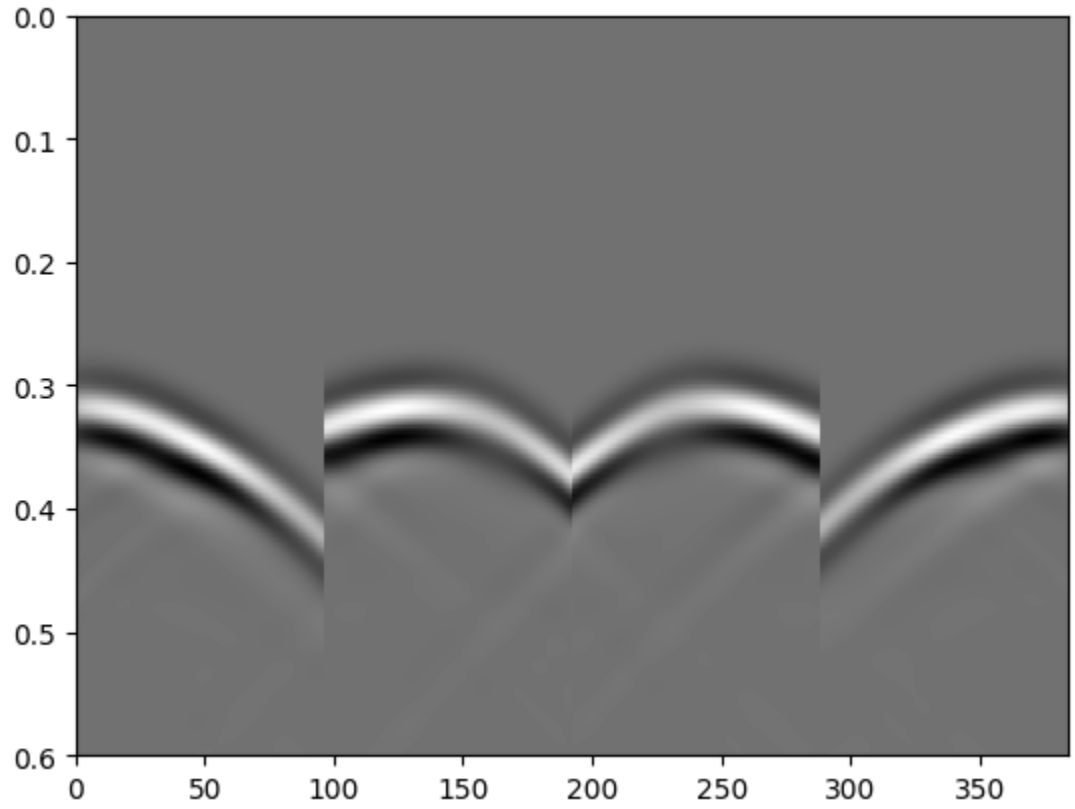
```
offsetx = inpa['dh'] * model_shape[1]
depth = inpa['dh'] * model_shape[0]

src_loc, rec_loc, n_surface_rec, n_well_rec = acq.acq_parameters(inpa['ns'],
                                                                 inpa['rec_dis'],
                                                                 offsetx,
                                                                 depth,
                                                                 inpa['dh'],
                                                                 inpa['sdo'],
                                                                 acq_type=inpa['acq_type
↪'])
# src_loc[:, 1] -= 5 * inpa['dh']

# Create the source
src = acq.Source(src_loc, inpa['dh'], inpa['dt'])
src.Ricker(inpa['fdom'])
```

Finally, we can have the forward modelling as

```
# Create the wave object
W = wave.WavePropagator(inpa, src, rec_loc, model_shape,
                        n_well_rec=n_well_rec,
                        components=seisout, chpr=0)

# Call the forward modelling
d_obs = W.forward_modeling(model, show=False)  # show=True can show the propagation of␣
↪the wave
```

```
plt.imshow(d_obs["taux"], cmap='gray',
           aspect="auto", extent=[0, d_obs["taux"].shape[1], inpa['t'], 0])
```

```
<matplotlib.image.AxesImage at 0x15144c760>
```



2. FWI

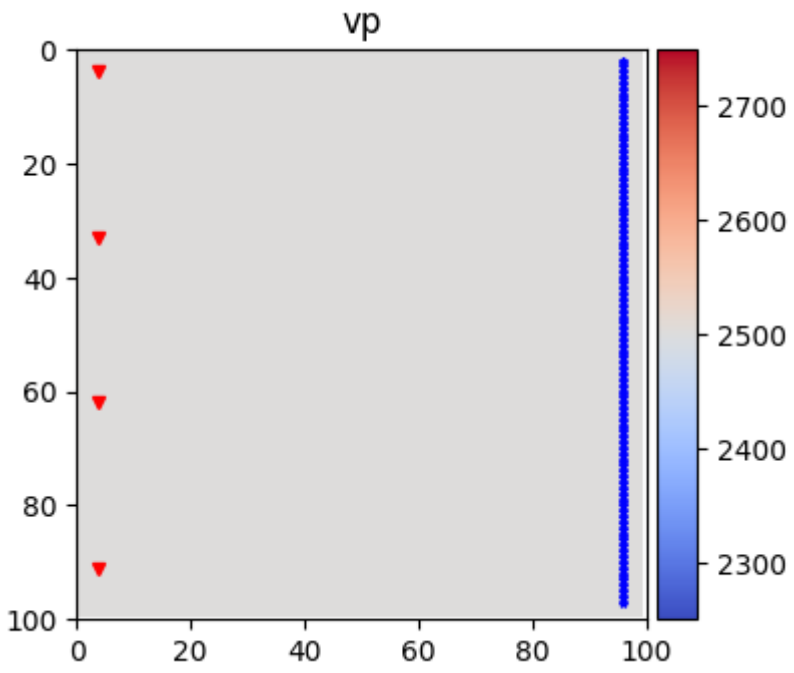
To perform FWI, we need the observed data and an initial model.

