Source code for particleman.filter

```
# -*- coding: utf-8 -*-
"""
Time-frequency filtering for seismic waves.

To estimate propagation azimuth for a prograde/retrograde Rayleigh wavetrain:

1. Make complex component s-transforms, Sn, Se, Sv
2. Apply phase delay to Sv for normal retrograde motion, advance for prograde motion
3. Convert complex transforms into MxNx2 real ("vector") arrays
4. Calculate theta from theta_r and theta_l

To make a normalized inner product (NIP) filter, given azimuth

5. Rotate Sn, Se through theta, into radial (Sr) and transverse (St) component transforms
6. Calculate NIP using the vector Sr and phase-shifted vector Sv
7. Extract surface waves using NIP >= 0.8 (smoothed) as a filter on Sr, St, Sv, and invert to time dom
   Rayleigh waves will be on the time-domain radial, Love on the time-domain transverse.


References
----------

Meza-Fajardo, K. C., Papageorgiou, A. S., and Semblat, J. F. (2015).
Identification and Extraction of Surface Waves from Three Component Seismograms
Based on the Normalized Inner Product. Bulletin of the Seismological Society of
America.

"""
from functools import lru_cache
import warnings
# TODO: consider moving the nip argument to inside get_filter.  People may only
#   rarely want to see the NIP, and they can do so directly, using NIP.
import numpy as np


from .core import stransform, istransform

def get_shift(polarization):                                                    [docs]
    """
    Get the appropriate pi/2 phase advance or delay for the provided
    polarization.

    Parameters
    ----------
    polarization : str, {'retrograde', 'prograde', 'linear'}
        'retrograde' returns i, for a pi/2 phase advance.
        'prograde' or 'linear' returns -i, for a pi/2 phase delay

    Returns
    -------
    numpy.complex128
        Multiply this value (i or -i) into a complex vertical S-transform to
        shift its phase.

    """
    if polarization is 'retrograde':
        # phase advance
        shft = np.array(1j)
    elif polarization in ('prograde', 'linear'):
        # phase delay
        shft = -np.array(1j)
    else:
        raise ValueError("Polarization must be either 'prograde', 'retrograde', or 'linear'")

    return shft
```

```
def shift_phase(Sv, polarization):                                    [docs]
    """
    Phase-shift an s-transform by the appropriate phase shift for
    prograde/retrograde motion.

    Shift is done on a complex MxN array by multiplication with either i or -i
    (imaginary unit).  This is mostly a reference for how to do/interpret phase
    shifts, as it's such a simple thing to do outside of a function.

    Parameters
    ----------
    Sv : numpy.ndarray (complex, rank 2)
    polarization : str, {'retrograde', 'prograde', 'linear'}
        'retrograde' will apply a pi/2 phase advance (normal Rayleigh waves)
        'prograde' or 'linear' will apply a pi/2 phase delay

    Returns
    -------
    numpy.ndarray (real, rank 2)

    References
    ----------
    Pages 5 and 10 from Meta-Fajardo et al. (2015)

    """
    shft = get_shift(polarization)

    return Sv * shft



def xpr(az):                                                          [docs]
    """
    Get the Meza-Fajardo "xpr" sense-of-propagation of wavefield.
    propagation azimuth.

    Parameters
    ----------
    az : int or float
        Propagation direction in degrees.

    Returns
    -------
    1 for eastward propagation
    -1 for westward

    Notes
    -----
    If the azimuth is 0 or 180, polarization type may be ambiguous.

    """
    return int(np.sign(np.sin(np.radians(az))))



def instantaneous_azimuth(Sv, Sn, Se, polarization, xpr):             [docs]
    """
    Get instantaneous propagation angle [degrees], under the Rayleigh wave
    assumption, using the Meza-Fajardo et al. Normalized Inner Product criterion.

    Parameters
    ----------
    Sv, Sn, Se : numpy.ndarray (complex, rank 2)
        The vertical, North, and East component equal-sized complex s-transforms.
    polarization : str, {'retrograde', 'prograde', 'linear'}
        'retrograde' will apply a pi/2 phase advance.
