AVO

Programa (afim de ser interativo) no Jupyter Lab para explorar os conceitos de AVO (Amplitude Versus Offset) e Tuning em Camadas Finas, baseado no artigo de Hamlyn (2014)

> Thin Beds, Tuning and AVO - Hamlyn (2014) | Geophysical Tutorial
> Coordinated by Matt Hall | The Leading Edge - SEG
>
> https://library.seg.org/doi/10.1190/tle33121394.1

> Repositório: https://github.com/seg/tutorials-2014/tree/master/1412_Tuning_and_AVO
> Repositório: https://github.com/seg

1. Configuração e Importação de Bibliotecas

1.0 Importação e Verificação

In [1]:
```
# Verificar versão do Python e instalação do Jupyter
import sys
print(f"Python version: {sys.version}")
```

```
Python version: 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944
64 bit (AMD64)]
```

In [2]:
```
# Importanto Bibliotecas/Pacotes:

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import gridspec
from scipy import signal # Para a função Ricker
from ipywidgets import interact, interactive, fixed, IntSlider, FloatSlider, Lay
from IPython.display import display

# Configuração visual para plots (opcional, mas recomendado)
plt.style.use('seaborn-v0_8-whitegrid')
%matplotlib inline
```

In [3]:
```
# Baixar os scripts do tutorial do GitHub
# Baixar APENAS os scripts específicos do tutorial de Tuning e AVO

import urllib.request
import os

# Criar Diretório Para os Scripts:
```

```
if not os.path.exists('tuning_scripts'):
    os.makedirs('tuning_scripts')

# URLs diretas para os scripts específicos do tutorial 1412_Tuning_and_AVO:
scripts = {
    'tuning_wedge.py': 'https://raw.githubusercontent.com/seg/tutorials-2014/mas
    'tuning_prestack.py': 'https://raw.githubusercontent.com/seg/tutorials-2014/
}

print("Baixando scripts específicos do tutorial Thin Beds, Tuning and AVO...")

# Baixar cada script:
success_count = 0
for filename, url in scripts.items():
    try:
        urllib.request.urlretrieve(url, f'tuning_scripts/{filename}')
        print(f'✓ {filename} baixado com sucesso!')
        success_count += 1
    except Exception as e:
        print(f'✗ Erro ao baixar {filename}: {e}')

print(f"\nTotal de scripts baixados: {success_count}/{len(scripts)}")
```

```
Baixando scripts específicos do tutorial Thin Beds, Tuning and AVO...
✓ tuning_wedge.py baixado com sucesso!
✓ tuning_prestack.py baixado com sucesso!

Total de scripts baixados: 2/2
```

In [4]:
```
# Verificar o conteúdo baixado:

import os

print("Estrutura de Diretórios:")
print(f"Diretório Atual: {os.getcwd()}")

if os.path.exists('tuning_scripts'):
    print("\nConteúdo de 'tuning_scripts':")
    print()
    files = os.listdir('tuning_scripts')
    for file in files:
        file_path = os.path.join('tuning_scripts', file)
        file_size = os.path.getsize(file_path)
        print(f"  {file} ({file_size} bytes)")

    if len(files) == 0:
        print("  (vazio)")
else:
    print("Diretório 'tuning_scripts' não encontrado")
```

```
Estrutura de Diretórios:
Diretório Atual: C:\Users\Musa Deck\Documents\PYTHON\ON

Conteúdo de 'tuning_scripts':

  tuning_prestack.py (14850 bytes)
  tuning_wedge.py (7788 bytes)
```

1.1 tuning_wedge.py

In [5]:
```
# Vamos dar uma olhada rápida no conteúdo do primeiro script:

if os.path.exists('tuning_scripts/tuning_wedge.py'):
    print("=== Primeiras linhas de tuning_wedge.py ===")
    with open('tuning_scripts/tuning_wedge.py', 'r') as f:
        for i, line in enumerate(f):
            if i < 10:  # Mostrar primeiras 10 linhas
                print(f"{i+1:3d}: {line.rstrip()}")
            else:
                print("... (continua)")
                break
else:
    print("Arquivo tuning_wedge.py não encontrado")
```

```
=== Primeiras linhas de tuning_wedge.py ===
  1: """
  2: Python script to generate a zero-offset synthetic from a 3-layer wedge mode
l.
  3:
  4: Created by:    Wes Hamlyn
  5: Create Date:   19-Aug-2014
  6: Last Mod:      1-Nov-2014
  7:
  8: This script is provided without warranty of any kind.
  9:
 10: """
... (continua)
```

> IMPORTANTE: Python Version !! Teremos que passar de Python 2 para Python 3 !!

tuning_wedge.py:

In [6]:
```
# Examinar o script tuning_wedge.py completo:

if os.path.exists('tuning_scripts/tuning_wedge.py'):
    print("=== Conteúdo completo de tuning_wedge.py ===")
    with open('tuning_scripts/tuning_wedge.py', 'r') as f:
        content = f.read()
        print(content)
else:
    print("Arquivo tuning_wedge.py não encontrado")
```

```
=== Conteúdo completo de tuning_wedge.py ===
"""
Python script to generate a zero-offset synthetic from a 3-layer wedge model.

Created by:    Wes Hamlyn
Create Date:   19-Aug-2014
Last Mod:      1-Nov-2014

This script is provided without warranty of any kind.

"""



import numpy as np
import matplotlib.pyplot as plt
from matplotlib import gridspec



###########################################################
#
#       DEFINE MODELING PARAMETERS HERE
#


#   3-Layer Model Parameters [Layer1, Layer2, Layer 3]
vp_mod = [2500.0, 2600.0, 2550.0]  # P-wave velocity (m/s)
vs_mod = [1200.0, 1300.0, 1200.0]  # S-wave velocity (m/s)
rho_mod= [1.95, 2.0, 1.98]         # Density (g/cc)

dz_min = 0.0   # Minimum thickness of Layer 2 (m)
dz_max = 60.0  # Maximum thickness of Layer 2 (m)
dz_step= 1.0   # Thickness step from trace-to-trace (normally 1.0 m)


#   Ricker Wavelet Parameters
wvlt_length= 0.128
wvlt_cfreq = 30.0
wvlt_phase = 0.0


#   Trace Parameters
tmin = 0.0
tmax = 0.5
dt = 0.0001 # changing this from 0.0001 can affect the display quality

#   Plot Parameters
min_plot_time = 0.15
max_plot_time = 0.3
excursion = 2




###########################################################
#
#       FUNCTIONS DEFINITIONS
#

def plot_vawig(axhdl, data, t, excursion, highlight=None):
```

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

    [ntrc, nsamp] = data.shape




    t = np.hstack([0, t, t.max()])

    for i in range(0, ntrc):
        tbuf = excursion * data[i] / np.max(np.abs(data)) + i

        tbuf = np.hstack([i, tbuf, i])

        if i==highlight:
            lw = 2
        else:
            lw = 0.5

        axhdl.plot(tbuf, t, color='black', linewidth=lw)

        plt.fill_betweenx(t, tbuf, i, where=tbuf>i, facecolor=[0.6,0.6,1.0], line
width=0)
        plt.fill_betweenx(t, tbuf, i, where=tbuf<i, facecolor=[1.0,0.7,0.7], line
width=0)


    axhdl.set_xlim((-excursion, ntrc+excursion))
    axhdl.xaxis.tick_top()
    axhdl.xaxis.set_label_position('top')
    axhdl.invert_yaxis()



def ricker(cfreq, phase, dt, wvlt_length):
    '''
    Calculate a zero-phase ricker wavelet

    Usage:
    ------
    t, wvlt = wvlt_ricker(cfreq, dt, wvlt_length)

    cfreq: central frequency of wavelet in Hz
    phase: wavelet phase in degrees
    dt: sample rate in seconds
    wvlt_length: length of wavelet in seconds
    '''

    import numpy as np
    import scipy.signal as signal

    nsamp = int(wvlt_length/dt + 1)
    t_max = wvlt_length*0.5
    t_min = -t_max

    t = np.arange(t_min, t_max, dt)

    t = np.linspace(-wvlt_length/2, (wvlt_length-dt)/2, wvlt_length/dt)
```

```
    wvlt = (1.0 - 2.0*(np.pi**2)*(cfreq**2)*(t**2)) * np.exp(-(np.pi**2)*(cfreq**
2)*(t**2))

    if phase != 0:
        phase = phase*np.pi/180.0
        wvlth = signal.hilbert(wvlt)
        wvlth = np.imag(wvlth)
        wvlt = np.cos(phase)*wvlt - np.sin(phase)*wvlth

    return t, wvlt



def calc_rc(vp_mod, rho_mod):
    '''
    rc_int = calc_rc(vp_mod, rho_mod)
    '''

    nlayers = len(vp_mod)
    nint = nlayers - 1

    rc_int = []
    for i in range(0, nint):
        buf1 = vp_mod[i+1]*rho_mod[i+1]-vp_mod[i]*rho_mod[i]
        buf2 = vp_mod[i+1]*rho_mod[i+1]+vp_mod[i]*rho_mod[i]
        buf3 = buf1/buf2
        rc_int.append(buf3)

    return rc_int


def calc_times(z_int, vp_mod):
    '''
    t_int = calc_times(z_int, vp_mod)
    '''

    nlayers = len(vp_mod)
    nint = nlayers - 1

    t_int = []
    for i in range(0, nint):
        if i == 0:
            tbuf = z_int[i]/vp_mod[i]
            t_int.append(tbuf)
        else:
            zdiff = z_int[i]-z_int[i-1]
            tbuf = 2*zdiff/vp_mod[i] + t_int[i-1]
            t_int.append(tbuf)

    return t_int



def digitize_model(rc_int, t_int, t):
    '''
    rc = digitize_model(rc, t_int, t)

    rc = reflection coefficients corresponding to interface times
    t_int = interface times
    t = regularly sampled time series defining model sampling
```