Note: For better visualization and avoiding crosstalk, I compute the gradient in acoustic media.

Here is a homogeneous initial model.

```
m0 = Model(smoothing=1)
m0['vs'] *= 0.0
m0['rho'] = np.ones_like(model['rho'])

fig = splt.earth_model(m0, ['vp'], cmap='coolwarm')

fig.axes[0].plot(src_loc[:,0]//inpa["dh"],
                 src_loc[:,1]//inpa["dh"], "rv", markersize=5)

fig.axes[0].plot(rec_loc[:,0]//inpa["dh"],
                 rec_loc[:,1]//inpa["dh"], "b*", markersize=3)
```

```
[<matplotlib.lines.Line2D at 0x158434ee0>]
```



Now, we can create a FWI object,

```
fwi = FWI(d_obs, inpa, src, rec_loc, model_shape,
          components=seisout, chpr=20, n_well_rec=n_well_rec)
```

and call it by providing the initial model `m0`, observed data `d_obs`, optimization method `method`, desired frequencies for inversion, number of iterations for each frequency, number of parameters for inversion `n_params`, index of the first parameter `k_0`, and index of the last parameter `k_end`. For example, if we have an elastic model, but we want to only invert for P-wave velocity, these parameters should be defined as

```
n_params = 1
k_0 = 1
k_end = 2
```

If we want to invert for P-wave velocity and then V_S, these parameters should be defined as

```
n_params = 1
k_0 = 1
k_end = 3
```

and for simultaneously inverting for these two parameters, we define these parameters as

```
n_params = 2
k_0 = 1
k_end = 3
```

```
m_est, _ = fwi(m0, method="lbfgs",
                   freqs=[25, 45], iter=[2, 2],
                   n_params=1, k_0=1, k_end=2)
```

```
Parameter number 1 to 1
2500.0 2500.0
 for f= 25: rms is: 0.0003187612455803901 with rms_reg: 0, and rms_data: 0.
↪0003187612455803901, rms_mp: 0.0, rms_model_relation: 0
Parameter number 1 to 1
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =        10000     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  3.18761D-04    |proj g|=  7.19684D-10

           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
10000      0      1      0     0     0   7.197D-10   3.188D-04
  F =   3.1876124558039010E-004

CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL
```

```
This problem is unconstrained.
```

```
2500.0 2500.0
 for f= 45: rms is: 0.004415073432028294 with rms_reg: 0, and rms_data: 0.
↪004415073432028294, rms_mp: 0.0, rms_model_relation: 0
RUNNING THE L-BFGS-B CODE

           * * *

Machine precision = 2.220D-16
 N =        10000     M =           10

At X0         0 variables are exactly at the bounds

At iterate    0    f=  4.41507D-03    |proj g|=  2.39370D-08


ITERATION     1
```

(continues on next page)

(continued from previous page)

```
---------------- CAUCHY entered-------------------
 There are            0   breakpoints

 GCP found in this segment
Piece      1 --f1, f2 at start point  -6.4549D-13  6.4549D-13
Distance to the stationary point =   1.0000D+00

---------------- exit CAUCHY----------------------

       10000  variables are free at GCP            1
```

```
This problem is unconstrained.
```

```
2499.9827111341883 2500.0297937900996
 for f= 45: rms is: 0.004414360038936138 with rms_reg: 0, and rms_data: 0.
↪004414360038936138, rms_mp: 0.0, rms_model_relation: 0
2499.9135556709416 2500.1489689504983
 for f= 45: rms is: 0.004411510191857815 with rms_reg: 0, and rms_data: 0.
↪004411510191857815, rms_mp: 0.0, rms_model_relation: 0
2499.636933817955 2500.6256695920933
 for f= 45: rms is: 0.004400133620947599 with rms_reg: 0, and rms_data: 0.
↪004400133620947599, rms_mp: 0.0, rms_model_relation: 0
2498.530446406008 2502.5324721584725
 for f= 45: rms is: 0.00435509392991662 with rms_reg: 0, and rms_data: 0.
↪00435509392991662, rms_mp: 0.0, rms_model_relation: 0
2494.1044967582216 2510.1596824239887
 for f= 45: rms is: 0.004182371310889721 with rms_reg: 0, and rms_data: 0.
↪004182371310889721, rms_mp: 0.0, rms_model_relation: 0
2476.400698167074 2540.668523486055
 for f= 45: rms is: 0.003607903141528368 with rms_reg: 0, and rms_data: 0.
↪003607903141528368, rms_mp: 0.0, rms_model_relation: 0
 LINE SEARCH           5  times; norm of step =    1365.0000000000000

At iterate    1    f=  3.60790D-03    |proj g|=  1.91296D-08


ITERATION     2

----------------SUBSM entered-----------------


----------------exit SUBSM --------------------

2356.1934399025804 2683.887118020453
 for f= 45: rms is: 0.0026031285524368286 with rms_reg: 0, and rms_data: 0.
↪0026031285524368286, rms_mp: 0.0, rms_model_relation: 0
 LINE SEARCH           0  times; norm of step =    3577.6768283163833

At iterate    2    f=  2.60313D-03    |proj g|=  1.26216D-08
```

