Parameterized inversion for frequency-dependent time delay between two dispersed signals

Bill Menke, October 15, 2025

The method described here can be used to determine parameters **m** in a parameterized travel time function $T(\omega, \mathbf{m})$, where ω is angular frequency, using observations of dispersed seismograms (time series) $A(t)$ and $B(t)$ from two neighboring stations. The method is based on minimizing the prediction error E using either a steepest-descent algorithm that utilizes the derivatives $\partial E/\partial m_i$ or a Newton method algorithm that uses both $\partial E/\partial m_i$ and $\partial^2 E/\partial m_i\, \partial m_j$. Analytic formula for these derivatives are derived using frequency-domain techniques.

This method is applicable to the problem of determining the dispersive velocity function $v(\omega)$, because travel time and velocity are related by $T(\omega) = \Delta x/v(\omega)$, where Δx is the distance between stations projected onto the direction of propagation. However, in some scenarios, Δx is initially unknown because the direction of propagation is unknown, so an inversion using $T(\omega)$ is preferred. One such case is when travel times between two pairs of stations in a triangular array are used to solve for the direction of propagation.

Derivation

Consider real time series (seismograms) $A(t)$ and $B(t)$ with Fourier transforms $\tilde{A}(\omega)$ and $\tilde{B}(\omega)$, respectively. A real phase delay filter $f(t)$ has Fourier transform $\tilde{f}(\omega) = \exp\big(-i\omega T(\omega, \mathbf{m})\big)$ with **m** unknown. Note that $\tilde{f}_R(\omega) = \cos\big(-\omega T(\omega, \mathbf{m})\big)$ and $\tilde{f}_I(\omega) = \sin\big(-\omega T(\omega, \mathbf{m})\big)$. The frequency-domain filter has unit amplitude; that is, $\left|\tilde{f}(\omega)\right|^2 = \tilde{f}(\omega)\tilde{f}^*(\omega) = 1$. For $f(t)$ to be real, $\tilde{f}(-\omega) = \tilde{f}^*(\omega)$ so $T(-\omega, \mathbf{m}) = T(\omega, \mathbf{m})$.

The model that we consider is when $B(t)$ is a dispersively-delayed version of $A(t)$; that is:

$$B(t) = A(t) * f(t) \tag{1}$$

where * signifies convolution. The L_2 prediction error is defined as:

$$E \equiv \int [B(t) - A(t) * f(t)]^2 \; dt = \int [A(t) * f(t)]^2 \; dt + \int B^2(t) \; dt - 2 \int [A(t) * f(t)]\, B(t) \; dt \tag{2}$$

Parseval's theorem for real time series $a(t)$ and $b(t)$ states:

$$\int_{-\infty}^{+\infty} a(t) b(t) \; dt = \frac{1}{2\pi} \int_{-\infty}^{+\infty} \tilde{a}(\omega) \tilde{b}^*(\omega) \; d\omega \tag{3}$$

A real function, say $h(t)$, has a Fourier transform with symmetry $\tilde{h}(-\omega) = \tilde{h}^*(\omega)$. Consequently:

$$\int_{-\infty}^{+\infty} h(\omega)\ d\omega = \int_{0}^{+\infty} h(\omega)\, d\omega + \int_{-\infty}^{0} h(\omega)\, d\omega = \int_{0}^{+\infty} h(\omega)\, d\omega + \int_{+\infty}^{0} h(-\omega)\, d(-\omega) =$$

$$= \int_{0}^{+\infty} h(\omega)\, d\omega + \int_{0}^{+\infty} h^*(\omega)\, d\omega = 2\int_{0}^{+\infty} \mathrm{Re}\, h(\omega)\, d\omega$$

(4)

Note that in a discrete implementation in which the integrals are approximated by a Reiman sum, the zero-frequency value is being counted once in the $(-\infty, +\infty)$ summation but twice in the $(-\infty, 0)$ plus $(0, +\infty)$ summations. Thus, $h(\omega = 0)$ must be subtracted from the latter to match the former within computer codes that implement the algorithm.