        'prograde' or 'linear' will apply a pi/2 phase delay
    xpr : int
        Sense of propagation.  1 for eastward, -1 for westward.
        Try -int(np.sign(np.sin(np.radians(baz)))), unless they're directly N-S
        from each other.

    Returns
    -------
    az : numpy.ndarray (real, rank 2)
        Instantaneous Rayleigh wave propagation angle [degrees]

    References
    ----------
    Equations (19), (20), and (21) from Meza-Fajardo et al. (2015)
```

```
    "Let us emphatically note that if the sense of propagation of the phase
    under investigation is not established, prograde or retrograde motion
    cannot be defined without ambiguity."

    """
    Svhat = shift_phase(Sv, polarization)

    num = (Se.real * Svhat.real) + (Se.imag * Svhat.imag)
    denom = (Sn.real * Svhat.real) + (Sn.imag * Svhat.imag)

    # zeros will become nans, then propate in bad ways.
    # put a tiny number in the denominator where both the numberator and denominator are zero.
    denom[ np.logical_and(num == 0.0, denom == 0.0) ] = np.finfo(float).eps

    theta_r = np.arctan(num / denom) # [-pi/2, pi/2]
    # theta_r = np.arctan2(num, denom)

    theta_I = theta_r + np.pi*(1 - np.sign(np.sin(theta_r))) + \
        np.pi*(1 - np.sign(np.cos(theta_r))) * np.sign(np.sin(theta_r))/2

    theta = theta_I + (np.pi/2)*(np.sign(np.sin(theta_I)) - np.sign(xpr))

    return np.degrees(theta)



def scalar_azimuth(e, n, vhat):                                                [docs]
    """
    Time domain estimation the scalar/average azimuth, in degrees.

    References
    ----------
    Equations (10) and (12) from Meza-Fajardo et al. (2015)

    "If the extracted signal is composed of more than one dispersive wave
    propagating in distinct, albeit similar, directions, then, equations
    (10)–(12) should be applied independently to each one of them. By
    inspecting the Stockwell transform of the signal, the analyst can observe
    if there are several wavetrains."

    """
    theta_r = np.arctan(np.dot(e, vhat) / np.dot(n, vhat))
    # theta_r = np.arctan2(np.dot(e, vhat), np.dot(n, vhat))
    theta = theta_r + np.pi*(1 - np.sign(np.sin(theta_r))) + \
              np.pi*(1 - np.sign(np.cos(theta_r))) * np.sign(np.sin(theta_r))/2

    return np.degrees(theta)



def rotate_NE_RT(Sn, Se, az):                                                  [docs]
    """
    Rotate North and East s-transforms to radial and transverse, through the propagation angle.

    Parameters
    ----------
    Sn, Se : numpy.ndarray (complex, rank 2)
        Complex, equal-sized s-transform arrays, for North and East components, respectively.
    az : float
        Rotation angle [degrees].

    Returns
    -------
    Sr, St : numpy.ndarray (rank 2)
        Complex s-transform arrays for radial and transverse components, respectively.

    References
    ----------
    Equation (17) from Meta-Fajardo et al. (2015)

    """
    theta = np.radians(az)

    Sr = np.cos(theta)*Sn + np.sin(theta)*Se
    St = -np.sin(theta)*Sn + np.cos(theta)*Se

    return Sr, St



def NIP(Sr, Sv, polarization=None, eps=None):                                  [docs]
    """
```

```
    """
    Get the normalized inner product of two complex MxN stockwell transforms.

    Parameters
    ----------
    Sr, Sv: numpy.ndarray (complex, rank 2)
        The radial and vertical component s-transforms. If the polarization argument is omitted,
        Sv is assumed to be phase-shifted according to the desired polarization.
    polarization : str, optional
        If provided, the Sv will be phase-shifted according to this string before calculating the NIP.
        'retrograde' will apply a pi/2 phase advance (1j * Sv)
        'prograde' or 'linear' will apply a pi/2 phase delay (-1j * Sv)
    eps : float, optional
        Tolerance for small denominator values, for numerical stability.
        Useful for synthetic noise-free data.  Authors used 0.04.

    Returns
    -------
    nip : numpy.ndarray (rank 2)
        MxN array of floats between -1 and 1.