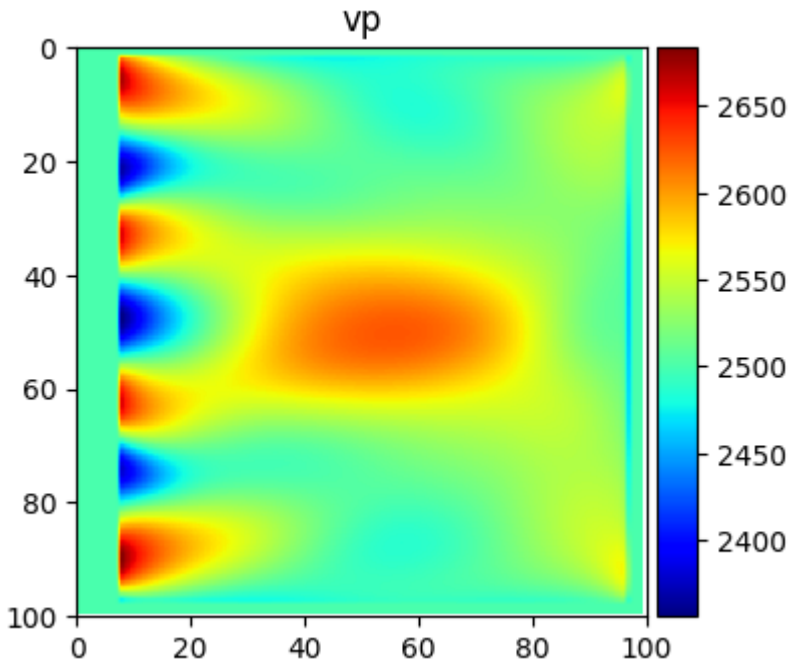
(continues on next page)

(continued from previous page)

```
           * * *

Tit   = total number of iterations
Tnf   = total number of function evaluations
Tnint = total number of segments explored during Cauchy searches
Skip  = number of BFGS updates skipped
Nact  = number of active bounds at final generalized Cauchy point
Projg = norm of the final projected gradient
F     = final function value

           * * *

   N    Tit     Tnf  Tnint  Skip  Nact     Projg        F
10000      2      8      1     0     0   1.262D-08   2.603D-03
  F =   2.6031285524368286E-003

STOP: TOTAL NO. of ITERATIONS REACHED LIMIT
```

Here is the estimated model

```
# Time to plot the results
fig = splt.earth_model(m_est, ['vp'], cmap='jet')
```



CHAPTER

FOUR

SYNTHETIC MODEL

CHAPTER

FIVE

ACQUISITION

class PyFWI.acquisition.**Source**(*src_loc*, *dh*, *dt*, *src_type=0*)

Bases: object

A class for defining different types of sources.

Parameters

- **src_loc** (*float32*) – location of sources.
- **dh** (*float*) – Spatial sampling rate.
- **dt** (*float*) – Temporal sampling rate
- **src_type** (*int, optional*) –

  Source type: 0: explosive
  1: directional x 2: directional z

Ricker(*fdom*)

Amethod to generate Ricker wavelet.

Parameters
fdom (*float32*) – Dominant frequency of wavelet

delta()

A method to generate spike.

Parameters
fdom (*float32*) – Dominant frequency of wavelet

PyFWI.acquisition.**acq_parameters**(*ns*, *rec_dis*, *offsetx*, *depth*, *dh*, *sdo*, *acq_type*)

A function to define the acquisition based on user's demand

Parameters
INPA (*dictionary*) –

A dictionnary containing required parameters for iversion, at least:

- ns: Number of sources
- rec_dis: Distance between receivers
- offsetx: Length of acquisition in x-direction
- depth: Depth of acquisition
- dh: spatial sampling rate
- sdo: Spatial differentiation order
- acq_type: Type of acquisition (0: crosswell, 1: surface, 2: both)

Returns

- **src_loc** (*float32*) – Location of sources
- **rec_loc** (*float32*) – Location of receivers
- **n_surface_rec** (*int*) – Number of receivers at the surface
- **n_well_rec** (*int*) – Number of receivers in wells at each side of the model

PyFWI.acquisition.**acquisition_plan**(*ns*, *nr*, *src_loc*, *rec_loc*, *acq_type*, *n_well_rec*, *dh*)

acquisition_plan generates the matrix of acquisition plan

[extended_summary]

Parameters

- **ns** (*[type]*) – [description]
- **nr** (*[type]*) – [description]
- **src_loc** (*[type]*) – [description]
- **rec_loc** (*[type]*) – [description]
- **acq_type** (*[type]*) – [description]
- **n_well_rec** (*[type]*) – [description]
- **dh** (*[type]*) – [description]

Returns

[description]

Return type

[type]

PyFWI.acquisition.**crosswell**(*ns*, *rec_dis*, *offsetx*, *depth*, *dh*, *sdo*)

A function to design a crosswell acquisition

Parameters

- **ns** (*int*) – Number of sources
- **rec_dis** (*float32*) – Distance between receivers
- **offsetx** (*flloat32*) – Length of survey in x-direction
- **depth** (*float32*) – Depth of survey
- **dh** (*float32*) – Sampling rate
- **sdo** (*{2, 4, 8}*) – Spatial order of finite difference method

Returns

- **src_loc** (*float32*) – Location of sources
- **rec_loc** (*float32*) – Location of receivers

PyFWI.acquisition.**discretized_acquisition_plan**(*data_guide*, *dh*, *npml=0*)

discretized_acquisition_plan discretizes the matrix of acquisition plan

[extended_summary]

Parameters

- **data_guide** (*[type]*) – [description]

- **dh** (*[type]*) – [description]
- **npml** (*int, optional*) – [description]. Defaults to 0.