Applying these results:

$$\int_{-\infty}^{+\infty} [A(t) * f(t)]^2\, dt = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \tilde{A}(\omega)\tilde{A}^*(\omega)\,\tilde{f}(\omega)\tilde{f}^*(\omega) d\omega =$$

$$\frac{1}{2\pi}\int_{-\infty}^{+\infty} |\tilde{A}(\omega)|^2\, d\omega = \frac{1}{\pi}\int_{0}^{+\infty} |\tilde{A}(\omega)|^2\, d\omega$$

$$\int_{-\infty}^{+\infty} [B(t)]^2\, dt = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \tilde{B}(\omega)\tilde{B}^*(\omega)\, d\omega =$$

$$\frac{1}{2\pi}\int_{-\infty}^{+\infty} \left|\tilde{B}(\omega)\right|^2 d\omega = \frac{1}{\pi}\int_{0}^{+\infty} \left|\tilde{B}(\omega)\right|^2 d\omega$$

$$\int_{-\infty}^{+\infty} [A(t) * f(t)]\, B(t)\, dt = \frac{1}{\pi}\int_{0}^{+\infty} \mathrm{Re}\left[\tilde{A}(\omega)\tilde{B}^*(\omega)\tilde{f}(\omega)\right] d\omega$$

(5)

The product of three complex numbers has real and imaginary parts

$$(a + ib)(c + id)(e + if) = (ace - bde - adf - bcf) + i(ade + bce + acf - bdf)$$

(6)

so

$$\mathrm{Re}\left[\tilde{A}(\omega)\tilde{B}^*(\omega)\tilde{f}(\omega)\right] = \tilde{A}_R\tilde{B}_R^*\tilde{f}_R - \tilde{A}_I\tilde{B}_I^*\tilde{f}_R - \tilde{A}_R\tilde{B}_I^*\tilde{f}_I - \tilde{A}_I\tilde{B}_R^*\tilde{f}_I$$

$$= \left(\tilde{A}_R\tilde{B}_R + \tilde{A}_I\tilde{B}_I\right)\tilde{f}_R + \left(\tilde{A}_R\tilde{B}_I - \tilde{A}_I\tilde{B}_R\right)\tilde{f}_I$$

(7)

We now differentiate E with respect to m_i. As $\tilde{A}(\omega)$ and $\tilde{B}(\omega)$ are not functions of m_i,

$$\frac{\partial E}{\partial m_i} = -2\frac{\partial}{\partial m_i}\int_{-\infty}^{+\infty}[A(t) * f(t)]\,B(t)\,dt =$$

$$-2\frac{1}{2\pi}\int_{-\infty}^{+\infty}\frac{\partial}{\partial m_i}\left[\tilde{A}(\omega)\tilde{B}^*(\omega)\tilde{f}(\omega)\right]d\omega =$$

$$-2\frac{1}{2\pi}2\int_{0}^{+\infty}\frac{\partial}{\partial m_i}\mathrm{Re}\left[\tilde{A}(\omega)\tilde{B}^*(\omega)\tilde{f}(\omega)\right]d\omega =$$

$$-\frac{2}{\pi}\int_{0}^{+\infty}\left(\tilde{A}_R\tilde{B}_R + \tilde{A}_I\tilde{B}_I\right)\frac{\partial\tilde{f}_R}{\partial m_i}d\omega - \frac{2}{\pi}\int_{0}^{+\infty}\left(\tilde{A}_R\tilde{B}_I - \tilde{A}_I\tilde{B}_R\right)\frac{\partial\tilde{f}_I}{\partial m_i}d\omega$$

(8)

Similarly, the second derivative is:

$$\frac{\partial E^2}{\partial m_i\partial m_j} = -\frac{2}{\pi}\int_{0}^{+\infty}\left(\tilde{A}_R\tilde{B}_R + \tilde{A}_I\tilde{B}_I\right)\frac{\partial^2\tilde{f}_R}{\partial m_i\partial m_j}d\omega - \frac{2}{\pi}\int_{0}^{+\infty}\left(\tilde{A}_R\tilde{B}_I - \tilde{A}_I\tilde{B}_R\right)\frac{\partial\tilde{f}_I}{\partial m_i\partial m_j}d\omega$$