```
    '''

    import numpy as np

    nlayers = len(rc_int)
    nint = nlayers - 1
    nsamp = len(t)

    rc = list(np.zeros(nsamp,dtype='float'))
    lyr = 0

    for i in range(0, nsamp):

        if t[i] >= t_int[lyr]:
            rc[i] = rc_int[lyr]
            lyr = lyr + 1

        if lyr > nint:
            break

    return rc




##########################################################
#
#       COMPUTATIONS BELOW HERE...
#



#   Some handy constants
nlayers = len(vp_mod)
nint = nlayers - 1
nmodel = int((dz_max-dz_min)/dz_step+1)

#   Generate ricker wavelet
wvlt_t, wvlt_amp = ricker(wvlt_cfreq, wvlt_phase, dt, wvlt_length)

#   Calculate reflectivities from model parameters
rc_int = calc_rc(vp_mod, rho_mod)


syn_zo = []
rc_zo = []
lyr_times = []
for model in range(0, nmodel):

    #   Calculate interface depths
    z_int = [500.0]
    z_int.append(z_int[0]+dz_min+dz_step*model)

    #   Calculate interface times
    t_int = calc_times(z_int, vp_mod)
    lyr_times.append(t_int)

    #   Digitize 3-layer model
    nsamp = int((tmax-tmin)/dt) + 1
    t = []
```

```
        for i in range(0,nsamp):
            t.append(i*dt)

        rc = digitize_model(rc_int, t_int, t)
        rc_zo.append(rc)

        #   Convolve wavelet with reflectivities
        syn_buf = np.convolve(rc, wvlt_amp, mode='same')
        syn_buf = list(syn_buf)
        syn_zo.append(syn_buf)
        print "finished step %i" % (model)

    syn_zo = np.array(syn_zo)
    t = np.array(t)
    lyr_times = np.array(lyr_times)
    lyr_indx = np.array(np.round(lyr_times/dt), dtype='int16')

    # Use the transpose because rows are traces;
    # columns are time samples.
    tuning_trace = np.argmax(np.abs(syn_zo.T)) % syn_zo.T.shape[1]
    tuning_thickness = tuning_trace * dz_step

    #   Plotting Code
    [ntrc, nsamp] = syn_zo.shape

    fig = plt.figure(figsize=(12, 14))
    fig.set_facecolor('white')

    gs = gridspec.GridSpec(3, 1, height_ratios=[1, 1, 1])

    ax0 = fig.add_subplot(gs[0])
    ax0.plot(lyr_times[:,0], color='blue', lw=1.5)
    ax0.plot(lyr_times[:,1], color='red', lw=1.5)
    ax0.set_ylim((min_plot_time,max_plot_time))
    ax0.invert_yaxis()
    ax0.set_xlabel('Thickness (m)')
    ax0.set_ylabel('Time (s)')
    plt.text(2,
            min_plot_time + (lyr_times[0,0] - min_plot_time)/2.,
            'Layer 1',
            fontsize=16)
    plt.text(dz_max/dz_step - 2,
            lyr_times[-1,0] + (lyr_times[-1,1] - lyr_times[-1,0])/2.,
            'Layer 2',
            fontsize=16,
            horizontalalignment='right')
    plt.text(2,
            lyr_times[0,0] + (max_plot_time - lyr_times[0,0])/2.,
            'Layer 3',
            fontsize=16)
    plt.gca().xaxis.tick_top()
    plt.gca().xaxis.set_label_position('top')
    ax0.set_xlim((-excursion, ntrc+excursion))

    ax1 = fig.add_subplot(gs[1])
    plot_vawig(ax1, syn_zo, t, excursion, highlight=tuning_trace)
    ax1.plot(lyr_times[:,0], color='blue', lw=1.5)
    ax1.plot(lyr_times[:,1], color='red', lw=1.5)
    ax1.set_ylim((min_plot_time,max_plot_time))
    ax1.invert_yaxis()
```

```
ax1.set_xlabel('Thickness (m)')
ax1.set_ylabel('Time (s)')

ax2 = fig.add_subplot(gs[2])
ax2.plot(syn_zo[:,lyr_indx[:,0]], color='blue')
ax2.set_xlim((-excursion, ntrc+excursion))
ax2.axvline(tuning_trace, color='k', lw=2)
ax2.grid()
ax2.set_title('Upper interface amplitude')
ax2.set_xlabel('Thickness (m)')
ax2.set_ylabel('Amplitude')
plt.text(tuning_trace + 2,
        plt.ylim()[0] * 1.1,
        'tuning thickness = {0} m'.format(str(tuning_thickness)),
        fontsize=16)

plt.savefig('figure_1.png')
plt.show()
```

In [7]:

```
# Fazer uma correção para Python 3 (print statement)
# Corrigir todos os problemas identificados:

try:
    # Ler o conteúdo original
    with open('tuning_scripts/tuning_wedge.py', 'r') as f:
        content = f.read()

    # Correção 1: print statement
    content = content.replace('print "finished step %i" % (model)', 'print("fini

    # Correção 2: Problema no linspace - linha 117
    content = content.replace(
        't = np.linspace(-wvlt_length/2, (wvlt_length-dt)/2, wvlt_length/dt)',
        't = np.linspace(-wvlt_length/2, (wvlt_length-dt)/2, int(wvlt_length/dt)
    )

    # Correção 3: Outro linspace - linha 186
    content = content.replace(
        't = np.linspace(-wvlt_length/2, (wvlt_length-dt)/2, wvlt_length/dt)',
        't = np.linspace(-wvlt_length/2, (wvlt_length-dt)/2, int(wvlt_length/dt)
    )

    # Salvar versão corrigida
    with open('tuning_scripts/tuning_wedge_py3.py', 'w') as f:
        f.write(content)

    print("✓ Script corrigido para Python 3 !!")

except Exception as e:
    print(f"Erro ao corrigir: {e}")
```

```
✓ Script corrigido para Python 3 !!
```

1.2 tuning_prestack.py

In [8]:

```
# Vamos dar uma olhada rápida no conteúdo do segundo script:
```

```python
if os.path.exists('tuning_scripts/tuning_prestack.py'):
    print("=== Primeiras linhas de tuning_prestack.py ===")
    with open('tuning_scripts/tuning_prestack.py', 'r') as f:
        for i, line in enumerate(f):
            if i < 10:  # Mostrar primeiras 10 linhas
                print(f"{i+1:3d}: {line.rstrip()}")
            else:
                print("... (continua)")
                break
else:
    print("Arquivo tuning_prestack.py não encontrado")
```

```
=== Primeiras linhas de tuning_prestack.py ===
  1: """
  2: Python script to generate a synthetic angle gather from a 3-layer property m
odel
  3: to examine pre-stack tuning effects.
  4: 
  5: Created by:    Wes Hamlyn
  6: Create Date:   19-Aug-2014
  7: Last Mod:      1-Nov-2014
  8: 
  9: This script is provided without warranty of any kind.
 10: 
... (continua)
```

tuning_prestack.py:

In [9]:

```python
# Examinar o script tuning_prestack.py completo:

if os.path.exists('tuning_scripts/tuning_prestack.py'):
    print("=== Conteúdo completo de tuning_prestack.py ===")
    with open('tuning_scripts/tuning_prestack.py', 'r') as f:
        content = f.read()
        print(content)
else:
    print("Arquivo tuning_prestack.py não encontrado")
```

```
=== Conteúdo completo de tuning_prestack.py ===
"""
Python script to generate a synthetic angle gather from a 3-layer property model
to examine pre-stack tuning effects.

Created by:    Wes Hamlyn
Create Date:   19-Aug-2014
Last Mod:      1-Nov-2014

This script is provided without warranty of any kind.

"""

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


###########################################################
#
#       DEFINE MODELING PARAMETERS HERE
#


#   3-Layer Model Parameters [Layer1, Layer2, Layer 3]
vp_mod = [2500.0, 2600.0, 2550.0]  # P-wave velocity (m/s)
vs_mod = [1200.0, 1300.0, 1200.0]  # S-wave velocity (m/s)
rho_mod= [1.95, 2.0, 1.98]         # Density (g/cc)

thickness = 17.0  # vertical thickness of layer 2 in metres

#   Angle range for incident rays
theta1_min = 0.0    # best to leave this set to zero
theta1_max = 40.0
theta1_step= 1.0


#   Ricker Wavelet Parameters
wvlt_length= 0.128  # milliseconds
wvlt_cfreq = 30.0   # Hz
wvlt_phase = 0.0    # Degrees


#   Trace Parameters
tmin = 0.0
tmax = 0.5
dt = 0.0001 # changing this from 0.00005 can affect the display quality


#   Plotting Display Parameters
min_plot_time = 0.15
max_plot_time = 0.3
excursion = 2




###########################################################
#
#       FUNCTIONS DEFINITIONS
```

```
#

def plot_vawig(axhdl, data, t, excursion):

    import numpy as np
    import matplotlib.pyplot as plt

    [ntrc, nsamp] = data.shape




    t = np.hstack([0, t, t.max()])

    for i in range(0, ntrc):
        tbuf = excursion * data[i,:] / np.max(np.abs(data)) + i

        tbuf = np.hstack([i, tbuf, i])

        axhdl.plot(tbuf, t, color='black', linewidth=0.5)
        plt.fill_betweenx(t, tbuf, i, where=tbuf>i, facecolor=[0.6,0.6,1.0], line
width=0)
        plt.fill_betweenx(t, tbuf, i, where=tbuf<i, facecolor=[1.0,0.7,0.7], line
width=0)

    axhdl.set_xlim((-excursion, ntrc+excursion))
    axhdl.xaxis.tick_top()
    axhdl.xaxis.set_label_position('top')
    axhdl.invert_yaxis()



def ricker(cfreq, phase, dt, wvlt_length):
    '''
    Calculate a zero-phase ricker wavelet