Returns
[description]

Return type
[type]

PyFWI.acquisition.**surface_seismic**(*ns*, *rec_dis*, *offsetx*, *dh*, *sdo*)

A function to design a surface seismic acquisition

Parameters

- **ns** (*int*) – Number of sources
- **rec_dis** (*float32*) – Distance between receivers
- **offsetx** (*flloat32*) – Length of survey in x-direction
- **depth** (*float32*) – Depth of survey
- **dh** (*float32*) – Spatial sampling rate
- **sdo** (*{2, 4, 8}*) – Spatial order of finite difference method

Returns

- **src_loc** (*float32*) – Location of sources
- **rec_loc** (*float32*) – Location of receivers

CHAPTER

SIX

WAVE PROPAGATION

class PyFWI.wave_propagation.**WavePropagator**(*inpa*, *src*, *rec_loc*, *model_shape*, *components=0*, *n_well_rec=0*, *chpr=0*, *set_env_variable=True*)

wave_propagator is a class to handle the forward modeling and gradient calculation.

[extended_summary]

Parameters

- **inpa** (*dict*) – A dictionary including most of the required inputs
- **src** (*class*) – Source object
- **rec_loc** (*ndarray*) – Location of the receivers
- **model_shape** (*tuple*) – Shape of the model
- **n_well_rec** (*int*) – Number of receivers in the well
- **chpr** (*percentage*) – Checkpoint ratio in percentage
- **component** – Seismic output

WavePropagator.**forward_modeling**(*model0*, *show=False*)

forward_modeling performs the forward modeling.

Parameters

- **model0** (*dict*) – The earth model
- **show** (*bool, optional*) – True if you desire to see the propagation of the wave, by default False

Returns

Seismic section

Return type

dict

CHAPTER

SEVEN

GRADIENT ESTIMATION

WavePropagator.**gradient**(*res*, *show=False*, *parameterization='dv'*)

gradient estimates the gradient using adjoint-state method.

Parameters

- **res** (*dict*) – The adjoint of the derivative of ost function with respect to wavefield
- **show** (*bool, optional*) – True if you desire to see the backward wave propagation, by default False
- **parameterization** (*str, optional*) – Specify the parameterization for output, by default 'dv'

Returns

Gradient

Return type

dict

CHAPTER

EIGHT

FULL-WAVEFORM INVERSION

class PyFWI.fwi.**FWI**(*d_obs*, *inpa*, *src*, *rec_loc*, *model_shape*, *components*, *chpr*, *n_well_rec=0*, *param_functions=None*)

FWI perform full-waveform inversion

Parameters

- **d_obs** (*dict*) – Observed data
- **inpa** (*dict*) – Input parameters
- **src** (*Class*) – Source object
- **rec_loc** (*float*) – Receiver location
- **model_size** (*tuple*) – Shape of the model
- **n_well_rec** (*int*) – Number of receivers in the well
- **chpr** (*float (percentage)*) – Percentage for check point
- **components** (*int*) – Type of output
- **param_functions** (*dict, optional*) – List of functions required in case of inversion with different parameterization than dv, by default None

__call__(*m0*, *method: str*, *iter: List[int]*, *freqs: List[float]*, *n_params*, *k_0*, *k_end*)

Calling this object performs the FWI

Parameters

- **m0** (*dict*) – The initial model
- **method** (*str*) – The optimization method. Either should be *cg* for congugate gradient or *lbfgs* for l-BFGS.
- **iter** (*List[int]*) – An array of iteration for each frequency
- **freqs** (*List[float]*) – Frequencies for multi-scale inversion.
- **n_params** (*int*) – Number of parameter to invert for in each time
- **k_0** (*int*) – The first parameter of interest
- **k_end** (*int*) – The last parameter of interest

Returns

- **m_est** (*dictionary*) – The estimated model
- **rms** (*ndarray*) – The rms error

CHAPTER

NINE

TIME-LAPSE FULL-WAVEFORM INVERSION

class PyFWI.tl_fwi.**TimeLapse**(*b_dobs*, *m_dobs*, *inpa*, *src*, *rec_loc*, *model_shape*, *components*, *chpr*, *n_well_rec=0*, *param_functions=None*)

TimeLapse is a class to perform time-lapse FWI.

Parameters

- **b_dobs** (*dict*) – Observed data from baseline
- **m_dobs** (*dict*) – Observed data from monitor vintage
- **inpa** (*dict*) – Dictionary of required parameters for FWI
- **src** (*obj*) – Source function
- **rec_loc** (*ndarray*) – Location of receivers
- **model_size** (*tuple*) – Shape pf model
- **n_well_rec** (*int*) – Number of receivers on the surface
- **chpr** (*int*) – Checkpoint percentage
- **components** (*int*) – Code for components to be recorded
- **param_functions** (*dict, optional*) – Required functions to switch the gradient

__call__(*b_m0*, *iter*, *freqs*, *tl_method*, *n_params*, *k_0*, *k_end*)

Calling this class will run time-lapse FWI and return the result of FWI and TL-FWI

Parameters

- **b_m0** (*dict*) – Initial model in form of dictionary
- **iter** (*list*) – List of iteration for each frequency
- **freqs** (*list*) – Frequencies for multiscale inversion
- **tl_method** (*str*) –

  Name of time-lapse method ('cc': Cascaded, 'sim': Simultaneous, 'wa': Weighted average,
  'cj': Cascaded joint, 'cd': Central difference, 'cu': Cross updating)
- **n_params** (*int*) – Number of parameters to invert
- **k_0** (*int*) – Index of th first parameter to invert (considering a b_m0 with three parameters (vp, vs, rho), if we can set k_0 as 2 to start the inversion for vs.)
- **k_end** (*int*) – Index of th first parameter to invert (considering a b_m0 with three parameters (vp, vs, rho), if we can set k_end as 2 to doesn't invert rho.)