(9)

By applying the chain rule, we find that the first derivative is:

$$\frac{\partial\tilde{f}_R}{\partial m_i} = \frac{\partial}{\partial m_i}\cos\big(-\omega T(\omega,\mathbf{m})\big) = \omega\sin(-\omega T)\frac{\partial T}{\partial m_i} = \omega\tilde{f}_I(\omega)\frac{\partial T}{\partial m_i}$$

$$\frac{\partial\tilde{f}_I}{\partial m_i} = \frac{\partial}{\partial m_i}\sin\big(-\omega T(\omega,\mathbf{m})\big) = -\omega\cos(-\omega T)\frac{\partial T}{\partial m_i} = -\omega\tilde{f}_R(\omega)\frac{\partial T}{\partial m_i}$$

(10)

and that the second derivative is:

$$\frac{\partial^2\tilde{f}_R}{\partial m_i\partial m_j} = \omega\frac{\partial}{\partial m_j}\left[\tilde{f}_I(\omega)\frac{\partial T}{\partial m_i}\right] = \omega\frac{\partial\tilde{f}_I(\omega)}{\partial m_j}\frac{\partial T}{\partial m_i} + \omega\tilde{f}_I(\omega)\frac{\partial^2 T}{\partial m_i\partial m_j}$$

$$\frac{\partial^2\tilde{f}_I}{\partial m_i\partial m_j} = -\omega\frac{\partial}{\partial m_j}\left[\tilde{f}_R(\omega)\frac{\partial T}{\partial m_i}\right] = -\omega\frac{\partial\tilde{f}_R(\omega)}{\partial m_j}\frac{\partial T}{\partial m_i} - \omega\tilde{f}_R(\omega)\frac{\partial^2 T}{\partial m_i\partial m_j}$$

(11)

In the special case of the linear model $T(\omega_k) = \sum_n G_{kn} m_n$:

$$\left.\frac{\partial T}{\partial m_i}\right|_{\omega_k} = G_{ki} \quad \text{and} \quad \left.\frac{\partial^2 T}{\partial m_i \partial m_j}\right|_{\omega_k} = 0$$

(12)

Eqns. (2)-(11) have been verified by numerically calculation. Only the first derivatives (Eqns. 8 and 10) are needed for a steepest-descent inversion for **m** (Menke, Geophysical Data Analysis, 2024, Sec. 11.9). For a Newton's method inversion (Menke, Geophysical Data Analysis, 2024, Eqn. 11.76), the second derivatives (Eqns. 9 and 11), are needed as well. Another use of the second derivative is to compute the covariance of the estimated **m** using the relationship (Menke, Geophysical Data Analysis, 2024, Eqn. 4.76):

$$\text{cov}(\mathbf{m}) = 2\sigma_{\tilde{B}}^2 \mathbf{D}^{-1} \text{ with } D_{ij} = \left.\frac{\partial E^2}{\partial m_i \partial m_j}\right|_{\mathbf{m}^{est}}$$

(13)

Here, $\sigma_{\tilde{B}}^2$ is posterior variance of the data. It is approximately

$$\sigma_{\tilde{B}}^2 = \frac{E_0}{(2N_\omega - M)}$$

(14)

Here, N_ω is the number of non-negative frequencies in the discrete version of $\tilde{B}(\omega)$ and M is the number of model parameters. The factor of two arises because the data are complex; both real and imaginary parts are noisy.

Computational efficiency can be gained by limiting the frequency-integrals to the $(0, \omega_2)$, where ω_2 is the largest significant frequency in the data.