    Usage:
    ------
    t, wvlt = wvlt_ricker(cfreq, dt, wvlt_length)

    cfreq: central frequency of wavelet in Hz
    phase: wavelet phase in degrees
    dt: sample rate in seconds
    wvlt_length: length of wavelet in seconds
    '''

    import numpy as np
    import scipy.signal as signal

    nsamp = int(wvlt_length/dt + 1)
    t_max = wvlt_length*0.5
    t_min = -t_max

    t = np.arange(t_min, t_max, dt)

    t = np.linspace(-wvlt_length/2, (wvlt_length-dt)/2, wvlt_length/dt)
    wvlt = (1.0 - 2.0*(np.pi**2)*(cfreq**2)*(t**2)) * np.exp(-(np.pi**2)*(cfreq**
2)*(t**2))

    if phase != 0:
```

```
        phase = phase*np.pi/180.0
        wvlth = signal.hilbert(wvlt)
        wvlth = np.imag(wvlth)
        wvlt = np.cos(phase)*wvlt - np.sin(phase)*wvlth

    return t, wvlt



def calc_times(z_int, vp_mod):
    '''
    Calculate two-way travel time through a layered model

    Usage:
    -----
    t_int = calc_times(z_int, vp_mod)

    '''

    nlayers = len(vp_mod)
    nint = nlayers - 1

    t_int = []
    for i in range(0, nint):
        if i == 0:
            tbuf = z_int[i]/vp_mod[i]
            t_int.append(tbuf)
        else:
            zdiff = z_int[i]-z_int[i-1]
            zdiff = zdiff*2.0   # multiply by 2 for two-way traveltimes
            tbuf = zdiff/vp_mod[i] + t_int[i-1]
            tbuf = tbuf
            t_int.append(tbuf)

    return t_int



def digitize_model(rc_int, t_int, t):
    '''
    Sample a simple layered reflectivity model

    Usage:
    ------
    rc = digitize_model(rc, t_int, t)

    rc = reflection coefficients corresponding to interface times
    t_int = interface times
    t = regularly sampled time series defining model sampling
    '''

    import numpy as np

    nlayers = len(rc_int)
    nint = nlayers - 1
    nsamp = len(t)

    rc = list(np.zeros(nsamp,dtype='float'))
    lyr = 0
```

```
    for i in range(0, nsamp):

        if t[i] >= t_int[lyr]:
            rc[i] = rc_int[lyr]
            lyr = lyr + 1

        if lyr > nint:
            break

    return rc


def rc_zoep(vp1, vs1, rho1, vp2, vs2, rho2, theta1):
    '''
    Reflection & Transmission coefficients calculated using full Zoeppritz
    equations.

    Usage:
    ------
    R = rc_zoep(vp1, vs1, rho1, vp2, vs2, rho2, theta1)

    Reference:
    ----------
    The Rock Physics Handbook, Dvorkin et al.
    '''

    import math

    # Cast inputs to floats
    vp1  = float(vp1)
    vp2  = float(vp2)
    vs1  = float(vs1)
    vs2  = float(vs2)
    rho1 = float(rho1)
    rho2 = float(rho2)
    theta1 = float(theta1)

    # Calculate reflection & transmission angles
    theta1 = math.radians(theta1)   # Convert theta1 to radians
    p      = ray_param(vp1, math.degrees(theta1)) # Ray parameter
    theta2 = math.asin(p*vp2);      # Transmission angle of P-wave
    phi1   = math.asin(p*vs1);      # Reflection angle of converted S-wave
    phi2   = math.asin(p*vs2);      # Transmission angle of converted S-wave

    # Matrix form of Zoeppritz Equations... M & N are two of the matricies
    M = np.array([ \
        [-math.sin(theta1), -math.cos(phi1), math.sin(theta2), math.cos(phi2)],\
        [math.cos(theta1), -math.sin(phi1), math.cos(theta2), -math.sin(phi2)],\
        [2*rho1*vs1*math.sin(phi1)*math.cos(theta1), rho1*vs1*(1-2*math.sin(phi1)
**2),\
            2*rho2*vs2*math.sin(phi2)*math.cos(theta2), rho2*vs2*(1-2*math.sin(ph
i2)**2)],\
        [-rho1*vp1*(1-2*math.sin(phi1)**2), rho1*vs1*math.sin(2*phi1), \
            rho2*vp2*(1-2*math.sin(phi2)**2), -rho2*vs2*math.sin(2*phi2)]
        ], dtype='float')

    N = np.array([ \
        [math.sin(theta1), math.cos(phi1), -math.sin(theta2), -math.cos(phi2)],\
        [math.cos(theta1), -math.sin(phi1), math.cos(theta2), -math.sin(phi2)],\
        [2*rho1*vs1*math.sin(phi1)*math.cos(theta1), rho1*vs1*(1-2*math.sin(phi1)
```

```
**2),\
            2*rho2*vs2*math.sin(phi2)*math.cos(theta2), rho2*vs2*(1-2*math.sin(ph
i2)**2)],\
        [rho1*vp1*(1-2*math.sin(phi1)**2), -rho1*vs1*math.sin(2*phi1),\
            -rho2*vp2*(1-2*math.sin(phi2)**2), rho2*vs2*math.sin(2*phi2)]\
        ], dtype='float')

    # This is the important step, calculating coefficients for all modes and rays
    R = np.dot(np.linalg.inv(M), N);

    return R


def ray_param(v, theta):
    '''
    Calculates the ray parameter p

    Usage:
    ------
        p = ray_param(v, theta)

    Inputs:
    -------
            v = interval velocity
        theta = incidence angle of ray (degrees)

    Output:
    -------
        p = ray parameter (i.e. sin(theta)/v )
    '''

    import math

    # Cast inputs to floats
    theta = float(theta)
    v = float(v)

    p = math.sin(math.radians(theta))/v # ray parameter calculation

    return p




##########################################################
#
#       COMPUTATIONS HAPPEN BELOW HERE
#


#   Some handy constants
nlayers = len(vp_mod)
nint = nlayers - 1
nangles = int( (theta1_max-theta1_min)/theta1_step + 1)


#   Generate ricker wavelet
wvlt_t, wvlt_amp = ricker(wvlt_cfreq, wvlt_phase, dt, wvlt_length)
```

```
#   Calculate reflectivities from model parameters
rc_zoep_pp = []
theta1 = []
for i in range(0, nangles):
    theta1_buf = i*theta1_step + theta1_min
    rc_buf1 = rc_zoep(vp_mod[0], vs_mod[0], rho_mod[0], vp_mod[1], vs_mod[1], rho
_mod[1], theta1_buf)
    rc_buf2 = rc_zoep(vp_mod[1], vs_mod[1], rho_mod[1], vp_mod[2], vs_mod[2], rho
_mod[2], theta1_buf)

    theta1.append(theta1_buf)
    rc_zoep_pp.append([rc_buf1[0,0], rc_buf2[0,0]])


#   Define time sample vector for output model & traces
nsamp = int((tmax-tmin)/dt) + 1
t = []
for i in range(0,nsamp):
    t.append(i*dt)


syn_zoep_pp = []
lyr_times = []
print "\n\nStarting synthetic calcuations...\n"
for angle in range(0, nangles):

    dz_app = thickness

    #   To calculate apparent thickness of layer 2 based on incidence angle
    #   uncomment the following three rows (e.g. ray-synthetics)
    #p = ray_param(vp_mod[0], angle)
    #angle2 = math.degrees(math.asin(p*vp_mod[1]))
    #dz_app = thickness/math.cos(math.radians(angle2))

    #   Calculate interface depths
    z_int = [500.0]
    z_int.append(z_int[0] + dz_app)

    #   Calculate interface times
    t_int = calc_times(z_int, vp_mod)
    lyr_times.append(t_int)

    #   Digitize 3-layer model
    rc = digitize_model(rc_zoep_pp[angle], t_int, t)

    #   Convolve wavelet with reflectivities
    syn_buf = np.convolve(rc, wvlt_amp, mode='same')
    syn_buf = list(syn_buf)
    syn_zoep_pp.append(syn_buf)
    print "Calculated angle %i" % (angle)


#    Convert data arrays from lists/tuples to numpy arrays
syn_zoep_pp = np.array(syn_zoep_pp)
rc_zoep_pp = np.array(rc_zoep_pp)
t = np.array(t)


#   Calculate array indicies corresponding to top/base interfaces
lyr_times = np.array(lyr_times)
```

```
lyr_indx = np.array(np.round(lyr_times/dt), dtype='int16')
lyr1_indx = list(lyr_indx[:,0])
lyr2_indx = list(lyr_indx[:,1])


#   Copy convoved top/base reflectivity values to Lists for easier plotting
[ntrc, nsamp] = syn_zoep_pp.shape
line1 = []
line2 = []
for i in range(0, ntrc):
    line1.append(syn_zoep_pp[i,lyr1_indx[i]])
    line2.append(syn_zoep_pp[i,lyr2_indx[i]])



#   AVO inversion for NI and GRAD from analytic and convolved reflectivity
#   values and print the results to the command line.  Linear least squares
#   method is used for estimating NI and GRAD coefficients.
Yzoep = np.array(rc_zoep_pp[:,0])
Yzoep = Yzoep.reshape((ntrc, 1))

Yconv = np.array(line1)
Yconv = Yconv.reshape((ntrc, 1))

ones = np.ones(ntrc)
ones = ones.reshape((ntrc,1))

sintheta2 = np.sin(np.radians(np.arange(0, ntrc)))**2
sintheta2 = sintheta2.reshape((ntrc, 1))

X = np.hstack((ones, sintheta2))

#   ... matrix solution of normal equations
Azoep = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), Yzoep)
Aconv = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), Yconv)

print'\n\n'
print '  Method       NI         GRAD'
print '---------------------------------'
print ' Zoeppritz%11.5f%12.5f' % (Azoep[0], Azoep[1])
print ' Convolved%10.5f%12.5f' % (Aconv[0], Aconv[1])