Returns

- **m** (*dict*) – A dictionary containing the result of FWI and TL-FWI
- **rms** (*adarray*) – rms error

CHAPTER

TEN

REGULARIZATION

class PyFWI.fwi_tools.**Regularization**(*nx*, *nz*, *dx*, *dz*)

Regularization Prepares tools for regularizing FWI problem

Parameters

- **nx** (*int scalar*) – Number of samples in x-direction
- **nz** (*int scalar*) – Number of samples in z-direction
- **dx** (*float scalar*) – Spatial sampling rate in x-direction
- **dz** (*float scalar*) – Spatial sampling rate in z-direction

10.1 Total variation (TV)

Regularization.**tv**(*x0*, *eps*, *alpha_z*, *alpha_x*)

Parameters

- **x0** (*float*) – Data
- **eps** (*scalar float*) – small value for make it deffrintiable at zero
- **alpha_z** (*scalar float*) – coefficient of Dz
- **alpha_x** (*scalar float*) – coefficient of Dx

Returns

- **rms** (*scalar float*) – loss
- **grad** (*scalar float*) – Gradient of loss w.r.t. model parameters

10.2 Tikhonov

Regularization.**tikhonov**(*x0*, *alpha_z*, *alpha_x*)

A method to implement Tikhonov regularization with order of 2

Parameters

- **x0** (*1D ndarray*) – Data
- **alpha_z** (*float*) – coefficient of Dz
- **alpha_x** (*float*) – coefficient of Dx

Returns

- **rms** (*scalar float*) – loss
- **grad** (*scalar float*) – Gradient of loss w.r.t. model parameters

10.3 Parameter relation

Regularization.**parameter_relation**(*m0*, *models*, *k0*, *kend*, *freq*)

parameter_relation considers regularization for the relation between parameters.

Parameters

- **m0** (*ndarray*) – Vector of parameters
- **models** (*dict*) – A dictionary containing couple of dictionaries which includes a numpy polyfit model and regularization parameter.
- **k0** (*int*) – Index of the first parameter in m0
- **kend** (*int*) – Index of the last parameter in m0

Returns

- **rms** (*float*) – rms of regularization
- **grad** (*ndarray*) – Vector of gradient od the regularization

10.4 Prior information

Regularization.**priori_regularization**(*m0*, *regularization_dict*, *k0*, *kend*, *freq*)

priori_regularization consider the priori information regularization.

Parameters

- **m0** (*float*) – Vector of parameters
- **regularization_dict** (*dict*) – A dictionary containing couple of priori model and regularization hyperparameter
- **k0** (*int*) – Index of the first parameter in m0
- **kend** (*int*) – Index of the last parameter in m0

Returns

- **rms** (*float*) – rms of regularization
- **grad** (*ndarray*) – Vector of gradient od the regularization

References

Asnaashari et al., 2013, Regularized seismic full waveform inversion with prior model information, Geophysics, 78(2), R25-R36, eq. 5.

CHAPTER

ELEVEN

GRADIENT SWITCHING

11.1 lmd to vd

grad_switcher.**grad_lmd_to_vd**(*gmu*, *grho*, *lam*, *mu*, *rho*)

grad_lmr_to_vd switch the gradient.

This function switch the gradient from [lambda, mu, rho] (LMD) to [vp, vs, rho] (DV).

Parameters

- **glam** (*float*) – Gradient w.r.t. lambda
- **gmu** – Gradient w.r.t. mu
- **grho** (*float*) – Gradient w.r.t. density
- **lam** (*float*) – Gradient w.r.t. lambda
- **mu** (*float*) – Gradient w.r.t. mu
- **rho** (*float*) – Gradient w.r.t. density

Returns

- **gvp** (*float*) – Gradient of cost function w.r.t. Vp
- **gvs** (*float*) – Gradient of cost function w.r.t. Vs
- **grho** (*float*) – Gradient of cost function w.r.t. density

11.2 vd to lmd

grad_switcher.**grad_vd_to_lmd**(*gvs*, *grho*, *vp*, *vs*, *rho*)

grad_vd_to_lmd grad_vd_to_lmd transder the gradient from DV to LMD

Parameters

- **gvp** (*float*) – Gradient of cost function w.r.t. Vp
- **gvs** (*float*) – Gradient of cost function w.r.t. Vs
- **grho** (*float*) – Gradient of cost function w.r.t. density
- **vp** (*float*) – P-wave velocity
- **vs** (*float*) – S-wave velocity
- **rho** (*float*) – Density

Returns

- **glam** (*float*) – Gradient of cost function w.r.t. first Lamé parameter (lambda)
- **gmu** (*float*) – Gradient of cost function w.r.t. second Lamé parameter (mu)
- **grho** (*float*) – Gradient of cost function w.r.t. density

CHAPTER

TWELVE

COST FUNCTION

class PyFWI.fwi_tools.**CostFunction**(*cost_function_type='l2'*)

CostFunction provides different cost functions.

Parameters

cost_function_type (*str, optional*) – Type of cost function, by default "l2"

__call__(*dest*, *dobs*)

By calling a CostFunction object, the loss is calculated.

Parameters

- **dest** (*dict*) – Estimated data
- **dobs** (*dict*) – Observed data

Returns

- **err** (*scalar float*) – Error
- **adj_src** (*dict*) – A dictionary containing adjoint of the residuals

CHAPTER

THIRTEEN

VISUALIZATION

13.1 Earth model

seiplot.**earth_model**(*keys=[]*, *offset=None*, *depth=None*, ***kwargs*)

earth_model show the earth model.

This function is provided to show the earth models.

Parameters

- **model** (*Dictionary*) – A dictionary containing the earth model.
- **keys** (*list, optional*) – List of parameters you want to show. Defaults to [].

Returns

The figure class to which the images are added for furthur settings like im.set-clim().