Numerical Test

Two exemplary dispersed seismograms (time series), $A(t)$ and $B(t)$, for stations with source-receiver ranges of 5000 km and 5100 km, respectively, are calculated with Fourier methods (Fig. 1). The phase velocity decreases from 4.0 km/s at $\omega_1 = 0.01 \times 2\pi$ rad/s to 3.5 km/s at $\omega_2 = 0.09 \times 2\pi$ rad/s (values typical for Rayleigh waves). A Gaussian source time function is used, with a standard deviation chosen so that the seismograms have little energy at frequencies above ω_2.

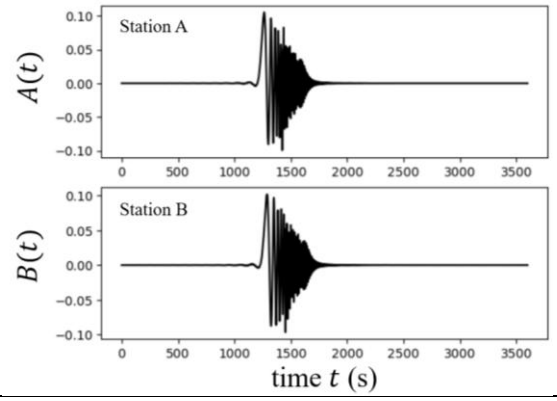

Fig. 1. Two dispersive seismograms, $A(t)$ and $B(t)$, for stations 5000 and 5100 km, respectively, from a hypothetical earthquake.

The travel time function is parameterized as the linear function $T(\omega, m_1, m_2) = m_1 + m_2\omega$, which varies linearly with angular frequency ω with intercept m_1 and slope m_2 (Fig. 2).

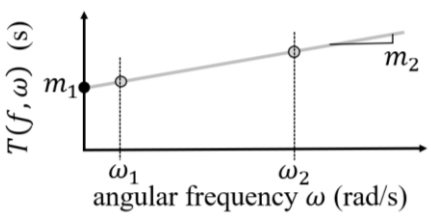

Fig. 2. The travel time function $T(\omega, m_1, m_2)$ is linear in angular frequency ω and is parameterized with intercept m_1 and slope m_2. The seismogram source has little energy outside of the (ω_1, ω_2) interval.

The error E and its derivatives are calculated for $\mathbf{m}_0 = [23.80,1.0]^T$ by the frequency-domain method described in the text and a time domain method that calculated derivatives using finite differences. The quantities computed by the two methods are in close agreement (Table 1).

Table 1. Comparison of frequency-domain and time-domain values.

Domain	E	$\partial E/\partial m_1$	$\partial E/\partial m_2$	$\partial^2 E/\partial m_1^2$	$\partial^2 E/\partial m_1 \partial m_2$	$\partial^2 E/\partial m_2^2$
frequency	0.4081	-0.2202	-0.0777	0.0351	0.0020	-0.0035
time	0.4081	-0.2202	-0.0777	0.0351	0.0020	-0.0035

The starting solution $\mathbf{m}_0 = [23.89,5.00]^T$ roughly aligns the seismograms. An estimate solution $\mathbf{m}^{est} = [24.74,6.16]^T$ is calculated using a gradient-descent inversion that employs only first derivatives. In the test shown, 60 iterations achieve an excellent match between the seismograms (Fig. 3), with an error reduction of 97%. A Python implementation executes in about 35 ms on a notebook computer (for $\omega_2 = 0.2 \times 2\pi$ rad/s). A Newton's method solution achieves a similar error reduction in only 6 iterations; however, as more computational effort is required per iteration, the reduction in execution time, to 23 ms, is not proportional.