#   Create a "digital" time domain version of the input property model for
#   easy plotting and comparison with the time synthetic traces
vp_dig = np.zeros(t.shape)
vs_dig = np.zeros(t.shape)
rho_dig = np.zeros(t.shape)

vp_dig[0:lyr1_indx[0]] = vp_mod[0]
vp_dig[(lyr1_indx[0]):lyr2_indx[0]] = vp_mod[1]
vp_dig[(lyr2_indx[0]):] = vp_mod[2]

vs_dig[0:lyr1_indx[0]] = vs_mod[0]
vs_dig[(lyr1_indx[0]):lyr2_indx[0]] = vs_mod[1]
vs_dig[(lyr2_indx[0]):] = vs_mod[2]

rho_dig[0:lyr1_indx[0]] = rho_mod[0]
rho_dig[(lyr1_indx[0]):lyr2_indx[0]] = rho_mod[1]
```

```
rho_dig[(lyr2_indx[0]):] = rho_mod[2]



#############################################
#
#       PLOTTING HAPPENS BELOW HERE
#


#   Create the plot figure
fig = plt.figure(figsize=(16, 12))
fig.set_facecolor('white')


#   Plot log curves in two-way time
ax0a = fig.add_subplot(261)
l_vp_dig, = ax0a.plot(vp_dig/1000, t, 'k', lw=2)
ax0a.set_ylim((min_plot_time,max_plot_time))
ax0a.set_xlim(1.5, 4.0)
ax0a.invert_yaxis()
ax0a.set_ylabel('TWT (sec)')
ax0a.xaxis.tick_top()
ax0a.xaxis.set_label_position('top')
ax0a.set_xlabel('Vp (km/s)')
ax0a.axhline(lyr_times[0,0], color='blue', lw=2, alpha=0.5)
ax0a.axhline(lyr_times[0,1], color='red', lw=2, alpha=0.5)
ax0a.grid()

ax0b = fig.add_subplot(262)
l_vs_dig, = ax0b.plot(vs_dig/1000, t, 'k', lw=2)
ax0b.set_ylim((min_plot_time,max_plot_time))
ax0b.set_xlim((0.8, 2.0))
ax0b.invert_yaxis()
ax0b.xaxis.tick_top()
ax0b.xaxis.set_label_position('top')
ax0b.set_xlabel('Vs (km/s)')
ax0b.set_yticklabels('')
ax0b.axhline(lyr_times[0,0], color='blue', lw=2, alpha=0.5)
ax0b.axhline(lyr_times[0,1], color='red', lw=2, alpha=0.5)
ax0b.grid()

ax0c = fig.add_subplot(263)
l_rho_dig, = ax0c.plot(rho_dig, t, 'k', lw=2)
ax0c.set_ylim((min_plot_time,max_plot_time))
ax0c.set_xlim((1.6, 2.6))
ax0c.invert_yaxis()
ax0c.xaxis.tick_top()
ax0c.xaxis.set_label_position('top')
ax0c.set_xlabel('Den')
ax0c.set_yticklabels('')
ax0c.axhline(lyr_times[0,0], color='blue', lw=2, alpha=0.5)
ax0c.axhline(lyr_times[0,1], color='red', lw=2, alpha=0.5)
ax0c.grid()

plt.text(2.55,
        min_plot_time + (lyr_times[0,0] - min_plot_time)/2.,
        'Layer 1',
        fontsize=14,
```

```
        horizontalalignment='right')
plt.text(2.55,
        lyr_times[0,1] + (lyr_times[0,0] - lyr_times[0,1])/2. + 0.002,
        'Layer 2',
        fontsize=14,
        horizontalalignment='right')
plt.text(2.55,
        lyr_times[0,0] + (max_plot_time - lyr_times[0,0])/2.,
        'Layer 3',
        fontsize=14,
        horizontalalignment='right')

#   Plot synthetic gather and model top & base interfaces in two-way time
ax1 = fig.add_subplot(222)
plot_vawig(ax1, syn_zoep_pp, t, excursion)
ax1.set_ylim((min_plot_time,max_plot_time))
l_int1, = ax1.plot(lyr_times[:,0], color='blue', lw=2)
l_int2, = ax1.plot(lyr_times[:,1], color='red', lw=2)

plt.legend([l_int1,l_int2], ['Interface 1', 'Interface 2'], loc=4)
ax1.invert_yaxis()
label_str = 'Synthetic angle gather\nLayer 2 thickness = %4.1fm' % thickness
ax1.set_xlabel(label_str, fontsize=14)
ax1.set_ylabel('TWT (sec)')


#   Plot Zoeppritz and convolved reflectivity curves
ax2 = fig.add_subplot(2,2,3)

l_syn1, = ax2.plot(line1, color='blue', linewidth=2)
l_rc1, = ax2.plot( rc_zoep_pp[:,0], '--', color='blue', lw=2)

ax2.set_xlim((-excursion, ntrc+excursion))
ax2.grid()
ax2.set_xlabel('Angle of incidence (deg)')
ax2.set_ylabel('Reflection coefficient')
ax2.set_title('Upper interface reflectivity')
plt.legend([l_syn1, l_rc1], ['Convolved', 'Zoepprtiz'], loc=0)

ax3 = fig.add_subplot(2,2,4)
l_syn2, = ax3.plot(line2, color='red', linewidth=2)
l_rc2, = ax3.plot( rc_zoep_pp[:,1], '--', color='red', lw=2)
ax3.set_xlim((-excursion, ntrc+excursion))
ax3.grid()
ax3.set_xlabel('Angle of incidence (deg)')
ax3.set_ylabel('Reflection coefficient')
ax3.set_title('Lower interface reflectivity')
plt.legend([l_syn2, l_rc2], ['Convolved', 'Zoepprtiz'], loc=0)


#   Save the plot
plt.savefig('figure_2.png')

#   Display the plot
plt.show()
```

> IMPORTANTE: Python Version !! Teremos que passar de Python 2 para Python 3 !!

Python 2 para Python 3

In [10]:
```
# Fazer uma correção para Python 3 (print statement)
# Corrigir todos os problemas identificados:

import numpy as np
import io

OUTPUT_FILE = 'tuning_scripts/tuning_prestack_py3.py'

# CONTEÚDO FINAL E CORRIGIDO DO SCRIPT
corrected_content = """
# Python script to generate a synthetic angle gather from a 3-layer property mod
# to examine pre-stack tuning effects.

# Created by:    Wes Hamlyn
# Create Date:   19-Aug-2014
# Last Mod:      1-Nov-2014

# This script is provided without warranty of any kind.


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


############################################################
#
#       DEFINE MODELING PARAMETERS HERE
#


#   3-Layer Model Parameters [Layer1, Layer2, Layer 3]
vp_mod = [2500.0, 2600.0, 2550.0]  # P-wave velocity (m/s)
vs_mod = [1200.0, 1300.0, 1200.0]  # S-wave velocity (m/s)
rho_mod= [1.95, 2.0, 1.98]         # Density (g/cc)

thickness = 17.0  # vertical thickness of layer 2 in metres

#   Angle range for incident rays
theta1_min = 0.0    # best to leave this set to zero
theta1_max = 40.0
theta1_step= 1.0


#   Ricker Wavelet Parameters
wvlt_length= 0.128  # milliseconds
wvlt_cfreq = 30.0   # Hz
wvlt_phase = 0.0    # Degrees


#   Trace Parameters
tmin = 0.0
```

```
tmax = 0.5
dt = 0.0001 # changing this from 0.00005 can affect the display quality


#   Plotting Display Parameters
min_plot_time = 0.15
max_plot_time = 0.3
excursion = 2




###########################################################
#
#       FUNCTIONS DEFINITIONS
#

def plot_vawig(axhdl, data, t, excursion):

    import numpy as np
    import matplotlib.pyplot as plt

    [ntrc, nsamp] = data.shape




    t = np.hstack([0, t, t.max()])

    for i in range(0, ntrc):
        tbuf = excursion * data[i,:] / np.max(np.abs(data)) + i

        tbuf = np.hstack([i, tbuf, i])

        axhdl.plot(tbuf, t, color='black', linewidth=0.5)
        plt.fill_betweenx(t, tbuf, i, where=tbuf>i, facecolor=[0.6,0.6,1.0], lin
        plt.fill_betweenx(t, tbuf, i, where=tbuf<i, facecolor=[1.0,0.7,0.7], lin

    axhdl.set_xlim((-excursion, ntrc+excursion))
    axhdl.xaxis.tick_top()
    axhdl.xaxis.set_label_position('top')
    axhdl.invert_yaxis()



def ricker(cfreq, phase, dt, wvlt_length):
    '''
    Calculate a zero-phase ricker wavelet

    Usage:
    ------
    t, wvlt = wvlt_ricker(cfreq, dt, wvlt_length)

    cfreq: central frequency of wavelet in Hz
    phase: wavelet phase in degrees
    dt: sample rate in seconds
    wvlt_length: length of wavelet in seconds
    '''

    import numpy as np
```

```
    import scipy.signal as signal

    nsamp = int(wvlt_length/dt + 1)
    t_max = wvlt_length*0.5
    t_min = -t_max

    t = np.arange(t_min, t_max, dt)

    # CORREÇÃO 1: Divisão de inteiro em np.linspace
    t = np.linspace(-wvlt_length/2, (wvlt_length-dt)/2, int(wvlt_length/dt))
    wvlt = (1.0 - 2.0*(np.pi**2)*(cfreq**2)*(t**2)) * np.exp(-(np.pi**2)*(cfreq*

    if phase != 0:
        phase = phase*np.pi/180.0
        wvlth = signal.hilbert(wvlt)
        wvlth = np.imag(wvlth)
        wvlt = np.cos(phase)*wvlt - np.sin(phase)*wvlth

    return t, wvlt



def calc_times(z_int, vp_mod):
    '''
    Calculate two-way travel time through a layered model

    Usage:
    -----
    t_int = calc_times(z_int, vp_mod)

    '''

    nlayers = len(vp_mod)
    nint = nlayers - 1

    t_int = []
    for i in range(0, nint):
        if i == 0:
            tbuf = z_int[i]/vp_mod[i]
            t_int.append(tbuf)
        else:
            zdiff = z_int[i]-z_int[i-1]
            zdiff = zdiff*2.0   # multiply by 2 for two-way traveltimes
            tbuf = zdiff/vp_mod[i] + t_int[i-1]
            tbuf = tbuf
            t_int.append(tbuf)

    return t_int



def digitize_model(rc_int, t_int, t):
    '''
    Sample a simple layered reflectivity model

    Usage:
    ------
    rc = digitize_model(rc, t_int, t)

    rc = reflection coefficients corresponding to interface times
```

```
    t_int = interface times
    t = regularly sampled time series defining model sampling
    '''

    import numpy as np

    nlayers = len(rc_int)
    nint = nlayers - 1
    nsamp = len(t)

    rc = list(np.zeros(nsamp,dtype='float'))
    lyr = 0

    for i in range(0, nsamp):

        if t[i] >= t_int[lyr]:
            rc[i] = rc_int[lyr]
            lyr = lyr + 1

        if lyr > nint:
            break

    return rc


def rc_zoep(vp1, vs1, rho1, vp2, vs2, rho2, theta1):
    '''
    Reflection & Transmission coefficients calculated using full Zoeppritz
    equations.