Return type

fig (class)

13.2 Seismic Section

seiplot.**seismic_section**(*data*, *x_axis=None*, *t_axis=None*, *aspect_preserving=False*, ***kargs*)

seismic_section show seismic section

Parameters

- **ax** (*Axes or array of Axes*) – ax to arrange plots
- **data** (*float*) – Seismic section
- **x_axis** (*array*) – X-axis, by default None
- **t_axis** (*array, optional*) – t-axis, by default None
- **aspect_preserving** (*bool, optional*) – _description_, by default False

Returns

ax – The ax for more adjustment

Return type

Axis

CHAPTER

FOURTEEN

ROCK PHYSICS

14.1 Han model

PyFWI.rock_physics.**Han**(*phi*, *cc*, *a1=5.5*, *a2=6.9*, *a3=2.2*, *b1=3.4*, *b2=4.7*, *b3=1.8*)

Han estimates velocity based on porosity and clasy content

Han found empirical regressions relating ultrasonic (laboratory) velocities to porosity and clay content

Parameters

- **phi** (*[type]*) – [porosity
- **cc** (*[type]*) – clay content
- **a1** (*float, optional*) – Constant value for Vp. Defaults to 5.77.
- **a2** (*float, optional*) – Constant value for Vp. Defaults to 6.94.
- **a3** (*float, optional*) – Constant value for Vp. Defaults to 1.728.
- **b1** (*float, optional*) – Constant value for Vs. Defaults to 5.77.
- **b2** (*float, optional*) – Constant value for Vs. Defaults to 6.94.
- **b3** (*float, optional*) – Constant value for Vs. Defaults to 1.728.

Returns

P-wave velocity (km/s) vs = S-wave velocity (km/s)

Return type

vp

References

1. Hu et al, 2021, Direct updating of rock-physics properties using elastice full-waveform inversion
2. Mavko, G., Mukerji, T., & Dvorkin, J., 2020, The rock physics handbook. Cambridge university press.

14.2 Drained *K* and

PyFWI.rock_physics.**drained_moduli**(*phi*, *k_s*, *g_s*, *cs*)

drained_moduli computes the effective mechanical moduli KD and GD

[extended_summary]

Parameters

- **phi** (*float*) – Porosity
- **k_s** (*float*) – Solid bulk modulus
- **g_s** (*float*) – Solid shear modulus
- **cs** (*float*) – general consolidation parameter

Returns

- **k_d** (*float*) – Effective drained bulk modulus
- **g_d** (*float*) – Effective drained shear modulus

References

Dupuy et al., 2016, Estimation of rock physics properties from seismic attributes — Part 1: Strategy and sensitivity analysis, Geophysics

14.3 Voigt Berie

14.4 Biot-Gassmann model

14.5 Weighted averaging

PyFWI.rock_physics.**weighted_average**(*prop1*, *prop2*, *volume1*)

weighted_average is a function to calculate the wighted average of properties

Parameters

- **prop1** (*float*) – Property of the material 1
- **prop2** (*float*) – Property of the material 2
- **volume1** (*float*) – Value specifying the rational volume of the material 1

Returns
mixed – Property of mixed material

Return type
float

14.6 Switch lmd to vd

PyFWI.rock_physics.**lmd2vd**(*lam*, *mu*, *rho*)

lmd2vd switches Lama modulus and density to vp, vs, density

[extended_summary]

Parameters

- **lam** (*[type]*) – [description]
- **mu** (*[type]*) – [description]
- **rho** (*[type]*) – [description]

Returns
[description]

Return type
[type]

14.7 Switch vd to lmd

PyFWI.rock_physics.**vd2lmd**(*vp*, *vs*, *rho*)

vd2lmd switches vp, vs, density to Lame modulus and density to

[extended_summary]

Parameters

- **vp** (*[type]*) – [description]
- **vs** (*[type]*) – [description]
- **rho** (*[type]*) – [description]

Returns
[description]

Return type
[type]

CHAPTER

FIFTEEN

PYFWI WITHIN PYTORCH

PyFWI is integrated with PyTorch to provide the gradient of a cost function with respect to model parameters. Here, a simple example is provided to show the application of this integration. This notebook can be compared to the Simple Example for a better understanding of the required changes and the results.

In this section, we first show the forward modeling, and then we estimate the gradient of the cost function with respect to V_P.

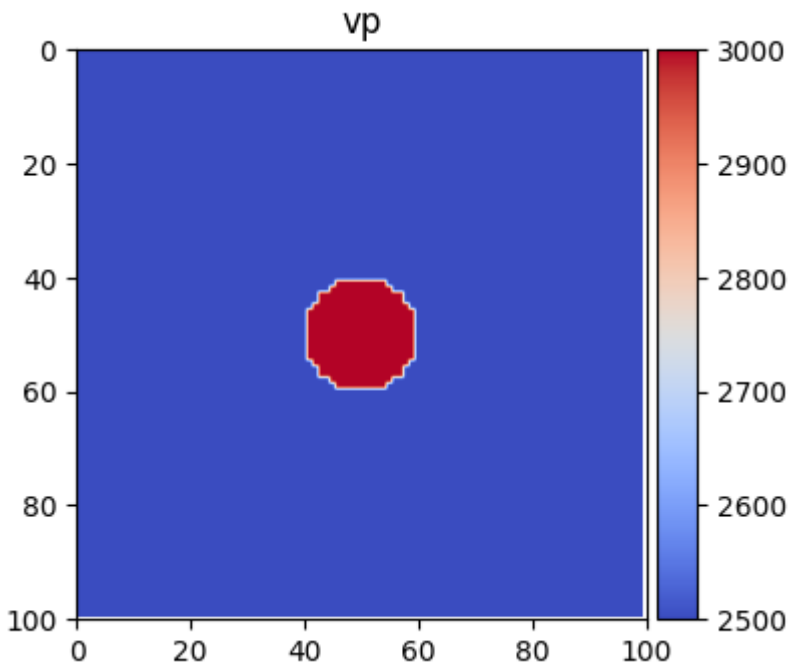
1. Forward modeling

In this simple example, we use PyFWI to do forward modeling. So, we need to first import the following packages and modulus.

```
import matplotlib.pyplot as plt
import numpy as np

import PyFWI.wave_propagation as wave
import PyFWI.acquisition as acq
import PyFWI.seiplot as splt
import PyFWI.model_dataset as md
import PyFWI.fwi_tools as tools
import PyFWI.processing as process

import torch
from PyFWI.torchfwi import Fwi
```

A simple model can be created by using `model_dataset` module as

```
Model = md.ModelGenerator('louboutin')
model = Model()
# Making medium acoustic
model['vs'] *= 0.0
model['rho'] = np.ones_like(model['rho'])

im = splt.earth_model(model, ['vp'], cmap='coolwarm')
```



Then we need to create an input dictionary as follow

```
model_shape = model[[*model][0]].shape

inpa = {
    'ns': 5,  # Number of sources
    'sdo': 4,  # Order of FD
    'fdom': 15,  # Central frequency of source
    'dh': 7,  # Spatial sampling rate
    'dt': 0.004,  # Temporal sampling rate
    'acq_type': 1,  # Type of acquisition (0: crosswell, 1: surface, 2: both)
    't': 0.8,  # Length of operation
    'npml': 20,  # Number of PML
    'pmlR': 1e-5,  # Coefficient for PML (No need to change)
    'pml_dir': 2,  # type of boundary layer
    'device': 1, # The device to run the program. Usually 0: CPU 1: GPU
    'seimogram_shape': '3d',
}

seisout = 0 # Type of output 0: Pressure

inpa['rec_dis'] =  1 * inpa['dh']  # Define the receivers' distance
```

Now, we obtain the location of sources and receivers based on specified parameters.

```
offsetx = inpa['dh'] * model_shape[1]
depth = inpa['dh'] * model_shape[0]

src_loc, rec_loc = acq.surface_seismic(inpa['ns'], inpa['rec_dis'], offsetx,
                                                    inpa['dh'], inpa['sdo'])
src_loc[:, 1] -= 5 * inpa['dh']
```

(continues on next page)

(continued from previous page)

```
# Create the source
src = acq.Source(src_loc, inpa['dh'], inpa['dt'])
src.Ricker(inpa['fdom'])
```

Model properties should be with type of `torch.tensor`. So, we need to convert these properties.

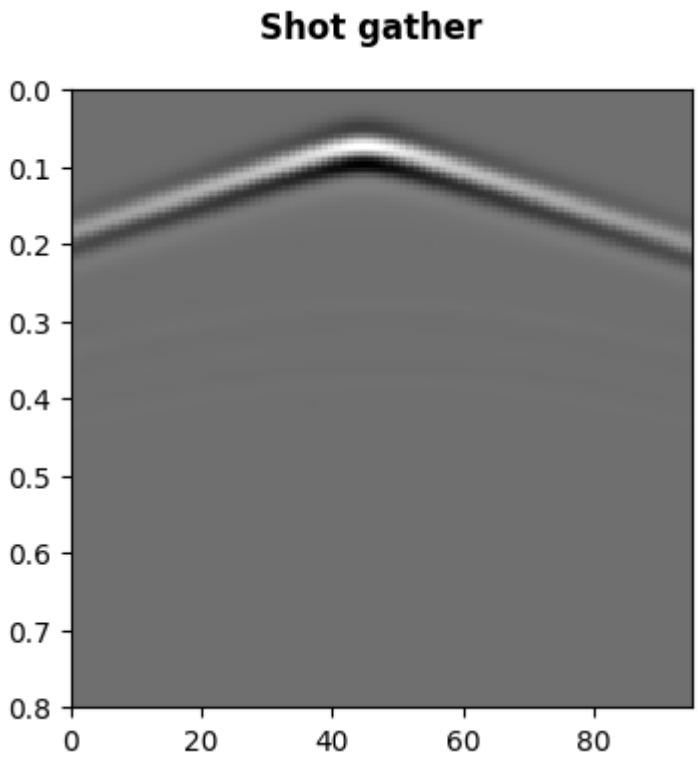
```
vp = torch.tensor(model['vp'])
vs = torch.tensor(model['vs'])
rho = torch.tensor(model['rho'])
```

Finally, we can have the forward modelling as

```
# Create the wave object
W = wave.WavePropagator(inpa, src, rec_loc, model_shape, components=seisout)

# Call the forward modelling
taux_obs, tauz_obs = Fwi.apply(W, vp, vs, rho)  # show=True can show the propagation of␣
↪the wave
```

To compute the gradient using the adjoint-state method, we need to save the wavefield during the forward wave propagation. This must be done for the wavefield obtained from estimated model. For example, the wavefield at four time steps are presented here in addition to a shot gather.

```
fig = plt.figure(figsize=(4, 4))

ax = fig.add_subplot(111)
ax = splt.seismic_section(ax, taux_obs[..., 2], t_axis=np.linspace(0, inpa['t'], int(1 +␣
↪inpa['t'] // inpa['dt'])))

fig.suptitle("Shot gather", fontweight='bold');
```



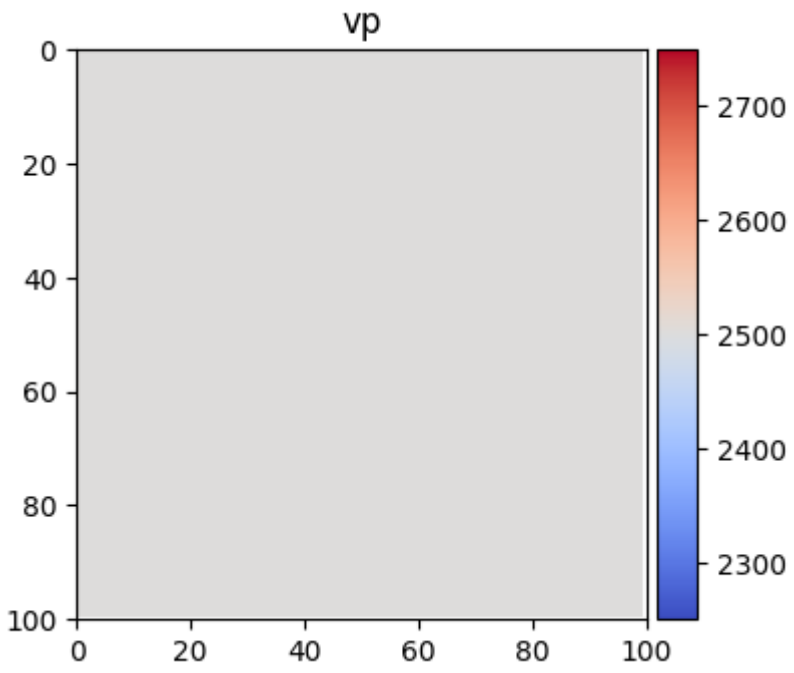
2. Gradient

To compute the gradient, we need the observed data and an initial model.

Note: For better visualization and avoiding crosstalk, I compute the gradient in acoustic media.

Then we create the initial model.

```
m0 = Model(smoothing=1)
m0['vs'] *= 0.0
m0['rho'] = np.ones_like(model['rho'])

# Convert to tensor
vp0 = torch.tensor(m0['vp'], requires_grad=True)
vs0 = torch.tensor(m0['vs'], requires_grad=True)
rho0 = torch.tensor(m0['rho'], requires_grad=True)

im = splt.earth_model(m0, ['vp'], cmap='coolwarm')
```



And we simulate the wave propagation to obtain estimated data. For computing the gradient, we can smooth the gradient and scale it by defining `g_smooth` and `energy_balancing`.

```
inpa['energy_balancing'] = True
```

We save the wavefield at 20% of the time steps (`chpr = 20`) to be used for gradient calculation. The value of wavefield is accessible using the attribute `W` which is a dictionary for V_x, V_z, τ_x, τ_z, and τ_{xz} as `vx`, `vz`, `taux`, `tauz`, and `tauxz`. Each parameter is a 4D tensor. For example, we can have access to the last time step of τ_x for the first shot as `W.W['taux'][:, :, 0, -1]`.

```
Lam = wave.WavePropagator(inpa, src, rec_loc, model_shape,
                          chpr=20, components=seisout)

taux_est, tauz_est = Fwi.apply(Lam,
                               vp0,
                               vs0,
                               rho0
                               )
```

Now, we define the cost function and obtaine the residuals for adjoint-state method.

```
criteria = torch.nn.MSELoss(reduction='sum')
mse0 = 0.5 * criteria(taux_est, taux_obs)
mse1 = 0.5 * criteria(tauz_est, taux_obs)
mse = mse0 + mse1
# print(mse.item())
mse.backward()
```

Using the adjoint source, we can estimate the gradient as

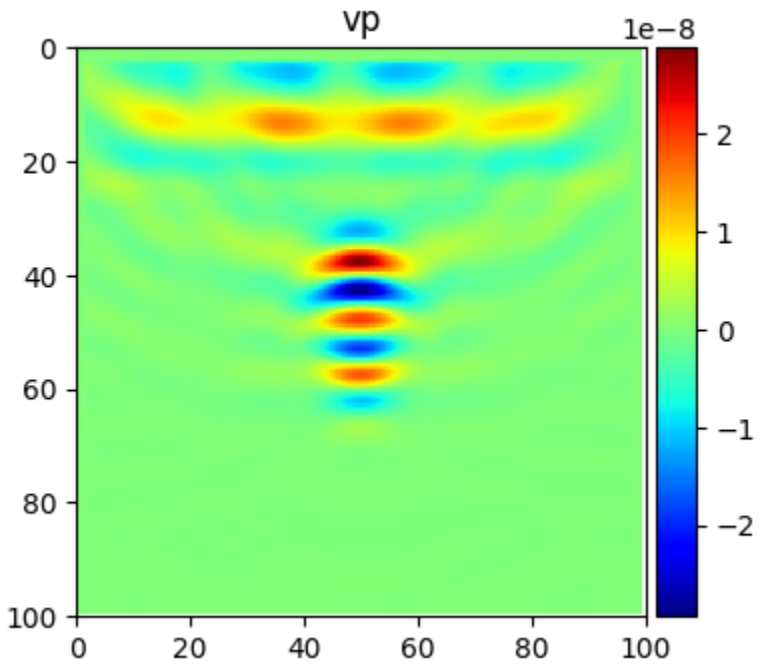
```
grad = {'vp': vp0.grad,
```

(continues on next page)

(continued from previous page)

```
        'vs': vs0.grad,
        'rho':rho0.grad
        }
```

```
# Time to plot the results
splt.earth_model(grad, ['vp'], cmap='jet');
```



CHAPTER

SIXTEEN

CITING PYFWI

```
 @article{mardan2023pyfwi,
title = {PyFWI: {A Python} package for full-waveform inversion and reservoir monitoring},
author = {Mardan, Amir and Giroux, Bernard and Fabien-Ouellet, Gabriel},
journal = {SoftwareX},
volume = {22},
pages = {101384},
year = {2023},
publisher = {Elsevier},
doi = {10.1016/j.softx.2023.101384}
```

}

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

D

F

G

H

M

P

R

S

T

W