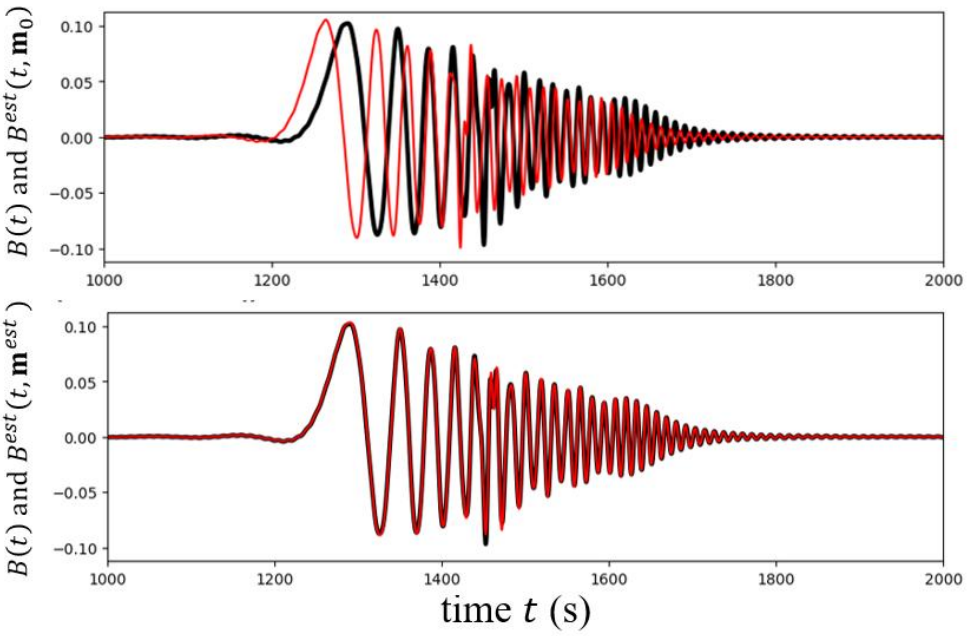

Fig. 3. Results of gradient-descent method. (Top) The observed seismogram $B(t)$ (black) and predicted seismogram $B^{pre}(t, \mathbf{m}_0)$ (black) are only poorly aligned for the initial choice $\mathbf{m}_0$. (Bottom) The alignment is much improved with the final estimate $B^{pre}(t, \mathbf{m}^{est})$.

Python code

```
# wave function derivative (wf) functions
# modify wfTandderivs() to reflect on your implementation of T(w,m)
# (second derivative not needed for inversion)
# decide on seismograms A, B
# they optionally can be generated with wffouriersetup()
# set up by calling wfderivssetup()
# decide upon starting model (must be close enough to avoid cycle skips)
# decide upon gradient descent parameters
# wfinvert() to get solution
# optionally, compute covariance with wfcovariance()

def wfsynthetic( N, Dt, dist ):
    # makes a dispersed seismogram for test purposes
    # hardwired with a particular v(f)
    # N: number of points in synthetic (I use N=3600*100)
    # Dt: time increment (I use Dt=0.01);
    # dist: source-receiver distance in km

    # standard frequency setup
    Nsave = N;
    N = 2*int(N/2);
    if( N<Nsave ):
        N=N+1;
    No2 = int(N/2);
    fmax=1/(2.0*Dt);
    Df=fmax/No2;      # frequency sampling
    Nf=No2+1;
    f = np.zeros((N,1));
```

```
    f[0:N,0] = Df*np.concatenate(
    (np.linspace(0,No2,Nf),np.linspace(-No2+1,-1,No2-1)), axis=0 );
    Dw=2*pi*Df;
    w=2*pi*f;
    Nw=Nf;
    fpos = np.zeros((Nf,1));
    fpos[0:Nf,0] = Df * np.linspace(0,No2,Nf); # non-neg f's
    wpos=2*pi*fpos;   # non-negative angular frequencies

    # time and source pulse
    t = np.zeros((N,1));
    t[0:N,0] = Dt*np.linspace(0,N-1,N);
    Ns = 20000;
    t0 = Dt*(Ns-1);
    sd0 = 3.0;
    u0 = np.exp( -0.5*np.power(t-t0*np.ones((N,1)),2) / (sd0**2) );
    u0t = np.fft.fft(u0,axis=0);

    # phase velocity
    v = np.zeros((N,1));
    f1 = 0.01; v1=4.0;
    f2 = 0.09; v2=3.5;
    for i in range(N):
        fa = abs(f[i,0]);
        if( fa<f1 ):
            v[i,0]=v1
        elif( fa>f2 ):
            v[i,0]=v2;
        else:
            v[i,0]= v1 + (fa-f1)*(v2-v1)/(f2-f1);
    s = np.reciprocal(v);
    ws = np.multiply(w,s);

    # disperse pulse and shif to
    ph1 = np.exp( complex(0.0,-1.0)*(ws*dist) );
    u1t = np.multiply( u0t, ph1 );
    u1 = np.real( np.fft.ifft(u1t,axis=0) );
    u1 = np.roll( u1, -Ns, axis=0 );

    if( N>Nsave ):
        u1 = u1[0:Nsave,0:1];
    A = sqrt(1000.0/dist);
    return(A*u1);

def wffouriersetup( A, B, Dt, f0 ):
    # A, B: timeseries
    # Dt: time sampling
    # f0: maximum frequency to use in fitting
    # primary quantities
    N, i = np.shape(A);
    fmax = 1.0/(2.0*Dt);
    Df = fmax/(N/2);
    Dw = 2*pi*Df;
    Nf = int( f0/Df );
    wpos = 2.0*pi*eda_cvec( Df*np.linspace(0,Nf-1,Nf) );
    At = Dt*np.fft.fft( A, axis=0 );
    Bt = Dt*np.fft.fft( B, axis=0 );
    Atpos = At[0:Nf,0:1];   # need
```

```
    Btpos = Bt[0:Nf,0:1];   # need
    return Dw, wpos, Atpos, Btpos;

def wfderivssetup( wpos, Atpos, Btpos ):
    # wpos: non-negative angular frequencies, must start at 0, can
    #       end before Nyquist
    # Apos, Bpos: non-negative Fourier components of A(t), B(t)
    #       at angular frequencies wpos
    # wffouriersetup() can calculate these quantities
    Dw = wpos[1,0]-wpos[0,0];
    AtposR = np.real(Atpos);
    AtposI = np.imag(Atpos);
    BtposR = np.real(Btpos);
    BtposI = np.imag(Btpos);
    Atabs2 = np.power(AtposR,2) + np.power(AtposI,2);
    Btabs2 = np.power(BtposR,2) + np.power(BtposI,2);
    E0A = (1.0/pi)*Dw*( np.sum(Atabs2) - 0.5*Atabs2[0,0]);
    E0B = (1.0/pi)*Dw*( np.sum(Btabs2) - 0.5*Btabs2[0,0]);
    XA = np.multiply(AtposR,BtposR)+np.multiply(AtposI,BtposI);
    XB = np.multiply(AtposR,BtposI)-np.multiply(AtposI,BtposR);