    Usage:
    ------
    R = rc_zoep(vp1, vs1, rho1, vp2, vs2, rho2, theta1)

    Reference:
    ----------
    The Rock Physics Handbook, Dvorkin et al.
    '''

    import math

    # Cast inputs to floats
    vp1  = float(vp1)
    vp2  = float(vp2)
    vs1  = float(vs1)
    vs2  = float(vs2)
    rho1 = float(rho1)
    rho2 = float(rho2)
    theta1 = float(theta1)

    # Calculate reflection & transmission angles
    theta1 = math.radians(theta1)   # Convert theta1 to radians
    p      = ray_param(vp1, math.degrees(theta1)) # Ray parameter
    theta2 = math.asin(p*vp2);      # Transmission angle of P-wave
    phi1   = math.asin(p*vs1);      # Reflection angle of converted S-wave
    phi2   = math.asin(p*vs2);      # Transmission angle of converted S-wave

    # Matrix form of Zoeppritz Equations... M & N are two of the matricies
    M = np.array([ \
        [-math.sin(theta1), -math.cos(phi1), math.sin(theta2), math.cos(phi2)],\
```

```
        [math.cos(theta1), -math.sin(phi1), math.cos(theta2), -math.sin(phi2)],\
        [2*rho1*vs1*math.sin(phi1)*math.cos(theta1), rho1*vs1*(1-2*math.sin(phi1
            2*rho2*vs2*math.sin(phi2)*math.cos(theta2), rho2*vs2*(1-2*math.sin(p
        [-rho1*vp1*(1-2*math.sin(phi1)**2), rho1*vs1*math.sin(2*phi1), \
            rho2*vp2*(1-2*math.sin(phi2)**2), -rho2*vs2*math.sin(2*phi2)]
        ], dtype='float')

    N = np.array([ \
        [math.sin(theta1), math.cos(phi1), -math.sin(theta2), -math.cos(phi2)],\
        [math.cos(theta1), -math.sin(phi1), math.cos(theta2), -math.sin(phi2)],\
        [2*rho1*vs1*math.sin(phi1)*math.cos(theta1), rho1*vs1*(1-2*math.sin(phi1
            2*rho2*vs2*math.sin(phi2)*math.cos(theta2), rho2*vs2*(1-2*math.sin(p
        [rho1*vp1*(1-2*math.sin(phi1)**2), -rho1*vs1*math.sin(2*phi1),\
            -rho2*vp2*(1-2*math.sin(phi2)**2), rho2*vs2*math.sin(2*phi2)]\
        ], dtype='float')

    # This is the important step, calculating coefficients for all modes and ray
    R = np.dot(np.linalg.inv(M), N);

    return R


def ray_param(v, theta):
    '''
    Calculates the ray parameter p

    Usage:
    ------
        p = ray_param(v, theta)

    Inputs:
    -------
            v = interval velocity
        theta = incidence angle of ray (degrees)

    Output:
    -------
        p = ray parameter (i.e. sin(theta)/v )
    '''

    import math

    # Cast inputs to floats
    theta = float(theta)
    v = float(v)

    p = math.sin(math.radians(theta))/v # ray parameter calculation

    return p




###########################################################
#
#       COMPUTATIONS HAPPEN BELOW HERE
#


#   Some handy constants
```

```
nlayers = len(vp_mod)
nint = nlayers - 1
nangles = int( (theta1_max-theta1_min)/theta1_step + 1)


#   Generate ricker wavelet
wvlt_t, wvlt_amp = ricker(wvlt_cfreq, wvlt_phase, dt, wvlt_length)


#   Calculate reflectivities from model parameters
rc_zoep_pp = []
theta1 = []
for i in range(0, nangles):
    theta1_buf = i*theta1_step + theta1_min
    rc_buf1 = rc_zoep(vp_mod[0], vs_mod[0], rho_mod[0], vp_mod[1], vs_mod[1], rh
    rc_buf2 = rc_zoep(vp_mod[1], vs_mod[1], rho_mod[1], vp_mod[2], vs_mod[2], rh

    theta1.append(theta1_buf)
    rc_zoep_pp.append([rc_buf1[0,0], rc_buf2[0,0]])


#   Define time sample vector for output model & traces
nsamp = int((tmax-tmin)/dt) + 1
t = []
for i in range(0,nsamp):
    t.append(i*dt)


syn_zoep_pp = []
lyr_times = []
# CORREÇÃO: print statement
print("\\n\\nStarting synthetic calcuations...\\n")
for angle in range(0, nangles):

    dz_app = thickness

    #   To calculate apparent thickness of layer 2 based on incidence angle
    #   uncomment the following three rows (e.g. ray-synthetics)
    #p = ray_param(vp_mod[0], angle)
    #angle2 = math.degrees(math.asin(p*vp_mod[1]))
    #dz_app = thickness/math.cos(math.radians(angle2))

    #   Calculate interface depths
    z_int = [500.0]
    z_int.append(z_int[0] + dz_app)

    #   Calculate interface times
    t_int = calc_times(z_int, vp_mod)
    lyr_times.append(t_int)

    #   Digitize 3-layer model
    rc = digitize_model(rc_zoep_pp[angle], t_int, t)

    #   Convolve wavelet with reflectivities
    syn_buf = np.convolve(rc, wvlt_amp, mode='same')
    syn_buf = list(syn_buf)
    syn_zoep_pp.append(syn_buf)
    # CORREÇÃO: print statement
    print("Calculated angle %i" % (angle))
```

```
#    Convert data arrays from lists/tuples to numpy arrays
syn_zoep_pp = np.array(syn_zoep_pp)
rc_zoep_pp = np.array(rc_zoep_pp)
t = np.array(t)


#   Calculate array indicies corresponding to top/base interfaces
lyr_times = np.array(lyr_times)
lyr_indx = np.array(np.round(lyr_times/dt), dtype='int16')
lyr1_indx = list(lyr_indx[:,0])
lyr2_indx = list(lyr_indx[:,1])


#   Copy convoved top/base reflectivity values to Lists for easier plotting
[ntrc, nsamp] = syn_zoep_pp.shape
line1 = []
line2 = []
for i in range(0, ntrc):
    line1.append(syn_zoep_pp[i,lyr1_indx[i]])
    line2.append(syn_zoep_pp[i,lyr2_indx[i]])



#   AVO inversion for NI and GRAD from analytic and convolved reflectivity
#   values and print the results to the command line.  Linear least squares
#   method is used for estimating NI and GRAD coefficients.
Yzoep = np.array(rc_zoep_pp[:,0])
Yzoep = Yzoep.reshape((ntrc, 1))

Yconv = np.array(line1)
Yconv = Yconv.reshape((ntrc, 1))

ones = np.ones(ntrc)
ones = ones.reshape((ntrc,1))

sintheta2 = np.sin(np.radians(np.arange(0, ntrc)))**2
sintheta2 = sintheta2.reshape((ntrc, 1))

X = np.hstack((ones, sintheta2))

#   ... matrix solution of normal equations
Azoep = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), Yzoep)
Aconv = np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), Yconv)

# CORREÇÃO FINAL: Indexação dupla para extrair o escalar (eliminando Deprecation
print('\\n\\n')
print('  Method       NI         GRAD')
print('---------------------------------')
print(' Zoeppritz%11.5f%12.5f' % (Azoep[0][0], Azoep[1][0])) # Corrigido aqui
print(' Convolved%10.5f%12.5f' % (Aconv[0][0], Aconv[1][0])) # Corrigido aqui



#   Create a "digital" time domain version of the input property model for
#   easy plotting and comparison with the time synthetic traces
vp_dig = np.zeros(t.shape)
vs_dig = np.zeros(t.shape)
rho_dig = np.zeros(t.shape)
```

```
vp_dig[0:lyr1_indx[0]] = vp_mod[0]
vp_dig[(lyr1_indx[0]):lyr2_indx[0]] = vp_mod[1]
vp_dig[(lyr2_indx[0]):] = vp_mod[2]

vs_dig[0:lyr1_indx[0]] = vs_mod[0]
vs_dig[(lyr1_indx[0]):lyr2_indx[0]] = vs_mod[1]
vs_dig[(lyr2_indx[0]):] = vs_mod[2]

rho_dig[0:lyr1_indx[0]] = rho_mod[0]
rho_dig[(lyr1_indx[0]):lyr2_indx[0]] = rho_mod[1]
rho_dig[(lyr2_indx[0]):] = rho_mod[2]