    References
    ----------
    Equation (16) and (26) from Meza-Fajardo et al. (2015)

    """
    if polarization:
        Svhat = shift_phase(Sv, polarization)
    else:
        # Just a literal inner product, no shift.
        Svhat = Sv

    Avhat = np.abs(Svhat)
    if eps is not None:
        mask = (Avhat / Avhat.max()) < eps
        Avhat[mask] += eps*Avhat.max()

    ip = (Sr.real)*(Svhat.real) + (Sr.imag)*(Svhat.imag)
    n = np.abs(Sr) * Avhat

    return ip/n


def get_filter(nip, polarization, threshold=None, width=0.1):                    [docs]
    r"""
    Get an NIP-based filter that will pass waves of the specified type.

    The filter is made from the NIP and cosine taper for the specified wave type.
    The nip and the polarization type must match.

    Parameters
    ----------
    nip : numpy.ndarray (real, rank 2)
        The NIP array [-1.0, 1.0]
    polarization : str
        The type of polarization that was used to calculate the provided NIP.
        'retrograde', 'prograde', or 'linear'.  See "NIP" function.
    threshold, width : float
        The cosine taper critical/crossover value ("x_r") and width ("\Delta x").
        If not supplied, the default for retrograde polarization is 0.8, and for
        prograde or linear polarization is 0.2.

    Returns
    -------
    numpy.ndarray (real, rank 2)
        The NIP-based filter array [0.0, 1.0] to multiply into the complex Stockwell arrays,
        before inverse transforming to the time-domain.

    References
    ----------
    Equation (27) and (28) from Meza-Fajardo et al. (2015)

    """
    if polarization in ('retrograde', 'prograde'):
        if threshold is None:
            threshold = 0.8
        filt = np.zeros(nip.shape)
        mid = (threshold - width < nip) & (nip < threshold)
        high = threshold < nip
        filt[mid] = 0.5 * np.cos((np.pi*(nip[mid]-threshold))/width) + 0.5
        filt[high] = 1.0
```

```
    elif polarization == 'linear':
        if threshold is None:
            threshold = 0.2
        filt = np.ones(nip.shape)
        mid = (threshold < nip) & (nip < threshold + width)
        high = threshold + width < nip
        filt[mid] = 0.5 * np.cos((np.pi*(nip[mid]-threshold))/width) + 0.5
        filt[high] = 0.0
    else:
        raise ValueError('Unknown polarization type: {}'.format(polarization))

    return filt



def NIP_filter(n, e, v, fs, xpr, polarization, threshold=0.8, width=0.1, eps=None):                [docs]
    r"""
    Filter a 3-component seismogram based on the NIP criterion.

    This is a composite convenience routine that uses sane defaults.
    If you want to get intermediate products, call internal routines individually.

    Parameters
    ----------
    n, e, v : numpy.ndarray (rank 1)
        Equal-length data arrays for North, East, and Vertical components, respectively.
    fs : float
        Sampling frequency [Hz]
    xpr : int
        Sense of wave propagation. -1 for westward, 1 for eastward.
        Try -int(np.sign(np.sin(np.radians(baz)))), unless they're directly N-S from each other.
    polarization : str
        'retrograde' to extract normal retrograde Rayleigh waves
        'prograde' to extract prograde Rayleigh waves
        'linear' to extract Love waves
    threshold, width : float
        The critical value ("x_r") and width ("\Delta x") for the NIP filter (cosine) taper.
    eps : float
        Tolerance for small NIP denominator values, for numerical stability.

    Returns
    -------
    n, e, v : numpy.ndarray (rank 1)
        Filtered north, east, and vertical components.
    theta : float
        Average propagation azimuth for the given polarization [degrees].
        Use this angle to rotate the data to radial and transverse.

    Examples
    --------
    # compare filtered north, east, vertical to original
    >>> import obspy.signal as signal
    >>> nf, ef, vf, theta = NIP_filter(n, e, v, fs, xpr)
    >>> if theta > 180:
            nip_baz = theta - 180
        else:
            nip_baz = theta + 180

    >>> rf, tf = signal.rotate_NE_RT(nf, ef, nip_baz)
    >>> r, t = signal.rotate_NE_RT(n, e, baz)


    """
    # TODO:
    # * rewrite this to get rid of all the potentially huge intermediate arrays
    # * Allow user to specify a scalar theta propagation azimuth, and sidestep
    #   the instantaneous calculation.

    shft = get_shift(polarization)

    #1. Get instantaneous theta from Sn, Se, Svhat
    Sn = stransform(n, Fs=fs)
    Se = stransform(e, Fs=fs)
    Sv = stransform(v, Fs=fs)

    theta = instantaneous_azimuth(Sv, Sn, Se, polarization, xpr)

    #2. Rotate Sn, Se through theta, and get nip from Svhat and Sr
    Sr, St = rotate_NE_RT(Sn, Se, theta)

    # damp values where theta is NaN (usually divide-by-zero problems)
    nanr = np.isnan(Sr)
```

```
    nant = np.isnan(St)
    if np.any(nanr) or np.any(nant):
        msg = "{} fraction of NaNs found".format(float(np.sum(nanr)) / nanr.size)
        warnings.warn(msg)
        Sr[nanr] = 0.0
        St[nant] = 0.0

    nip = NIP(Sr, shft * Sv, eps=eps)

    #3. get appropriate nip filter, and stamp on Sn, Se, Sv
    filt = get_filter(nip, polarization, threshold, width)

    nf = istransform(Sn*filt, Fs=fs)
    ef = istransform(Se*filt, Fs=fs)
    vf = istransform(Sv*filt, Fs=fs)

    # get the average propogation azimuth
    theta_bar = scalar_azimuth(e, n, istransform((shft * Sv) * filt, Fs=fs))

    # return to time domain
    #if theta_bar > 180:
    #    baz = theta_bar - 180
    #else:
    #    baz = theta_bar + 180

    #import obspy.signal as signal
    #rf, tf = signal.rotate_NE_RT(nf, ef, baz)

    return nf, ef, vf, theta_bar



class NIPFilter:                                                              [docs]
    """Filter data according to the normalized inner product.

    Examples
    --------
    Filter a 40 samples-per-second radial, vertical between 1 and 10 Hz.
    The transverse is passed back unaltered b/c it's not involved in the filtering.
    >>> retro_east = NIPFilter('retrograde', 'eastward')
    >>> rf, t, vf = retro_east.filter(r, t, v, fs=40, fmin=1.0, fmax=10.0)

    Filter a 40 samples-per-second radial, vertical between 1 and 10 Hz
    The radial and vertical are passed back unaltered b/c they're not involved in the filtering.
    >>> linear = NIPFilter('linear')
    >>> r, tf, v = linear.filter(r, t, v, fs=40, freqmin=5, freqmax=15)

    """
    def __init__(self, polarization, threshold=None, width=0.1, eps=None):
        polarization = polarization.lower()

        if polarization in ('retrograde', 'prograde'):
            threshold = threshold or 0.8
        elif polarization == 'linear':
            threshold = threshold or 0.2
        else:
            msg = "polarization '{}' not recognized".format(polarization)
            raise ValueError(msg)

        self.threshold = threshold
        self.polarization = polarization
        self.width = width
        self.eps = eps

    def __repr__(self):
        out = "NIPFilter('{}', threshold={}, width={}, eps={})"
        return out.format(self.polarization, self.threshold, self.width, self.eps)

    # Cache such that repeat calls to the last set of inputs aren't recalculated.
    # Could be useful for plotting calls?
    # @lru_cache(maxsize=1)
    def filter(self, radial, transverse, vertical, fs=None, freqmin=None,    [docs]
               freqmax=None):
        """Filter time series data for polarization.
        """
        pass


    def rotate(self, e, n, v, sense_of_motion):                               [docs]
    # Rotate data into the instantaneous propagation azimuth
        """Rotate time series data according to the instantaneous azimuth.
        """
```

```
        if sense_of_motion == 'eastward':
            xpr = 1
        elif sense_of_motion == 'westward':
            xpr = -1
        else:
            msg = "Unrecognized sense_of_motion: {}".format(sense_of_motion)
            raise ValueError(msg)
```