    return E0A, E0B, XA, XB

def wfTandderivs(wpos, m, dosecond):
    # implements T(w,m), dT/dm and d2T/dm2
    # computes and returns d2T/dm2 only when dosecond is True
    Nf,i = np.shape(wpos);
    M,i = np.shape(m);
    # must be changed to reflect choice of T(w,m)
    Tpos = m[0,0]*np.ones((Nf,1))+m[1,0]*wpos;
    # note that first derivative is G zero when T=Gm and G independent of m
    dTdmi = np.zeros((Nf,M));
    dTdmi[0:Nf,0:1] = np.ones((Nf,1));
    dTdmi[0:Nf,1:2] = wpos;
    if dosecond :
        # note that secong derivative is identically zero
        # when T=Gm and G independent of m
        d2Tdmi2 = np.zeros((Nf,M,M)); # is identically zero in this case
                                      # but in general, not zero
        return Tpos, dTdmi, d2Tdmi2;
    else:
        return Tpos, dTdmi;

def wfderivs( wpos, Atpos, Btpos, E0A, E0B, XA, XB, m, dosecond ):

    # derivates with respect to model parameters
    M, i = np.shape(m);
    Nf, i = np.shape(wpos);
    Dw = wpos[1,0]-wpos[0,0];
    dEdm = np.zeros((M,1));

    if dosecond:
        Tpos, dTdmi, d2Tdmij = wfTandderivs(wpos,m,dosecond);
    else:
        Tpos, dTdmi = wfTandderivs(wpos,m,dosecond);

    # deoends on mi
    wTpos = np.multiply(wpos,Tpos);
    ftposR = np.cos( -wTpos );
```

```
    ftposI = np.sin( -wTpos );
    ftpos = ftposR+complex(0.0,1.0)*ftposI;
    wftposR = np.multiply(wpos,ftposR);
    wftposI = np.multiply(wpos,ftposI);
    V1 = np.real( np.multiply( np.multiply(Atpos,np.conjugate(Btpos)), ftpos ) );
    E0C = (2.0/pi)*Dw*(np.sum(V1)-0.5*V1[0,0]);
    E0 = E0A+E0B-E0C;

    # do for each mi
    for i in range(M):
        dfRdm =  np.multiply( wftposI, dTdmi[0:Nf,i:i+1]);
        dfIdm = -np.multiply( wftposR, dTdmi[0:Nf,i:i+1]);
        IA = np.multiply( XA, dfRdm );
        IB = np.multiply( XB, dfIdm );
        IC = IA+IB;
        dEdm[i,0] = -(2/pi)*(Dw*np.sum(IC) - 0.5*Dw*IC[0,0]);
    if( dosecond ):
        d2Edm2 = np.zeros((M,M));
        for i in range(M):
            dfRdmi =  np.multiply( wftposI, dTdmi[0:Nf,i:i+1]);
            dfIdmi = -np.multiply( wftposR, dTdmi[0:Nf,i:i+1]);
            for j in range(i,M):
                dfRdmj =  np.multiply( wftposI, dTdmi[0:Nf,j:j+1]);
                dfIdmj = -np.multiply( wftposR, dTdmi[0:Nf,j:j+1]);
                D2ijR = np.multiply(wpos,np.multiply(dfIdmj,dTdmi[0:Nf,i:i+1]));
                D2ijR = D2ijR + np.multiply(wftposI,d2Tdmij[0:Nf,i:i+1,j]);
                D2ijI = np.multiply(-wpos,np.multiply(dfRdmj,dTdmi[0:Nf,i:i+1]));
                D2ijI = D2ijI - np.multiply(wftposR,d2Tdmij[0:Nf,i:i+1,j]);
                IA = np.multiply( XA, D2ijR );
                IB = np.multiply( XB, D2ijI );
                IC = IA+IB;
                D2 = -(2/pi)*(Dw*np.sum(IC) - 0.5*Dw*IC[0,0]);
                d2Edm2[i,j] = D2;
                d2Edm2[j,i] = D2;
        return E0, Tpos, dEdm, d2Edm2;
    else:
        return E0, Tpos, dEdm;

def wfinvert(wpos, Atpos, Btpos, E0A, E0B, XA, XB, mstart, dm, Niter, DEstagnate):
    # This version uses gradient descent, utilizing only dEdm
    # wpos, Atpos, Btpos, E0A, E0B, XA, XB from wfderivssetup()
    # mstart: staring guess for solution
    # dm: starting step size (I used 1.0);
    # Niter: maximum number of iterations (I used 100);
    # DEstagnate: change in relative error below which
    #    iterations aer terminated (I used 1e-3)

    # gradient descent constants
    mo = np.copy(mstart)
    alpha = dm;
    c1 = 0.0001;
    c2 = 0.9;
    tau = 0.5;

    # setup for iteration
    Eo, Tposo, dEdmo  = wfderivs( wpos, Atpos, Btpos, E0A, E0B, XA, XB, mo, False );
    Estart = Eo;
    Tposstart = np.copy(Tposo)
```

```
    count=1;
    for k in range(Niter):
        v = - dEdmo / sqrt( np.matmul( dEdmo.T, dEdmo )[0,0] );

        # backstep
        for kk in range(10):
            mg = mo + alpha*v;
            Eg,Tposg,dEdmg = wfderivs( wpos, Atpos, Btpos, E0A, E0B, XA, XB, mg, False
);
            count = count+1;
            if( Eg <= (Eo + c1*alpha*np.matmul(v.T,dEdmo)) ):
                break;
            alpha = tau*alpha;

        # reduction in error during this iteration
        DE = (Eo-Eg)/Eg;

        # update
        mo=mg;
        Eo = Eg;
        dEdmo = dEdmg;
        Tposo = Tposg;

        if( (DE>=0.0) and (DE<DEstagnate) ):
            break; # terminate iterations when change in solution is sufficiently
small
    return( Tposstart, Estart, mo, Tposo, Eo, count );

def wfinvert2(wpos, Atpos, Btpos, E0A, E0B, XA, XB, mstart, dm, Niter, DEstagnate):
    # This version uses Newton's method, utilizing both dEdm and d2Edm2
    # wpos, Atpos, Btpos, E0A, E0B, XA, XB from wfderivssetup()
    # mstart: staring guess for solution
    # dm: starting step size (I used 1.0);
    # Niter: maximum number of iterations (I used 100);
    # DEstagnate: change in relative error below which
    #    iterations aer terminated (I used 1e-3)

    # gradient descent constants
    mo = np.copy(mstart)

    # setup for iteration
    Eo, Tposo, dEdmo,  d2Edm2o = wfderivs( wpos, Atpos, Btpos, E0A, E0B, XA, XB, mo,
True );
    Estart = Eo;
    Elast = Eo;
    Tposstart = np.copy(Tposo)
    count=1;
    for k in range(Niter):

        dm = -la.solve( d2Edm2o, dEdmo );
        mo = mo + dm;
        Eo, Tposo, dEdmo,  d2Edm2o = wfderivs( wpos, Atpos, Btpos, E0A, E0B, XA, XB,
mo, True );
        count=count+1;

        # reduction in error during this iteration
        DE = (Elast-Eo)/Elast;
        if( k>3 ): # do at least 3 iterations
```

```
            if( (DE>=0.0) and (DE<DEstagnate) ):
                break; # terminate iterations when change in solution is sufficiently
small
        Elast = Eo;
    return( Tposstart, Estart, mo, Tposo, Eo, count );

def wfpredictedB( A, B, Dt, mo):
# compute predicted seismogram Bpre
    N, i = np.shape(A);
    fmax = 1.0/(2.0*Dt);
    Df = fmax/(N/2);
    Nf = int(N/2)+1;
    wpos = 2.0*pi*eda_cvec( Df*np.linspace(0,Nf-1,Nf) );  # need
    Tpos, dTdmi = wfTandderivs(wpos,mo,False);
    w = np.concatenate( (wpos, -np.flip(wpos[1:Nf-1],axis=0)), axis=0 );
    T = np.concatenate( (Tpos, np.flip(Tpos[1:Nf-1],axis=0)), axis=0 );
    ftp = np.exp( -complex(0.0,1.0)*np.multiply(w,T) );
    At = np.fft.fft(A,axis=0);
    Bpre = np.real( np.fft.ifft(np.multiply(At,ftp),axis=0) );
    return Bpre;

def wfcovariance(wpos, Atpos, Btpos, E0A, E0B, XA, XB, mo):

    E0, Tpos, dEdm, d2Edm2 = wfderivs( wpos, Atpos, Btpos, E0A, E0B, XA, XB, mo, True
);

    Nf, i = np.shape(Btpos);
    M, i = np.shape(mo);
    Dw = wpos[1,0]-wpos[0,0];

    # E0*Dw is the sum of squares of frequency values in Btpos-Atpos*ftpos
    # 2Nf is the number of real and imaginary parts
    # so variance is E0/Dw divided by degrees of freedome 2Nf-M
    sigmaB2 = E0/(2*Nf-M);

    # formula for covariance from Menke 2024 Eqn 4.76
    covm = 2.0*sigmaB2*la.inv(d2Edm2);

    return sigmaB2, covm, E0;
```