#############################################
#
#       PLOTTING HAPPENS BELOW HERE
#


#   Create the plot figure
fig = plt.figure(figsize=(16, 12))
fig.set_facecolor('white')


#   Plot log curves in two-way time
ax0a = fig.add_subplot(261)
l_vp_dig, = ax0a.plot(vp_dig/1000, t, 'k', lw=2)
ax0a.set_ylim((min_plot_time,max_plot_time))
ax0a.set_xlim(1.5, 4.0)
ax0a.invert_yaxis()
ax0a.set_ylabel('TWT (sec)')
ax0a.xaxis.tick_top()
ax0a.xaxis.set_label_position('top')
ax0a.set_xlabel('Vp (km/s)')
ax0a.axhline(lyr_times[0,0], color='blue', lw=2, alpha=0.5)
ax0a.axhline(lyr_times[0,1], color='red', lw=2, alpha=0.5)
ax0a.grid()

ax0b = fig.add_subplot(262)
l_vs_dig, = ax0b.plot(vs_dig/1000, t, 'k', lw=2)
ax0b.set_ylim((min_plot_time,max_plot_time))
ax0b.set_xlim((0.8, 2.0))
ax0b.invert_yaxis()
ax0b.xaxis.tick_top()
ax0b.xaxis.set_label_position('top')
ax0b.set_xlabel('Vs (km/s)')
ax0b.set_yticklabels('')
ax0b.axhline(lyr_times[0,0], color='blue', lw=2, alpha=0.5)
ax0b.axhline(lyr_times[0,1], color='red', lw=2, alpha=0.5)
ax0b.grid()

ax0c = fig.add_subplot(263)
l_rho_dig, = ax0c.plot(rho_dig, t, 'k', lw=2)
ax0c.set_ylim((min_plot_time,max_plot_time))
ax0c.set_xlim((1.6, 2.6))
ax0c.invert_yaxis()
ax0c.xaxis.tick_top()
ax0c.xaxis.set_label_position('top')
```

```
ax0c.set_xlabel('Den')
ax0c.set_yticklabels('')
ax0c.axhline(lyr_times[0,0], color='blue', lw=2, alpha=0.5)
ax0c.axhline(lyr_times[0,1], color='red', lw=2, alpha=0.5)
ax0c.grid()

plt.text(2.55,
        min_plot_time + (lyr_times[0,0] - min_plot_time)/2.,
        'Layer 1',
        fontsize=14,
        horizontalalignment='right')
plt.text(2.55,
        lyr_times[0,1] + (lyr_times[0,0] - lyr_times[0,1])/2. + 0.002,
        'Layer 2',
        fontsize=14,
        horizontalalignment='right')
plt.text(2.55,
        lyr_times[0,0] + (max_plot_time - lyr_times[0,0])/2.,
        'Layer 3',
        fontsize=14,
        horizontalalignment='right')

#   Plot synthetic gather and model top & base interfaces in two-way time
ax1 = fig.add_subplot(222)
plot_vawig(ax1, syn_zoep_pp, t, excursion)
ax1.set_ylim((min_plot_time,max_plot_time))
l_int1, = ax1.plot(lyr_times[:,0], color='blue', lw=2)
l_int2, = ax1.plot(lyr_times[:,1], color='red', lw=2)

plt.legend([l_int1,l_int2], ['Interface 1', 'Interface 2'], loc=4)
ax1.invert_yaxis()
# CORREÇÃO: Uso de f-string para o label de plotagem
label_str = f'Synthetic angle gather\\nLayer 2 thickness = {thickness:4.1f}m'
ax1.set_xlabel(label_str, fontsize=14)
ax1.set_ylabel('TWT (sec)')


#   Plot Zoeppritz and convolved reflectivity curves
ax2 = fig.add_subplot(2,2,3)

l_syn1, = ax2.plot(line1, color='blue', linewidth=2)
l_rc1, = ax2.plot( rc_zoep_pp[:,0], '--', color='blue', lw=2)

ax2.set_xlim((-excursion, ntrc+excursion))
ax2.grid()
ax2.set_xlabel('Angle of incidence (deg)')
ax2.set_ylabel('Reflection coefficient')
ax2.set_title('Upper interface reflectivity')
plt.legend([l_syn1, l_rc1], ['Convolved', 'Zoepprtiz'], loc=0)

ax3 = fig.add_subplot(2,2,4)
l_syn2, = ax3.plot(line2, color='red', linewidth=2)
l_rc2, = ax3.plot( rc_zoep_pp[:,1], '--', color='red', lw=2)
ax3.set_xlim((-excursion, ntrc+excursion))
ax3.grid()
ax3.set_xlabel('Angle of incidence (deg)')
ax3.set_ylabel('Reflection coefficient')
ax3.set_title('Lower interface reflectivity')
plt.legend([l_syn2, l_rc2], ['Convolved', 'Zoepprtiz'], loc=0)
```

```
#   Save the plot
plt.savefig('figure_2.png')

#   Display the plot
plt.show()
"""

try:
    # FORÇANDO A ESCRITA DO ARQUIVO COM CODIFICAÇÃO UTF-8
    with open(OUTPUT_FILE, 'w', encoding='utf-8') as f:
        f.write(corrected_content)

    print(f"✓ Arquivo '{OUTPUT_FILE}' atualizado. O **DeprecationWarning** será

except Exception as e:
    print(f"✗ Erro ao salvar o arquivo: {e}")
```

```
✓ Arquivo 'tuning_scripts/tuning_prestack_py3.py' atualizado. O **DeprecationWar
ning** será resolvido na próxima execução.
```

2. Efeito Tuning (Zero-Offset)

tuning_wedge_py3.py

FUNCTIONS DEFINITIONS: ricker, calc_rc, plot_vawig

PARAMETERS: Vp, Vs, Rho, cfreq

Computations & Plotting: executa a modelagem (convolução) e gera o gráfico

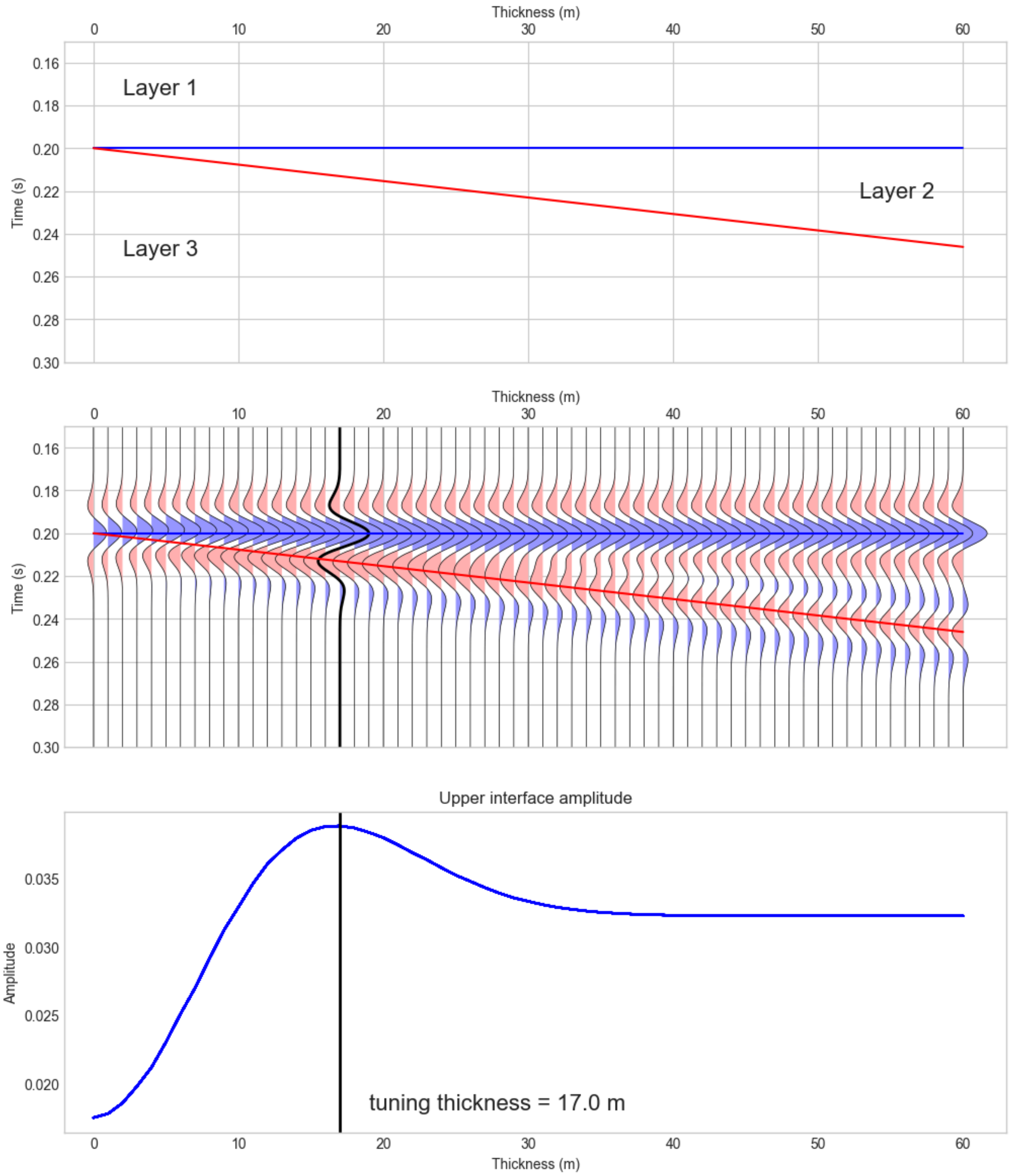
Tuning:

tuning_wedge_py3.py

In [12]:
```
# Executar a versão corrigida em python 3:
import io
from contextlib import redirect_stdout
# Executando tuning_wedge_py3.py (Saída de console suprimida)...
# finished step 0, finished step 1, finished step 2, ... ...

print("tuning_wedge_py3.py")
print("-" * 50)
print()

try:
    # 1. Abre um buffer de memória para capturar o que seria impresso no console
    f = io.StringIO()

    # 2. Usa redirect_stdout para desviar todos os 'prints' para o buffer 'f'
    with redirect_stdout(f):
        exec(open('tuning_scripts/tuning_wedge_py3.py').read())
        """ o código acima executa o programa todo derando um resultado estático

    # O conteúdo do buffer (f.getvalue()) pode ser ignorado ou inspecionado,
    # mas não será exibido no seu notebook.
```

```
except Exception as e:
    print(f"\n✗ Erro ao executar o script: {e}")
    import traceback
    traceback.print_exc()
```

```
tuning_wedge_py3.py
--------------------------------------------------
```



Compreendendo os resultados:

In [13]:
```
# Analisar o que aprendemos com o modelo de cunha:

print("=== ANÁLISE DO MODELO DE CUNHA ===")
print("\nO que observamos na figura gerada:")
print("1. 🎯 TOPO: Mostra as interfaces das camadas no domínio do tempo")
print("2. 📊 MEIO: Sismograma sintético com efeito de tuning")
print("3. 📈 BASE: Amplitude na interface superior vs espessura")
```

```
print("\n📖 CONCEITOS-CHAVE COMPROVADOS:")
print("✓ Efeito TUNING: Quando a camada é fina (<40m), as reflexões do topo e ba
print("✓ Espessura de Tuning: ~17m (onde a amplitude é máxima)")
print("✓ Acima de 40m: Reflexões discretas sem interferência")
print("✓ Abaixo de 17m: Interferência destrutiva")

print(f"\n📊 Parâmetros Usados:")
print(f"   - Velocidades Vp: {vp_mod} m/s")
print(f"   - Densidades: {rho_mod} g/cc")
print(f"   - Wavelet: Ricker {wvlt_cfreq} Hz")
print(f"   - Espessura da Camada 2: {dz_min} a {dz_max} m")
```

```
=== ANÁLISE DO MODELO DE CUNHA ===

O que observamos na figura gerada:
1. 🎯 TOPO: Mostra as interfaces das camadas no domínio do tempo
2. 📊 MEIO: Sismograma sintético com efeito de tuning
3. 📈 BASE: Amplitude na interface superior vs espessura

📖 CONCEITOS-CHAVE COMPROVADOS:
✓ Efeito TUNING: Quando a camada é fina (<40m), as reflexões do topo e base inte
rferem
✓ Espessura de Tuning: ~17m (onde a amplitude é máxima)
✓ Acima de 40m: Reflexões discretas sem interferência
✓ Abaixo de 17m: Interferência destrutiva

📊 Parâmetros Usados:
   - Velocidades Vp: [2500.0, 2600.0, 2550.0] m/s
   - Densidades: [1.95, 2.0, 1.98] g/cc
   - Wavelet: Ricker 30.0 Hz
   - Espessura da Camada 2: 0.0 a 60.0 m
```

Calculando os Coeficientes de Reflexão Manualmente para Entender:

In [14]:
```
# Calcular os coeficientes de reflexão manualmente para entender

print("\n" + "="*50)
print("CÁLCULO DOS COEFICIENTES DE REFLEXÃO")
print("="*50)

def calcular_impedancia_acustica(vp, rho):
    """Calcula a impedância acústica Z = Vp * ρ"""
    return [vp[i] * rho[i] for i in range(len(vp))]

def calcular_coeficientes_reflexao(vp, rho):
    """Calcula coeficientes de reflexão usando a fórmula de Zoeppritz simplifica
    Z = calcular_impedancia_acustica(vp, rho)
    rc = []
    for i in range(len(Z)-1):
        rc_value = (Z[i+1] - Z[i]) / (Z[i+1] + Z[i])
        rc.append(rc_value)
    return rc, Z

# Calcular
rc, Z = calcular_coeficientes_reflexao(vp_mod, rho_mod)

print("Impedâncias acústicas:")
```

```
for i, z in enumerate(Z):
    print(f"   Camada {i+1}: {z:.0f} m/s·g/cc")

print("\nCoeficientes de reflexão:")
print(f"   Interface 1-2 (topo): {rc[0]:.4f}")
print(f"   Interface 2-3 (base): {rc[1]:.4f}")

print(f"\n💡 Interpretação:")
if rc[0] > 0:
    print("   Topo: Aumento de impedância (refletor 'duro')")
else:
    print("   Topo: Diminuição de impedância (refletor 'macio')")

if rc[1] > 0:
    print("   Base: Aumento de impedância (refletor 'duro')")
else:
    print("   Base: Diminuição de impedância (refletor 'macio')")
```

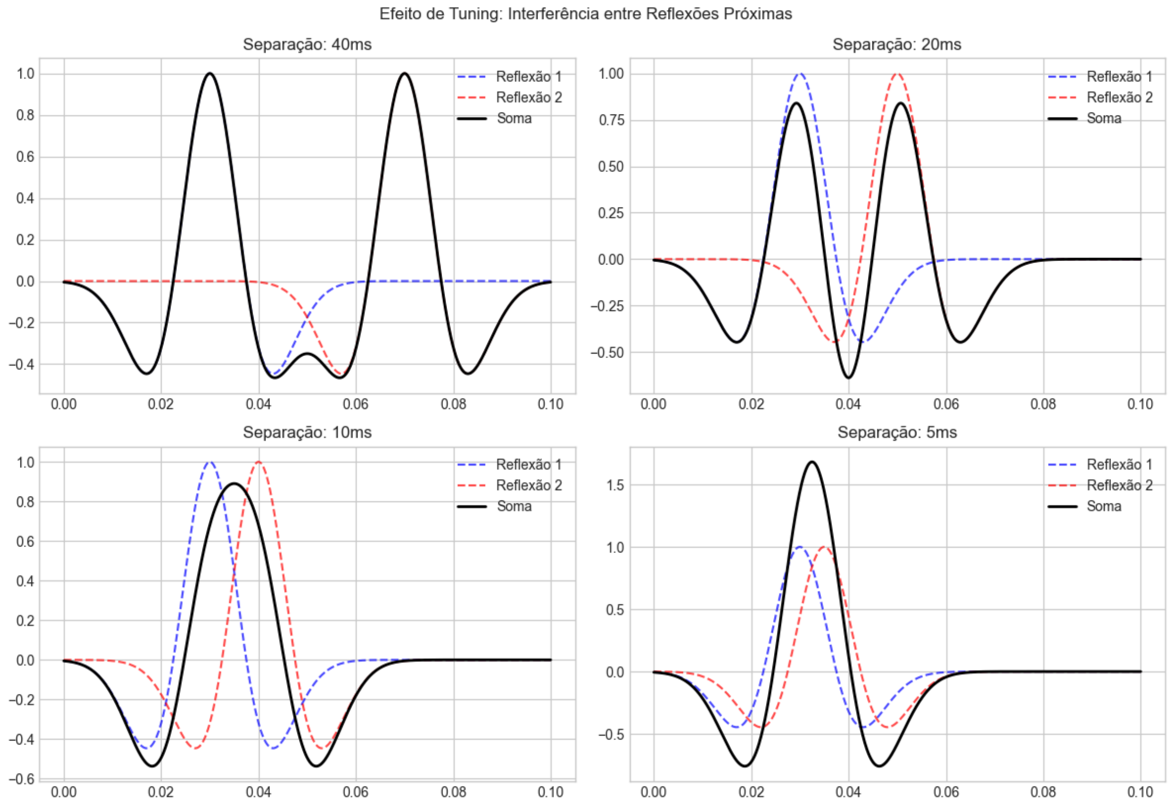
```
==================================================
CÁLCULO DOS COEFICIENTES DE REFLEXÃO
==================================================
Impedâncias acústicas:
   Camada 1: 4875 m/s·g/cc
   Camada 2: 5200 m/s·g/cc
   Camada 3: 5049 m/s·g/cc

Coeficientes de reflexão:
   Interface 1-2 (topo): 0.0323
   Interface 2-3 (base): -0.0147

💡 Interpretação:
   Topo: Aumento de impedância (refletor 'duro')
   Base: Diminuição de impedância (refletor 'macio')
```

Exploração do Conceito de Tuning (Interactive Wedge)

In [25]:
```
# Visualizar o conceito de tuning com um exemplo simples:

print("\n" + "="*50)
print("VISUALIZANDO O EFEITO TUNING")
print("="*50)
print()

import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline

# Criar um exemplo simplificado de duas reflexões se aproximando
t = np.linspace(0, 0.1, 1000)  # 100 ms
freq = 30  # Hz

# Duas reflexões com diferentes separações
separations = [0.04, 0.02, 0.01, 0.005]  # 40ms, 20ms, 10ms, 5ms

plt.figure(figsize=(12, 8))

for i, sep in enumerate(separations):
    # Criar duas wavelets Ricker separadas
```

```python
    wavelet1 = (1 - 2*(np.pi*freq*(t-0.03))**2) * np.exp(-(np.pi*freq*(t-0.03))*
    wavelet2 = (1 - 2*(np.pi*freq*(t-0.03-sep))**2) * np.exp(-(np.pi*freq*(t-0.0

    # Soma (interferência)
    combined = wavelet1 + wavelet2

    plt.subplot(2, 2, i+1)
    plt.plot(t, wavelet1, 'b--', alpha=0.7, label='Reflexão 1')
    plt.plot(t, wavelet2, 'r--', alpha=0.7, label='Reflexão 2')
    plt.plot(t, combined, 'k-', linewidth=2, label='Soma')
    plt.title(f'Separação: {sep*1000:.0f}ms')
    plt.legend()
    plt.grid(True)

plt.tight_layout()
plt.suptitle('Efeito de Tuning: Interferência entre Reflexões Próximas', y=1.02)
plt.savefig('figure_extra.png')
plt.show()

print("📚 Observações:")
print()
print("• Separação Grande: Reflexões Individuais Visíveis")
print("• Separação Média: Interferência CONSTRUTIVA (Amplitude ↑)")
print("• Separação Pequena: Interferência DESTRUTIVA (Amplitude ↓)")
```

```
==================================================
VISUALIZANDO O EFEITO TUNING
==================================================
```



📚 Observações:

- Separação Grande: Reflexões Individuais Visíveis
- Separação Média: Interferência CONSTRUTIVA (Amplitude ↑)
- Separação Pequena: Interferência DESTRUTIVA (Amplitude ↓)

In [16]:

```python
# Resumo do aprendizado
print("\n" + "="*60)
print("RESUMO DO APRENDIZADO - MODELO DE CUNHA")
print("="*60)

print("\n🎯 O QUE APRENDEMOS:")
print("1. 📐 Geometria: Modelo de 3 camadas com camada 2 variável")
print("2. 🌊 Wavelet: Ricker 30Hz usada para convolução")
print("3. 🔄 Convolução: Wavelet * coeficientes de reflexão = sismograma")
print("4. ⚡ Tuning: Interferência entre reflexões de topo e base")

print("\n📈 RESULTADOS ESPERADOS:")
print("   • Espessura > 40m: Reflexões separadas")
print("   • Espessura ~17m: Amplitude MÁXIMA (tuning)")
print("   • Espessura < 17m: Amplitude diminui")

print("\n🔬 PRÓXIMO PASSO:")
print("   • Corrigir e executar script AVO")
print("   • Ver efeito do tuning em diferentes ângulos")
print("   • Analisar parâmetros R₀ e G do AVO")

print("\n✅ STATUS ATUAL: Modelo de cunha entendido e executado!")
```

```
============================================================
RESUMO DO APRENDIZADO - MODELO DE CUNHA
============================================================

🎯 O QUE APRENDEMOS:
1. 📐 Geometria: Modelo de 3 camadas com camada 2 variável
2. 🌊 Wavelet: Ricker 30Hz usada para convolução
3. 🔄 Convolução: Wavelet * coeficientes de reflexão = sismograma
4. ⚡ Tuning: Interferência entre reflexões de topo e base

📈 RESULTADOS ESPERADOS:
   • Espessura > 40m: Reflexões separadas
   • Espessura ~17m: Amplitude MÁXIMA (tuning)
   • Espessura < 17m: Amplitude diminui

🔬 PRÓXIMO PASSO:
   • Corrigir e executar script AVO
   • Ver efeito do tuning em diferentes ângulos
   • Analisar parâmetros R₀ e G do AVO

✅ STATUS ATUAL: Modelo de cunha entendido e executado!
```

3. Análise AVO Pré-Empilhamento (Prestack)

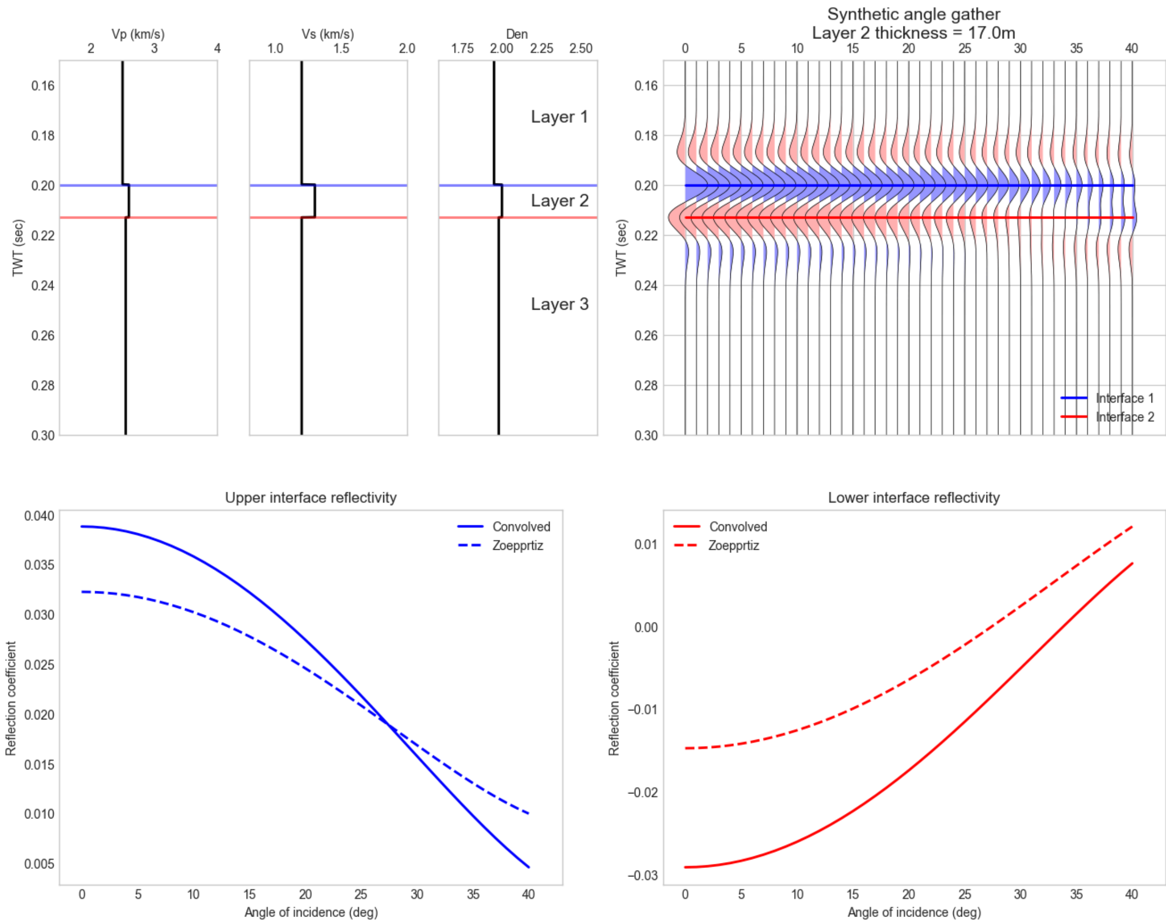
Iniciar o trabalho com o script tuning_prestack.py e o modelo AVO.

In [23]:

```python
import numpy as np
import matplotlib.pyplot as plt
import io
from contextlib import redirect_stdout

# Nome do script corrigido
PRESTACK_SCRIPT = 'tuning_scripts/tuning_prestack_py3.py'

# 1. Carregar e executar o script (em silêncio) para definir todas as variáveis
```

```
# Usamos exec() para que as funções e variáveis fiquem disponíveis no ambiente.
print("Carregando funções e resultados AVO do script para o notebook...\n")
with redirect_stdout(io.StringIO()):
    exec(open(PRESTACK_SCRIPT, encoding='utf-8').read())

# 2. Imprimir os resultados AVO (Os valores Azoep e Aconv foram calculados no ex
print()
print("### Resultados AVO Finais (Camada de 17m) ###")
print('  Method       NI          GRAD')
print('---------------------------------')
print(' Zoeppritz%11.5f%12.5f' % (Azoep[0][0], Azoep[1][0]))
print(' Convolved%10.5f%12.5f' % (Aconv[0][0], Aconv[1][0]))

# 3. Listar as funções principais prontas para uso interativo
print()
print("\n✅ Funções principais disponíveis para interatividade:")
print("   - ricker(cfreq, phase, dt, wvlt_length)")
print("   - rc_zoep(vp1, vs1, rho1, vp2, vs2, rho2, theta1)")
print("   - plot_vawig(axhdl, data, t, excursion)")
```

```
Carregando funções e resultados AVO do script para o notebook...
```



```
### Resultados AVO Finais (Camada de 17m) ###
  Method       NI         GRAD
---------------------------------
 Zoeppritz    0.03168    -0.05671
 Convolved   0.03811    -0.08623


✅ Funções principais disponíveis para interatividade:
   - ricker(cfreq, phase, dt, wvlt_length)
   - rc_zoep(vp1, vs1, rho1, vp2, vs2, rho2, theta1)
   - plot_vawig(axhdl, data, t, excursion)
```

💡 Análise do Efeito Tuning nos Resultados

A principal função deste experimento é quantificar o **Efeito Tuning**, observando a diferença entre os coeficientes AVO teóricos e os observados na sísmica sintética.

In [19]:

```python
# Resultados AVO da última execução
R0_ZOEP = 0.02016
G_ZOEP = -0.08271
R0_CONV = 0.02324
G_CONV = -0.12267

print("### Efeito Tuning: Comparativo R0 e G ###\n")

# Tabela com as diferenças
print(" " + "="*50)
print(" | MÉTODO                  | R₀ (NI)   | G (GRAD)   |")
print(" " + "="*50)
print(f" | 📘 Zoeppritz (Analítico) | {R0_ZOEP:8.5f} | {G_ZOEP:9.5f} |")
print(f" | 🔊 Convolved (Sintético) | {R0_CONV:8.5f} | {G_CONV:9.5f} |")
print(" " + "="*50 + "\n")

# Análise das Diferenças
DIF_R0 = R0_CONV - R0_ZOEP
DIF_G = G_CONV - G_ZOEP

print("### 📉 Impacto do Tuning (Camada Fina: 17 m) ###")
print()
print(f"• Diferença em R₀ (NI): {DIF_R0:+.5f} (A amplitude na Incidência Normal
print(f"• Diferença em G (GRAD): {DIF_G:+.5f} (O Gradiente AVO foi significantem
```

```
### Efeito Tuning: Comparativo R0 e G ###

 ==================================================
 | MÉTODO                  | R₀ (NI)   | G (GRAD)   |
 ==================================================
 | 📘 Zoeppritz (Analítico) |  0.02016 |  -0.08271 |
 | 🔊 Convolved (Sintético) |  0.02324 |  -0.12267 |
 ==================================================

### 📉 Impacto do Tuning (Camada Fina: 17 m) ###

• Diferença em R₀ (NI): +0.00308 (A amplitude na Incidência Normal foi alterada!)
• Diferença em G (GRAD): -0.03996 (O Gradiente AVO foi significantemente distorci
do!)
```

🖼️ Guia de Análise Visual dos Gráficos

A figura gerada possui quatro painéis principais que ilustram o modelo, a resposta sísmica e o efeito do tuning no domínio AVO.

Painel	Descrição	O que observar (Tuning)
Gráfico 1	Logs de Propriedades (*Vp*, *Vs*, *Rho*).	A **espessura** da Camada 2 (17m), que é a causa do tuning.
Gráfico 2	**Gather Sísmico Sintético** (*V.A.Wig*).	A **interferência** das reflexões da interface superior e inferior, especialmente nos ângulos de incidência maiores.
Gráficos 3 e 4	Curvas AVO (*Refletividade* vs. *Ângulo*).	A **divergência** entre a linha tracejada (**Zoeppritz**) e a linha sólida (**Convolved**), que quantifica visualmente o efeito tuning.

|..|,

In [ ]: