arlpy Documentation

Release 1.8

Mandar Chitre

Sep 21, 2023

Contents

Packages such as *numpy* and *scipy* provide excellent mathematical tools for scientists and engineers using Python. However, these packages are still young and evolving, and understandably have some gaps, especially when it comes to domain-specific requirements. The *arlpy* package aims to fill in some of the gaps in the areas of underwater acoustics, signal processing, and communication. Additionally, *arlpy* also includes some commonly needed utilities and plotting routines based on *bokeh*.

CHAPTER 1

General modules

The following modules are general and are likely to be of interest to researchers and developers working on signal processing, communication and underwater acoustics:

1.1 Signal processing

Signal processing toolbox.

arlpy.signal.**bb2pb**(*x*, *fd*, *fc*, *fs=None*, *axis=-1*)

Convert baseband signal to passband.

For communication applications, one may wish to use *arlpy.comms.upconvert()* instead, as that function supports pulse shaping.

The convention used in that exp(2j*pi*fc*t) is a positive frequency carrier.

Parameters

- **x** – complex baseband signal
- **fd** – sampling rate of baseband signal in Hz
- **fc** – carrier frequency in passband in Hz
- **fs** – sampling rate of passband signal in Hz (None => same as *fd*)
- **axis** – axis of the signal, if multiple signals specified

Returns real passband signal, sampled at *fs*

arlpy.signal.**correlate_periodic**(*a*, *v=None*)

Cross-correlation of two 1-dimensional periodic sequences.

a and v must be sequences with the same length. If v is not specified, it is assumed to be the same as a (i.e. the function computes auto-correlation).

Parameters

- **a** – input sequence #1
- **v** – input sequence #2

Returns discrete periodic cross-correlation of a and v

arlpy.signal.**cw**(*fc*, *duration*, *fs*, *window=None*, *complex_output=False*)
Generate a sinusoidal pulse.

Parameters

- **fc** – frequency of the pulse in Hz
- **duration** – duration of the pulse in s
- **fs** – sampling rate in Hz
- **window** – window function to use (`None` means rectangular window)
- **complex_output** – True to return complex signal, False for a real signal

For supported window functions, see documentation for `scipy.signal.get_window()`.

```
>>> import arlpy
>>> x1 = arlpy.signal.cw(fc=27000, duration=0.5, fs=250000)
>>> x2 = arlpy.signal.cw(fc=27000, duration=0.5, fs=250000, window='hamming')
>>> x3 = arlpy.signal.cw(fc=27000, duration=0.5, fs=250000, window=('kaiser', 4.
↪0))
```

arlpy.signal.**detect_impulses**(*x*, *fs*, *k=10*, *tdist=0.001*)
Detect impulses in *x*

The minimum height of impulses is defined by *a+k*b* where *a* is median of the envelope of *x* and *b* is median absolute deviation (MAD) of the envelope of *x*.

Parameters

- **x** – real signal
- **fs** – sampling frequency in Hz
- **k** – multiple of MAD for the impulse minimum height (default: 10)
- **tdist** – minimum time difference between neighbouring impulses in sec (default: 1e-3)

Returns indices and heights of detected impulses

```
>>> nsamp = 1000
>>> ind_impulses = np.array([10, 115, 641, 888])
>>> x = np.zeros((nsamp))
>>> x[ind_impulses] = 1
>>> x += np.random.normal(0, 0.1, nsamp)
>>> ind_pks, h_pks = signal.detect_impulses(x, fs=100000, k=10, tdist=1e-3)
```

arlpy.signal.**envelope**(*x*, *axis=-1*)
Generate a Hilbert envelope of the real signal x.

Parameters

- **x** – real passband signal
- **axis** – axis of the signal, if multiple signals specified

arlpy.signal.**gmseq**(*spec*, *theta=None*)
Generate generalized m-sequence.

Generalized m-sequences are related to m-sequences but have an additional parameter θ. When $\theta = \pi/2$, generalized m-sequences become normal m-sequences. When $\theta < \pi/2$, generalized m-sequences contain a DC-component that leads to an exalted carrier after modulation.

When theta is $\arctan(\sqrt{n})$ where n is the length of the m-sequence, the m-sequence is considered to be period matched. Period matched m-sequences are complex sequences with perfect discrete periodic auto-correlation properties, i.e., all non-zero lag periodic auto-correlations are zero. The zero-lag autocorrelation is $n = 2^m - 1$, where m is the shift register length.

This function currently supports shift register lengths between 2 and 30.

Parameters

- **spec** – m-sequence specifier (shift register length or taps)
- **theta** – transmission angle (`None` to use period-matched angle)

```
>>> import arlpy
>>> x = arlpy.signal.gmseq(7)
>>> len(x)
127
```

`arlpy.signal.`**`goertzel`**(*f*, *x*, *fs=2.0*, *filter=False*)

Goertzel algorithm for single tone detection.

The output of the Goertzel algorithm is the same as a single bin DFT if `f/(fs/N)` is an integer, where `N` is the number of points in signal `x`.

The detection metric returned by this function is the magnitude of the output of the Goertzel algorithm at the end of the input block. If `filter` is set to `true`, the complex time series at the output of the IIR filter is returned, rather than just the detection metric.

Parameters

- **f** – frequency of tone of interest in Hz
- **x** – real or complex input sequence
- **fs** – sampling frequency of x in Hz
- **filter** – output complex time series if true, detection metric otherwise (default: false)

Returns detection metric or complex time series

```
>>> import arlpy
>>> x1 = arlpy.signal.cw(64, 1, 512)
>>> g1 = arlpy.signal.goertzel(64, x1, 512)
>>> g1
256.0
>>> g2 = arlpy.signal.goertzel(32, x1, 512)
>>> g2
0.0
```

`arlpy.signal.`**`lfilter0`**(*b*, *a*, *x*, *axis=-1*)

Filter data with an IIR or FIR filter with zero DC group delay.

`scipy.signal.lfilter()` provides a way to filter a signal *x* using a FIR/IIR filter defined by *b* and *a*. The resulting output is delayed, as compared to the input by the group delay. This function corrects for the group delay, resulting in an output that is synchronized with the input signal. If the filter as an acausal impulse response, some precursor signal from the output will be lost. To avoid this, pad input signal *x* with sufficient zeros at the beginning to capture the precursor. Since both, `scipy.signal.lfilter()` and this function

return a signal with the same length as the input, some signal tail is lost at the end. To avoid this, pad input signal *x* with sufficient zeros at the end.

See documentation for `scipy.signal.lfilter()` for more details.

```
>>> import arlpy
>>> import numpy as np
>>> fs = 250000
>>> b = arlpy.uwa.absorption_filter(fs, distance=500)
>>> x = np.pad(arlpy.signal.sweep(20000, 40000, 0.5, fs), (127, 127), 'constant')
>>> y = arlpy.signal.lfilter0(b, 1, x)
```

`arlpy.signal.`**`lfilter_gen`**(*b*, *a*)

Generator form of an FIR/IIR filter.

The filter is a direct form I implementation of the standard difference equation. Data samples can be passed to the filter using the `send()` method, and the output can be read a sample at a time.

```
>>> import arlpy
>>> import numpy as np
>>> import scipy.signal as sp
>>> b, a = sp.iirfilter(2, 0.1, btype='lowpass')  # generate a biquad lowpass
>>> f = arlpy.signal.filter_gen(b, a)             # create the filter
>>> x = np.random.normal(0, 1, 1000)              # get some random data
>>> y = [f.send(v) for v in x]                    # filter data by stepping␣
↪through it
```

`arlpy.signal.`**`mfilter`**(*s*, *x*, *complex_output=False*, *axis=-1*)

Matched filter recevied signal using a reference signal.

Parameters

- **`s`** – reference signal
- **`x`** – recevied signal
- **`complex_output`** – True to return complex signal, False for absolute value of complex signal
- **`axis`** – axis of the signal, if multiple recevied signals specified

`arlpy.signal.`**`mseq`**(*spec*, *n=None*)

Generate m-sequence.

m-sequences are sequences of ± 1 values with near-perfect discrete periodic auto-correlation properties. All non-zero lag periodic auto-correlations are -1. The zero-lag autocorrelation is $2^m - 1$, where m is the shift register length.

This function currently supports shift register lengths between 2 and 30.

Parameters

- **`spec`** – m-sequence specifier (shift register length or taps)
- **`n`** – length of sequence (`None` means full length of $2^m - 1$)

```
>>> import arlpy
>>> x = arlpy.signal.mseq(7)
>>> len(x)
127
```

`arlpy.signal.`**`nco`**(*fc*, *fs=2.0*, *phase0=0*, *wrap=6.283185307179586*, *func=<function <lambda>>*)

Numerically controlled oscillator (NCO).

If fs is specified, fc is given in Hz, otherwise it is specified as normalized frequency (Nyquist = 1).

The default oscillator function is `exp(i*phase)` to generate a complex sinusoid. Alternate oscillator functions that take in the phase angle and generate other outputs can be specifed. For example, a real sinusoid can be generated by specifying `sin` as the function. The phase angle can be generated by specifying `None` as the function.

Parameters

- **fc** – array of instantaneous oscillation frequency
- **fs** – sampling frequency in Hz
- **phase0** – initial phase in radians (default: 0)
- **wrap** – phase angle to wrap phase around to 0 (default: 2π)
- **func** – oscillator function of phase angle (default: complex sinusoid)

```
>>> import arlpy
>>> import numpy as np
>>> fc = np.append([27000]*12, [54000]*5)
>>> x = arlpy.signal.nco(fc, 108000, func=np.sin)
>>> x
[0, 1, 0, -1, 0, 1, 0, -1, 0, 1, 0, -1, 1, -1, 1, -1, 1]
```

arlpy.signal.**nco_gen**(*fc*, *fs=2.0*, *phase0=0*, *wrap=6.283185307179586*, *func=<function <lambda>>*)

Generator form of a numerically controlled oscillator (NCO).

Samples at the output of the oscillator can be generated using the `next()` method. The oscillator frequency can be modified during operation using the `send()` method, with *fc* as the argument.

If fs is specified, fc is given in Hz, otherwise it is specified as normalized frequency (Nyquist = 1).

The default oscillator function is `exp(i*phase)` to generate a complex sinusoid. Alternate oscillator functions that take in the phase angle and generate other outputs can be specifed. For example, a real sinusoid can be generated by specifying `sin` as the function. The phase angle can be generated by specifying `None` as the function.

Parameters

- **fc** – oscillation frequency
- **fs** – sampling frequency in Hz
- **phase0** – initial phase in radians (default: 0)
- **wrap** – phase angle to wrap phase around to 0 (default: 2π)
- **func** – oscillator function of phase angle (default: complex sinusoid)

```
>>> import arlpy
>>> import math
>>> nco = arlpy.signal.nco_gen(27000, 108000, func=math.sin)
>>> x = [nco.next() for i in range(12)]
>>> x = np.append(x, nco.send(54000))      # change oscillation frequency
>>> x = np.append(x, [nco.next() for i in range(4)])
>>> x
[0, 1, 0, -1, 0, 1, 0, -1, 0, 1, 0, -1, 1, -1, 1, -1, 1]
```

arlpy.signal.**pb2bb**(*x*, *fs*, *fc*, *fd=None*, *flen=127*, *cutoff=None*, *axis=-1*)

Convert passband signal to baseband.

The baseband conversion uses a low-pass filter after downconversion, with a default cutoff frequency of *0.6*fd*, if *fd* is specified, or *1.1*fc* if *fd* is not specified. Alternatively, the user may specify the cutoff frequency explicitly.

For communication applications, one may wish to use `arlpy.comms.downconvert()` instead, as that function supports matched filtering with a pulse shape rather than a generic low-pass filter.

The convention used in that exp(2j*pi*fc*t) is a positive frequency carrier.

Parameters

- **x** – passband signal
- **fs** – sampling rate of passband signal in Hz
- **fc** – carrier frequency in passband in Hz
- **fd** – sampling rate of baseband signal in Hz (`None` => same as *fs*)
- **flen** – number of taps in the low-pass FIR filter
- **cutoff** – cutoff frequency in Hz (`None` means auto-select)
- **axis** – axis of the signal, if multiple signals specified

Returns complex baseband signal, sampled at *fd*

`arlpy.signal.`**sweep**(*f1*, *f2*, *duration*, *fs*, *method='linear'*, *window=None*)

Generate frequency modulated sweep.

Parameters

- **f1** – starting frequency in Hz
- **f2** – ending frequency in Hz
- **duration** – duration of the pulse in s
- **fs** – sampling rate in Hz
- **method** – type of sweep (`'linear'`, `'quadratic'`, `'logarithmic'`, `'hyperbolic'`)
- **window** – window function to use (`None` means rectangular window)

For supported window functions, see documentation for `scipy.signal.get_window()`.

```
>>> import arlpy
>>> x1 = arlpy.signal.sweep(20000, 30000, duration=0.5, fs=250000)
>>> x2 = arlpy.signal.sweep(20000, 30000, duration=0.5, fs=250000, window='hamming
↪')
>>> x2 = arlpy.signal.sweep(20000, 30000, duration=0.5, fs=250000, window=('kaiser
↪', 4.0))
```

`arlpy.signal.`**time**(*n*, *fs*)

Generate a time vector for time series.

Parameters

- **n** – time series, or number of samples
- **fs** – sampling rate in Hz

Returns time vector starting at time 0

```
>>> import arlpy
>>> t = arlpy.signal.time(100000, fs=250000)
>>> t
array([  0.00000000e+00,   4.00000000e-06,   8.00000000e-06, ...,
     3.99988000e-01,   3.99992000e-01,   3.99996000e-01])
>>> x = arlpy.signal.cw(fc=27000, duration=0.5, fs=250000)
>>> t = arlpy.signal.time(x, fs=250000)
>>> t
array([  0.00000000e+00,   4.00000000e-06,   8.00000000e-06, ...,
     4.99988000e-01,   4.99992000e-01,   4.99996000e-01])
```

1.2 Communications

Communications toolbox.

arlpy.comms.**awgn**(*x*, *snr*, *measured=False*, *complex=None*)

Add Gaussian noise to signal.

Parameters

- **x** – real passband or complex baseband signal
- **snr** – SNR in dB
- **measured** – True to measure signal strength, False to assume unit strength signal
- **complex** – True for complex noise, False for real noise, None to automatically decide

```
>>> import arlpy
>>> d1 = arlpy.comms.random_data(100, 4)
>>> qpsk = arlpy.comms.psk(4)
>>> x = arlpy.comms.modulate(d1, qpsk)
>>> y = arlpy.comms.awgn(x, 6)
>>> d2 = arlpy.comms.demodulate(y, qpsk)
>>> arlpy.comms.ser(d1, d2)
0.02
```

arlpy.comms.**ber**(*x*, *y*, *m=2*)

Measure bit error rate between symbols in x and y.

Parameters

- **x** – symbol array #1
- **y** – symbol array #2
- **m** – symbol alphabet size (maximum 64)

Returns bit error rate

```
>>> import arlpy
>>> arlpy.comms.ber([0,1,2,3], [0,1,2,2], m=4)
0.125
```

arlpy.comms.**bi2sym**(*x*, *m*)

Convert bits to symbols.

Parameters

- **x** – bit array

- **m** – symbol alphabet size (must be a power of 2)

Returns symbol array

```
>>> import arlpy
>>> arlpy.comms.bi2sym([0, 0, 1, 0, 1, 0, 1, 1, 1], 8)
array([1, 2, 7])
```

arlpy.comms.**demodulate**(*x*, *const*, *metric=None*, *decision=<function <lambda>>*)

Demodulate complex baseband signal based on the specified constellation.

Parameters

- **x** – complex baseband signal to demodulate
- **const** – constellation to use
- **metric** – distance metric to use as a measure of closeness of signals
- **decision** – rule for decision making, None to return soft decisions

Returns demodulated data symbols (or metric in soft decision mode)

The metric is a function that takes in two signal segments and measures a "distance" between them. The smaller the distance (allowed to be negative), the closer the signals. Usually one signal is from the constellation while the other is a segment of the input signal to demodulate. When unspecified, the metric for a complex valued constellation (such as PSK/QAM) is the Euclidean distance, for a real valued constellation (such as OOK) is the incoherent difference in signal level, and for a vector valued constellation is the dot product.

The decision rule is a function that takes in the metric of all possible constellation points and decides on the demodulated data. By default, this is the argmin function. If the decision rule is set to None, no hard decision is made, and the metric is returned as a "soft decision".

```
>>> import arlpy
>>> bpsk = arlpy.comms.psk()
>>> d1 = arlpy.comms.random_data(100)
>>> x = arlpy.comms.modulate(d1, bpsk)
>>> d2 = arlpy.comms.demodulate(x, bpsk)
>>> arlpy.comms.ber(d1, d2)
0.0
```

arlpy.comms.**diff_decode**(*x*)

Decode phase differential baseband signal.

Parameters **x** – complex baseband differential data to decode

Returns decoded complex baseband data of length len(x)-1

```
>>> import arlpy
>>> d1 = arlpy.comms.random_data(100, 4)
>>> qpsk = arlpy.comms.psk(4)
>>> x = arlpy.comms.modulate(d1, qpsk)
>>> y = arlpy.comms.diff_encode(x)
>>> z = arlpy.comms.diff_decode(y)
>>> d2 = arlpy.comms.demodulate(z, qpsk)
>>> arlpy.comms.ser(d1, d2)
0.0
```

arlpy.comms.**diff_encode**(*x*)

Encode phase differential baseband signal.

Parameters **x** – complex baseband data to encode differentially

Returns differentially encoded complex baseband data of length len(x)+1

```
>>> import arlpy
>>> x = arlpy.comms.modulate(arlpy.comms.random_data(100, 4), arlpy.comms.psk(4))␣
↪  # QPSK
>>> len(x)
100
>>> y = arlpy.comms.diff_encode(x)  # DQPSK
>>> len(y)
101
>>> x[0]
(0.707+0.707j)
>>> y[1]/y[0]
(0.707+0.707j)
```

arlpy.comms.**downconvert**(*x*, *sps*, *fc*, *fs=2.0*, *g=None*)

Downconvert a passband signal with a matched pulse shaping filter.

This function supports downconversion by an integer factor. For a more general baseband conversion (but without matched filtering), see *arlpy.signal.pb2bb()*.

If the carrier frequency is 0, the input is assumed to be complex baseband, and only undergoes matched filtering and downsampling. If the pulse shape is None, a rectangular pulse shape is assumed.

The downconversion process introduces a group delay depending on the matched filter. It is usually (len(g)-1)/2 passband samples.

Parameters

- **x** – real passband data (or complex baseband data at passband sampling rate, if fc=0)
- **sps** – number of passband samples per baseband symbol
- **fc** – carrier frequency in Hz
- **fs** – passband sampling rate
- **g** – pulse shaping filter (for matched filtering)

```
>>> import arlpy
>>> d1 = arlpy.comms.random_data(100, 4)
>>> qpsk = arlpy.comms.psk(4)
>>> bb1 = arlpy.comms.modulate(d1, qpsk)
>>> rrc = arlpy.comms.rrcosfir(0.25, 6)
>>> pb = arlpy.comms.upconvert(bb1, 6, 27000, 108000, rrc)
>>> bb2 = arlpy.comms.downconvert(pb, 6, 27000, 108000, rrc)
>>> delay = (len(rrc)-1)/2    # compute passband group delay of rrc FIR filter
>>> delay = 2*delay/6         # compute baseband group delay after filtering twice
>>> d2 = arlpy.comms.demodulate(bb2[delay:-delay], qpsk)
>>> arlpy.comms.ser(d1, d2)
0.0
```

arlpy.comms.**fsk**(*m=2*, *n=None*)

Generate an m-FSK constellation.

The concept of signal constellation is generalized to allow vectors to enable representation of FSK as signal points. The signal constellation then becomes a set of vectors, each vector representing the baseband signal to be used when the corresponding symbol is to be transmitted.

The constellation is scaled for unit average energy per sample, assuming the symbols are randomly distributed.

If n is unspecified, 2m baseband samples are generated per symbol. This ensures an integral number of cycles per symbol, and hence continuous phase.

Parameters

- **m** – symbol alphabet size
- **n** – number of baseband samples per symbol

```
>>> import arlpy
>>> x = arlpy.comms.modulate(arlpy.comms.random_data(100, m=2), arlpy.comms.fsk())
```

arlpy.comms.**gray_code**(*m*)

Generate a Gray code map of size m.

Parameters **m** – symbol alphabet size

Returns a mapping from integers (indices) to Gray coded integers

```
>>> import arlpy
>>> x = arlpy.comms.gray_code(8)
>>> x
array([0, 1, 3, 2, 6, 7, 5, 4])
>>> x[3]   # 3 => 2
2
```

arlpy.comms.**invert_map**(*x*)

Generate an inverse map.

Parameters **x** – map, such as that generated by *gray_code()*

Returns an inverse map y, such that `y[x[j]] = j`

```
>>> import arlpy
>>> x = arlpy.comms.gray_code(8)
>>> y = arlpy.comms.invert_map(x)
>>> x[2]
3
>>> y[3]
2
```

arlpy.comms.**modulate**(*data*, *const*)

Modulate data into signal points for the specified constellation.

The data must use only symbol alphabet defined for the specified constellation.

Parameters

- **data** – data symbols to modulate
- **const** – constellation to use

Returns modulated complex baseband signal

```
>>> import arlpy
>>> x = arlpy.comms.modulate(arlpy.comms.random_data(100), arlpy.comms.psk())
```

arlpy.comms.**ook**()

Generate an on-off keying constellation.

The constellation represents the baseband values for bits 0 and 1 respectively. The constellation is scaled for unit average energy per bit, assuming the bits are randomly distributed.

```
>>> import arlpy
>>> arlpy.comms.ook()
array([0, 1.414])
```

arlpy.comms.**pam**(*m=2*, *gray=True*, *centered=True*)
Generate a PAM constellation with m signal points.

The constellation represents the baseband values for symbols 0 through m-1 respectively. The constellation is scaled for unit average energy per sample, assuming the symbols are randomly distributed.

Parameters

- **m** – symbol alphabet size
- **gray** – True to use Gray coding, False otherwise
- **centered** – True to center constellation around 0, False otherwise

```
>>> import arlpy
>>> arlpy.comms.pam()
array([-1, 1])
>>> arlpy.comms.pam(m=4, gray=False, centered=False)
array([0, 0.535, 1.069, 1.604])
```

arlpy.comms.**psk**(*m=2*, *phase0=None*, *gray=True*)
Generate a PSK constellation with m signal points.

The constellation represents the baseband values for symbols 0 through m-1 respectively. The constellation is scaled for unit average energy per sample, assuming the symbols are randomly distributed.

Parameters

- **m** – symbol alphabet size
- **phase0** – phase of the 0 symbol (None => pi/4 for QPSK, 0 otherwise)
- **gray** – True to use Gray coding, False otherwise

```
>>> import arlpy
>>> arlpy.comms.psk()
array([1+0j, -1+0j])
>>> arlpy.comms.psk(4)
array([0.707+0.707j, -0.707+0.707j, 0.707-0.707j, -0.707-0.707j])
>>> arlpy.comms.iqplot(arlpy.comms.psk(4))
```

arlpy.comms.**qam**(*m=16*, *gray=True*)
Generate a QAM constellation with m signal points.

The constellation represents the baseband values for symbols 0 through m-1 respectively. The constellation is scaled for unit average energy per sample, assuming the symbols are randomly distributed.

Parameters

- **m** – symbol alphabet size (must be a square of an integer)
- **gray** – True to use Gray coding, False otherwise

```
>>> import arlpy
>>> arlpy.comms.iqplot(arlpy.comms.qam(16))
```

arlpy.comms.**random_data**(*size*, *m=2*)
Generate random symbols in the range [0, m-1].

Parameters

- **size** – number of data points (or shape) to produce
- **m** – symbol alphabet size

```
>>> import arlpy
>>> arlpy.comms.random_data(10)
array([1, 0, 0, 1, 1, 0, 1, 0, 1, 0])
>>> arlpy.comms.random_data(10, 4)
array([0, 2, 2, 3, 2, 3, 2, 0, 1, 0])
>>> arlpy.comms.random_data((2,2))
array([[0, 1],
       [0, 0]])
```

arlpy.comms.**rcosfir**(*beta*, *sps*, *span=None*)

Generates a raised cosine FIR filter.

Parameters

- **beta** – shape of the raised cosine filter (0-1)
- **sps** – number of samples per symbol
- **span** – length of the filter in symbols (None => automatic selection)

```
>>> import arlpy
>>> rc = arlpy.comms.rcosfir(0.25, 6)
>>> bb = arlpy.comms.modulate(arlpy.comms.random_data(100), arlpy.comms.psk())
>>> pb = arlpy.comms.upconvert(bb, 6, 27000, 18000, rc)
```

arlpy.comms.**rrcosfir**(*beta*, *sps*, *span=None*)

Generates a root raised cosine FIR filter.

Parameters

- **beta** – shape of the root raised cosine filter (0-1)
- **sps** – number of samples per symbol
- **span** – length of the filter in symbols (None => automatic selection)

```
>>> import arlpy
>>> rrc = arlpy.comms.rrcosfir(0.25, 6)
>>> bb = arlpy.comms.modulate(arlpy.comms.random_data(100), arlpy.comms.psk())
>>> pb = arlpy.comms.upconvert(bb, 6, 27000, 18000, rrc)
```

arlpy.comms.**ser**(*x*, *y*)

Measure symbol error rate between symbols in x and y.

Parameters

- **x** – symbol array #1
- **y** – symbol array #2

Returns symbol error rate

```
>>> import arlpy
>>> arlpy.comms.ser([0,1,2,3], [0,1,2,2])
0.25
```

arlpy.comms.**sym2bi**(*x*, *m*)
Convert symbols to bits.

Parameters

- **x** – symbol array
- **m** – symbol alphabet size (must be a power of 2)

Returns bit array

```
>>> import arlpy
>>> arlpy.comms.sym2bi([1, 2, 7], 8)
array([0, 0, 1, 0, 1, 0, 1, 1, 1])
```

arlpy.comms.**upconvert**(*x*, *sps*, *fc*, *fs=2.0*, *g=None*)
Upconvert a complex baseband signal with pulse shaping.

This function supports upconversion by an integer factor. For a more general passband conversion (but without pulse shaping), see *arlpy.signal.bb2pb()*.

If the carrier frequency is 0, the upsampled (at passband sampling rate) and pulse shaped complex baseband data is returned. If the pulse shape is None, a rectangular pulse shape is assumed.

The upconversion process introduces a group delay depending on the pulse shaping filter. It is usually (len(g)-1)/2 passband samples. When g is None, the group delay is (sps-1)/2 passband samples.

Parameters

- **x** – complex baseband data
- **sps** – number of passband samples per baseband symbol
- **fc** – carrier frequency in Hz
- **fs** – passband sampling rate
- **g** – pulse shaping filter

```
>>> import arlpy
>>> rrc = arlpy.comms.rrcosfir(0.25, 6)
>>> bb = arlpy.comms.modulate(arlpy.comms.random_data(100), arlpy.comms.psk())
>>> pb = arlpy.comms.upconvert(bb, 6, 27000, 108000, rrc)
```

1.3 Beamforming and array processing

Array signal processing / beamforming toolbox.

arlpy.bf.**bartlett**(*x*, *fc*, *sd*, *shading=None*, *complex_output=False*)
Frequency-domain Bartlett beamformer.

The timeseries data must be 2D with narrowband complex timeseries for each sensor in individual rows. The steering delays must also be 2D with a row per steering direction.

If the timeseries data is specified as 1D array, it is assumed to represent multiple sensors at a single time.

Parameters

- **x** – narrowband complex timeseries data for multiple sensors (row per sensor)
- **fc** – carrier frequency for the array data (Hz)
- **sd** – steering delays (s)

- **shading** – window function to use for array shading (None means no shading)
- **complex_output** – True for complex signal, False for beamformed power

Returns beamformer output averaged across time

```
>>> from arlpy import bf
>>> import numpy as np
>>> # narrowband (1 kHz) timeseries array data assumed to be loaded in x
>>> # sensor positions assumed to be in pos
>>> y = bf.bartlett(x, 1000, bf.steering_plane_wave(pos, 1500, np.linspace(-np.pi/
↪2, np.pi/2, 181)))
```

arlpy.bf.**bartlett_beampattern**(*i*, *fc*, *sd*, *shading=None*, *theta=None*, *show=False*)

Computes the beampattern for a Bartlett or delay-and-sum beamformer.

Parameters

- **i** – row index of target steering distances
- **fc** – carrier frequency for the array data (Hz)
- **sd** – steering delays (s)
- **shading** – window function to use for array shading (None means no shading)
- **theta** – angles (in radians) for display if beampattern is plotted
- **show** – True to plot the beampattern, False to return it

Returns beampattern power response at all directions corresponding to rows in sd

```
>>> from arlpy import bf
>>> import numpy as np
>>> sd = bf.steering_plane_wave(np.linspace(0, 5, 11), 1500, np.linspace(-np.pi/2,
↪ np.pi/2, 181))
>>> bp = bf.bartlett_beampattern(90, 1500, sd, show=True)
```

arlpy.bf.**broadband**(*x*, *fs*, *nfft*, *sd*, *f0=0*, *fmin=None*, *fmax=None*, *overlap=0*, *beamformer=<function bartlett>*, ***kwargs*)

Frequency-domain broadband beamformer operating on time-domain input data.

The broadband beamformer is implementing by taking STFT of the data, applying narrowband beamforming to each frequency bin, and integrating the beamformer output power across the entire bandwidth.

The array data must be 2D with timeseries for each sensor in individual rows. The steering delays must also be 2D with a row per steering direction.

The STFT window size should be chosen such that the corresponding distance (based on wave propagation speed) is much larger than the aperture size of the array.

If the array data is real and f0 is zero, the data is assumed to be passband and so only half the frequency components are computed.

Parameters

- **x** – timeseries data for multiple sensors (row per sensor)
- **fs** – sampling rate for array data (Hz)
- **c** – wave propagation speed (m/s)
- **nfft** – STFT window size
- **sd** – steering delays (s)

- **f0** – carrier frequency (for baseband data) (Hz)
- **fmin** – minimum frequency to integrate (Hz)
- **fmax** – maximum frequency to integrate (Hz)
- **overlap** – window overlap for STFT
- **beamformer** – narrowband beamformer to use

Returns beamformer output with steering directions as the first axis, time as the second, and if complex output, fft bins as the third

```
>>> from arlpy import bf
>>> # passband timeseries array data assumed to be loaded in x, sampled at fs
>>> # sensor positions assumed to be in pos
>>> sd = bf.steering_plane_wave(pos, 1500, np.linspace(-np.pi/2, np.pi/2, 181))
>>> y = bf.broadband(x, fs, 256, sd, beamformer=capon)
>>> y1 = bf.music(x, fs, 256, sd, beamformer=music, nsignals=1)
```

arlpy.bf.**capon**(*x*, *fc*, *sd*, *complex_output=False*)

Frequency-domain Capon beamformer.

The timeseries data must be 2D with narrowband complex timeseries for each sensor in individual rows. The steering delays must also be 2D with a row per steering direction.

If the timeseries data is specified as 1D array, it is assumed to represent multiple sensors at a single time.

The covariance matrix of x is estimated over the entire timeseries, and used to compute the optimal weights for the Capon beamformer.

Parameters

- **x** – narrowband complex timeseries data for multiple sensors (row per sensor)
- **fc** – carrier frequency for the array data (Hz)
- **sd** – steering delays (s)
- **complex_output** – True for complex signal, False for beamformed power

Returns beamformer output averaged across time

```
>>> from arlpy import bf
>>> import numpy as np
>>> # narrowband (1 kHz) timeseries array data assumed to be loaded in x
>>> # sensor positions assumed to be in pos
>>> y = bf.capon(x, 1000, bf.steering_plane_wave(pos, 1500, np.linspace(-np.pi/2,␣
↪np.pi/2, 181)))
```

arlpy.bf.**covariance**(*x*)

Compute array covariance matrix.

Parameters **x** – narrowband complex timeseries data for multiple sensors (row per sensor)

arlpy.bf.**delay_and_sum**(*x*, *fs*, *sd*, *shading=None*)

Time-domain delay-and-sum beamformer.

The array data must be 2D with timeseries for each sensor in individual rows. The steering delays must also be 2D with a row per steering direction.

Parameters

- **x** – passband real timeseries data for multiple sensors (row per sensor)

- **fs** – sampling rate for the array data (Hz)
- **sd** – steering delays (s)
- **shading** – window function to use for array shading (None means no shading)

Returns beamformer timeseries output with time as the last axis, and steering directions as the first

```
>>> from arlpy import bf
>>> import numpy as np
>>> # timeseries array data assumed to be loaded in x
>>> # sensor positions assumed to be in pos
>>> y = bf.delay_and_sum(x, 1000, bf.steering_plane_wave(pos, 1500, np.linspace(-
↪np.pi/2, np.pi/2, 181)))
```

arlpy.bf.**music**(*x*, *fc*, *sd*, *nsignals=1*, *complex_output=False*)

Frequency-domain MUSIC beamformer.

The timeseries data must be 2D with narrowband complex timeseries for each sensor in individual rows. The steering delays must also be 2D with a row per steering direction.

If the timeseries data is specified as 1D array, it is assumed to represent multiple sensors at a single time.

The covariance matrix of x is estimated over the entire timeseries, and used to compute the optimal weights for the MUSIC beamformer via eigen-decomposition.

Parameters

- **x** – narrowband complex timeseries data for multiple sensors (row per sensor)
- **fc** – carrier frequency for the array data (Hz)
- **sd** – steering delays (s)
- **nsignals** – number of signal eigenvectors (rest are considered noise)
- **complex_output** – True for complex signal, False for beamformed power

Returns beamformer output averaged across time

```
>>> from arlpy import bf
>>> import numpy as np
>>> # narrowband (1 kHz) timeseries array data assumed to be loaded in x
>>> # sensor positions assumed to be in pos
>>> y = bf.music(x, 1000, bf.steering_plane_wave(pos, 1500, np.linspace(-np.pi/2,␣
↪np.pi/2, 181)))
```

arlpy.bf.**normalize**(*x*, *unit_variance=True*)

Normalize array timeseries data to be zero-mean and equal variance.

The average signal power across the array is retained if *unit_variance* is set to True so that the beamformed data can be compared with other datsets.

Parameters

- **x** – passband real timeseries data for multiple sensors (row per sensor)
- **unit_variance** – True to make timeseries unit variance, False to retain average signal power across the array

Returns normalized passband real timeseries data

arlpy.bf.**steering_plane_wave**(*pos*, *c*, *theta*)

Compute steering delays assuming incoming signal has a plane wavefront.

For linear arrays, pos is 1D array. For planar and 3D arrays, pos is a 2D array with a sensor position vector in each row.

For linear arrays, theta is a 1D array of angles (in radians) with 0 being broadside. For planar and 3D arrays, theta is a 2D array with an (azimuth, elevation) pair in each row. Such arrays can be easily generated using the *arlpy.utils.linspace2d()* function.

The broadside direction is along the x-axis of a right-handed coordinate system with z-axis pointing upwards, and has azimuth and elevation as 0. In case of linear arrays, the y-coordinate is the sensor position. In case of planar arrays, if only 2 coordinates are provided, these coordinates are assumed to be y and z.

Parameters

- **pos** – sensor positions (m)
- **c** – signal propagation speed (m/s)
- **theta** – steering directions (radians)

Returns steering delays with a row for each direction (s)

```
>>> import numpy as np
>>> from arlpy import bf, utils
>>> pos1 = [0.0, 0.5, 1.0, 1.5, 2.0]
>>> a1 = bf.steering_plane_wave(pos1, 1500, np.deg2rad(np.linspace(-90, 90, 181)))
>>> pos2 = [[0.0, 0.0],
            [0.0, 0.5],
            [0.5, 0.0],
            [0.5, 0.5]]
>>> a2 = bf.steering_plane_wave(pos2, 1500, np.deg2rad(utils.linspace2d(-20, 20,␣
↪41, -10, 10, 21)))
```

arlpy.bf.**stft**(*x*, *nfft*, *overlap=0*, *window=None*)

Compute short time Fourier transform (STFT) of array data.

Parameters

- **x** – passband real timeseries data for multiple sensors (row per sensor)
- **nfft** – window length in samples
- **overlap** – number of samples of overlap between windows
- **window** – window function to use (None means rectangular window)

Returns 3d array of time x frequency x sensor

For supported window functions, see documentation for `scipy.signal.get_window()`.

1.4 Stable distributions

Stable distribution toolbox.

arlpy.stable.**rnd**(*alpha=1.5*, *beta=0*, *scale=1*, *loc=0.0*, *size=1*)

Generate independent stable random numbers.

Parameters

- **alpha** – characteristic exponent (0.1 to 2.0)
- **beta** – skew (-1 to +1)
- **scale** – scale parameter

- **loc** – location parameter (mean for alpha > 1, median/mode when beta=0)
- **size** – size of the random array to generate

This implementation is based on the method in J.M. Chambers, C.L. Mallows and B.W. Stuck, "A Method for Simulating Stable Random Variables," JASA 71 (1976): 340-4. McCulloch's MATLAB implementation (1996) served as a reference in developing this code.

arlpy.stable.**sstabfit**(*x*)

Fit a symmetric alpha stable distribution to data.

Parameters **x** – data

Returns (alpha, c, delta)

alpha, c and delta are the characteristic exponent, scale parameter (dispersion^1/alpha) and location parameter respectively.

alpha is computed based on McCulloch (1986) fractile. c is computed based on Fama & Roll (1971) fractile. delta is the 50% trimmed mean of the sample.

1.5 Geographical coordinates

Geographical coordinates toolbox.

This toolbox provides functions to work with different geographical coordinate systems. Latitude/longitude position is represented by parameter *latlong* with one of these formats:

- *(latitude degrees, longitude degrees)*
- *(latitude degrees, longitude degrees, altitude)*
- *(latitude degrees, minutes, longitude degrees, minutes)*
- *(latitude degrees, minutes, longitude degrees, minutes, altitude)*
- *(latitude degrees, minutes, seconds, longitude degrees, minutes, seconds)*
- *(latitude degrees, minutes, seconds, longitude degrees, minutes, seconds, altitude)*

The *altitude* is always specified in meters, with negative values being depth below water surface.

pos represents position in a local coordinate system *(easting, northing)* or *(easting, northing, altitude)* specified by a UTM zone.

arlpy.geo.**d**(*latlong*)

Convert latitude/longitude to (latitude degrees, longitude degrees) format.

arlpy.geo.**distance**(*pos1*, *pos2*)

Compute distance between two UTM positions.

arlpy.geo.**dm**(*latlong*)

Convert latitude/longitude to (latitude degrees, minutes, longitude degrees, minutes) format.

arlpy.geo.**dms**(*latlong*)

Convert latitude/longitude to (latitude degrees, minutes, seconds, longitude degrees, minutes, seconds) format.

arlpy.geo.**dmsz**(*latlong*)

Convert latitude/longitude to (latitude degrees, minutes, seconds, longitude degrees, minutes, seconds, altitude) format.

arlpy.geo.**dmz**(*latlong*)

Convert latitude/longitude to (latitude degrees, minutes, longitude degrees, minutes, altitude) format.

arlpy.geo.**dz**(*latlong*)
Convert latitude/longitude to (latitude degrees, longitude degrees, altitude) format.

arlpy.geo.**latlong**(*pos*, *zone=None*, *origin=None*)
Convert local coordinate system position to latitude/longitude.

To convert a UTM position into a global latitude/longitude, the local coordinate system has to be specified in terms of a UTM zone 2-tuple, e.g. `(32, 'U')`. Alternatively a local coordinate system can be specified in terms of an origin latitude/longitude.

arlpy.geo.**pos**(*latlong*, *zonenum=None*, *origin=None*)
Convert latitude/longitude to local coordinate system position.

If an origin is specified, the local coordinate system is set up with that origin and East-North axis. If no origin is specified, the UTM local coordinate system is used. A specific UTM zone can be forced by specifying *zonenum*, if desired.

arlpy.geo.**str**(*latlong*)
Convert latitude/longitude information in various formats into a pretty printable Unicode string.

arlpy.geo.**zone**(*latlong*)
Convert latitude/longitude to UTM zone.

1.6 Underwater acoustics

Underwater acoustics toolbox.

arlpy.uwa.**absorption**(*frequency*, *distance=1000*, *temperature=27*, *salinity=35*, *depth=10*, *pH=8.1*)
Get the acoustic absorption in water.

Computes acoustic absorption in water using Francois-Garrison model.

Parameters

- **frequency** – frequency in Hz
- **distance** – distance in m
- **temperature** – temperature in deg C
- **salinity** – salinity in ppt
- **depth** – depth in m
- **pH** – pH of water

Returns absorption as a linear multiplier

```
>>> import arlpy
>>> arlpy.uwa.absorption(50000)
0.2914
>>> arlpy.utils.mag2db(arlpy.uwa.absorption(50000))
-10.71
>>> arlpy.utils.mag2db(arlpy.uwa.absorption(50000, distance=3000))
-32.13
```

arlpy.uwa.**absorption_filter**(*fs*, *ntaps=31*, *nfreqs=64*, *distance=1000*, *temperature=27*, *salinity=35*, *depth=10*)
Design a FIR filter with response based on acoustic absorption in water.

Parameters

- **fs** – sampling frequency in Hz
- **ntaps** – number of FIR taps
- **nfreqs** – number of frequencies to use for modeling frequency response
- **distance** – distance in m
- **temperature** – temperature in deg C
- **salinity** – salinity in ppt
- **depth** – depth in m

Returns tap weights for a FIR filter that represents absorption at the given distance

```
>>> import arlpy
>>> import numpy as np
>>> fs = 250000
>>> b = arlpy.uwa.absorption_filter(fs, distance=500)
>>> x = arlpy.signal.sweep(20000, 40000, 0.5, fs)
>>> y = arlpy.signal.lfilter0(b, 1, x)
>>> y /= 500**2      # apply spreading loss for 500m
```

arlpy.uwa.**bubble_resonance**(*radius*, *depth=0.0*, *gamma=1.4*, *p0=101300.0*, *rho_water=1022.476*)

Compute resonance frequency of a freely oscillating has bubble in water, using implementation based on Medwin & Clay (1998). This ignores surface-tension, thermal, viscous and acoustic damping effects, and the pressure-volume relationship is taken to be adiabatic. Parameters:

Radius bubble *radius* in meters

Depth depth of bubble in water in meters

Gamma gas ratio of specific heats. Default 1.4 for air

P0 atmospheric pressure. Default 1.013e5 Pa

Rho_water Density of water. Default 1022.476 kg/m^3

```
>>> import arlpy
>>> arlpy.uwa.bubble_resonance(100e-6)
32465.56
```

arlpy.uwa.**bubble_soundspeed**(*void_fraction*, *c=1539.0866009307247*, *c_gas=340*, *relative_density=1000*)

Get the speed of sound in a 2-phase bubbly water.

The sound speed is computed based on Wood (1964) or Buckingham (1997).

Parameters

- **void_fraction** – void fraction
- **c** – speed of sound in water in m/s
- **c_gas** – speed of sound in gas in m/s
- **relative_density** – ratio of density of water to gas

Returns sound speed in m/s

```
>>> import arlpy
>>> arlpy.uwa.bubble_soundspeed(1e-5)
1402.133
```

arlpy.uwa.**bubble_surface_loss**(*windspeed*, *frequency*, *angle*)
Get the surface loss due to bubbles.

The surface loss is computed based on APL model (1994).

Parameters

- **windspeed** – windspeed in m/s (measured 10 m above the sea surface)
- **frequency** – frequency in Hz
- **angle** – incidence angle in radians

Returns absorption as a linear multiplier

```
>>> import numpy
>>> import arlpy
>>> arlpy.utils.mag2db(uwa.bubble_surface_loss(3,10000,0))
-1.44
>>> arlpy.utils.mag2db(uwa.bubble_surface_loss(10,10000,0))
-117.6
```

arlpy.uwa.**density**(*temperature=27*, *salinity=35*)
Get the density of sea water near the surface.

Computes sea water density using Fofonoff (1985 - IES 80).

Parameters

- **temperature** – temperature in deg C
- **salinity** – salinity in ppt

Returns density in kg/m^3

```
>>> import arlpy
>>> arlpy.uwa.density()
1022.7
```

arlpy.uwa.**doppler**(*speed*, *frequency*, *c=1539.0866009307247*)
Get the Doppler-shifted frequency given relative speed between transmitter and receiver.

The Doppler approximation used is only valid when *speed* << *c*. This is usually the case for underwater vehicles.

Parameters

- **speed** – relative speed between transmitter and receiver in m/s
- **frequency** – transmission frequency in Hz
- **c** – sound speed in m/s

Returns the Doppler shifted frequency as perceived by the receiver

```
>>> import arlpy
>>> arlpy.uwa.doppler(2, 50000)
50064.97
>>> arlpy.uwa.doppler(-1, 50000)
49967.51
```

arlpy.uwa.**pressure**(*x*, *sensitivity*, *gain*, *volt_params=None*)
Convert the real signal x to an acoustic pressure signal in micropascal.

Parameters

- **x** – real signal in voltage or bit depth (number of bits)
- **sensitivity** – receiving sensitivity in dB re 1V per micropascal
- **gain** – preamplifier gain in dB
- **volt_params** – (nbits, v_ref) is used to convert the number of bits to voltage where nbits is the number of bits of each sample and v_ref is the reference voltage, default to None

Returns acoustic pressure signal in micropascal

If *volt_params* is provided, the sample unit of x is in number of bits, else is in voltage.

```
>>> import arlpy
>>> nbits = 16
>>> V_ref = 1.0
>>> x_volt = V_ref*signal.cw(64, 1, 512)
>>> x_bit = x_volt*(2**(nbits-1))
>>> sensitivity = 0
>>> gain = 0
>>> p1 = arlpy.uwa.pressure(x_volt, sensitivity, gain)
>>> p2 = arlpy.uwa.pressure(x_bit, sensitivity, gain, volt_params=(nbits, V_ref))
```

arlpy.uwa.**reflection_coeff**(*angle*, *rho1*, *c1*, *alpha=0*, *rho=1022.7198310217424*, *c=1539.0866009307247*)

Get the Rayleigh reflection coefficient for a given angle.

Parameters

- **angle** – angle of incidence in radians
- **rho1** – density of second medium in kg/m^3
- **c1** – sound speed in second medium in m/s
- **alpha** – attenuation
- **rho** – density of water in kg/m^3
- **c** – sound speed in water in m/s

Returns reflection coefficient as a linear multiplier

```
>>> from numpy import pi
>>> import arlpy
>>> arlpy.uwa.reflection_coeff(pi/4, 1200, 1600)
0.1198
>>> arlpy.uwa.reflection_coeff(0, 1200, 1600)
0.0990
>>> arlpy.utils.mag2db(arlpy.uwa.reflection_coeff(0, 1200, 1600))
-20.1
```

arlpy.uwa.**soundspeed**(*temperature=27*, *salinity=35*, *depth=10*)

Get the speed of sound in water.

Uses Mackenzie (1981) to compute sound speed in water.

Parameters

- **temperature** – temperature in deg C
- **salinity** – salinity in ppt
- **depth** – depth in m

Returns sound speed in m/s

```
>>> import arlpy
>>> arlpy.uwa.soundspeed()
1539.1
>>> arlpy.uwa.soundspeed(temperature=25, depth=20)
1534.6
```

arlpy.uwa.**spl**(*x*, *ref=1*)

Get Sound Pressure Level (SPL) of the acoustic pressure signal x.

Parameters

- **x** – acoustic pressure signal in micropascal
- **ref** – reference acoustic pressure in micropascal, default to 1

Returns average SPL in dB re micropascal

In water, the common reference is 1 micropascal. In air, the common reference is 20 micropascal.

```
>>> import arlpy
>>> p = signal.cw(64, 1, 512)
>>> arlpy.uwa.spl(p)
-3.0103
```

1.7 Underwater acoustic propagation modeling

Underwater acoustic propagation modeling toolbox.

This toolbox currently uses the Bellhop acoustic propagation model. For this model to work, the acoustic toolbox must be installed on your computer and *bellhop.exe* should be in your PATH.

Sample Jupyter notebook

For usage examples of this toolbox, see Bellhop notebook.

arlpy.uwapm.**arrivals_to_impulse_response**(*arrivals*, *fs*, *abs_time=False*)

Convert arrival times and coefficients to an impulse response.

Parameters

- **arrivals** – arrivals times (s) and coefficients
- **fs** – sampling rate (Hz)
- **abs_time** – absolute time (True) or relative time (False)

Returns impulse response

If *abs_time* is set to True, the impulse response is placed such that the zero time corresponds to the time of transmission of signal.

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> arrivals = pm.compute_arrivals(env)
>>> ir = pm.arrivals_to_impulse_response(arrivals, fs=192000)
```

arlpy.uwapm.**check_env2d**(*env*)
Check the validity of a 2D underwater environment definition.

Parameters **env** – environment definition

Exceptions are thrown with appropriate error messages if the environment is invalid.

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> check_env2d(env)
```

arlpy.uwapm.**compute_arrivals**(*env*, *model=None*, *debug=False*)
Compute arrivals between each transmitter and receiver.

Parameters

- **env** – environment definition
- **model** – propagation model to use (None to auto-select)
- **debug** – generate debug information for propagation model

Returns arrival times and coefficients for all transmitter-receiver combinations

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> arrivals = pm.compute_arrivals(env)
>>> pm.plot_arrivals(arrivals)
```

arlpy.uwapm.**compute_eigenrays**(*env*, *tx_depth_ndx=0*, *rx_depth_ndx=0*, *rx_range_ndx=0*, *model=None*, *debug=False*)
Compute eigenrays between a given transmitter and receiver.

Parameters

- **env** – environment definition
- **tx_depth_ndx** – transmitter depth index
- **rx_depth_ndx** – receiver depth index
- **rx_range_ndx** – receiver range index
- **model** – propagation model to use (None to auto-select)
- **debug** – generate debug information for propagation model

Returns eigenrays paths

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> rays = pm.compute_eigenrays(env)
>>> pm.plot_rays(rays, width=1000)
```

arlpy.uwapm.**compute_rays**(*env*, *tx_depth_ndx=0*, *model=None*, *debug=False*)
Compute rays from a given transmitter.

Parameters

- **env** – environment definition
- **tx_depth_ndx** – transmitter depth index
- **model** – propagation model to use (None to auto-select)
- **debug** – generate debug information for propagation model

Returns ray paths

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> rays = pm.compute_rays(env)
>>> pm.plot_rays(rays, width=1000)
```

arlpy.uwapm.**compute_transmission_loss**(*env*, *tx_depth_ndx=0*, *mode='coherent'*, *model=None*, *debug=False*)

Compute transmission loss from a given transmitter to all receviers.

Parameters

- **env** – environment definition
- **tx_depth_ndx** – transmitter depth index
- **mode** – coherent, incoherent or semicoherent
- **model** – propagation model to use (None to auto-select)
- **debug** – generate debug information for propagation model

Returns complex transmission loss at each receiver depth and range

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> tloss = pm.compute_transmission_loss(env, mode=pm.incoherent)
>>> pm.plot_transmission_loss(tloss, width=1000)
```

arlpy.uwapm.**create_env2d**(***kv*)

Create a new 2D underwater environment.

A basic environment is created with default values. To see all the parameters available and their default values:

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> pm.print_env(env)
```

The environment parameters may be changed by passing keyword arguments or modified later using a dictionary notation:

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d(depth=40, soundspeed=1540)
>>> pm.print_env(env)
>>> env['depth'] = 25
>>> env['bottom_soundspeed'] = 1800
>>> pm.print_env(env)
```

The default environment has a constant sound speed. A depth dependent sound speed profile be provided as a Nx2 array of (depth, sound speed):

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d(depth=20, soundspeed=[[0,1540], [5,1535], [10,1535],␣
↪[20,1530]])
```

A range-and-depth dependent sound speed profile can be provided as a Pandas frame:

```
>>> import arlpy.uwapm as pm
>>> import pandas as pd
```

(continues on next page)

(continued from previous page)

```
>>> ssp2 = pd.DataFrame({
          0: [1540, 1530, 1532, 1533],      # profile at 0 m range
        100: [1540, 1535, 1530, 1533],      # profile at 100 m range
        200: [1530, 1520, 1522, 1525] },    # profile at 200 m range
        index=[0, 10, 20, 30])              # depths of the profile entries in m
>>> env = pm.create_env2d(depth=20, soundspeed=ssp2)
```

The default environment has a constant water depth. A range dependent bathymetry can be provided as a Nx2 array of (range, water depth):

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d(depth=[[0,20], [300,10], [500,18], [1000,15]])
```

arlpy.uwapm.**models**(*env=None*, *task=None*)

List available models.

Parameters

- **env** – environment to model
- **task** – arrivals/eigenrays/rays/coherent/incoherent/semicoherent

Returns list of models that can be used

```
>>> import arlpy.uwapm as pm
>>> pm.models()
['bellhop']
>>> env = pm.create_env2d()
>>> pm.models(env, task=coherent)
['bellhop']
```

arlpy.uwapm.**plot_arrivals**(*arrivals*, *dB=False*, *color='blue'*, ***kwargs*)

Plots the arrival times and amplitudes.

Parameters

- **arrivals** – arrivals times (s) and coefficients
- **dB** – True to plot in dB, False for linear scale
- **color** – line color (see Bokeh colors)

Other keyword arguments applicable for *arlpy.plot.plot()* are also supported.

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> arrivals = pm.compute_arrivals(env)
>>> pm.plot_arrivals(arrivals)
```

arlpy.uwapm.**plot_env**(*env*, *surface_color='dodgerblue'*, *bottom_color='peru'*, *tx_color='orangered'*, *rx_color='midnightblue'*, *rx_plot=None*, ***kwargs*)

Plots a visual representation of the environment.

Parameters

- **env** – environment description
- **surface_color** – color of the surface (see Bokeh colors)
- **bottom_color** – color of the bottom (see Bokeh colors)
- **tx_color** – color of transmitters (see Bokeh colors)

- **rx_color** – color of receviers (see Bokeh colors)
- **rx_plot** – True to plot all receivers, False to not plot any receivers, None to automatically decide

Other keyword arguments applicable for *arlpy.plot.plot()* are also supported.

The surface, bottom, transmitters (marker: '*') and receivers (marker: 'o') are plotted in the environment. If *rx_plot* is set to None and there are more than 2000 receivers, they are not plotted.

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d(depth=[[0, 40], [100, 30], [500, 35], [700, 20], [1000,
↪45]])
>>> pm.plot_env(env)
```

arlpy.uwapm.**plot_rays**(*rays*, *env=None*, *invert_colors=False*, ***kwargs*)

Plots ray paths.

Parameters

- **rays** – ray paths
- **env** – environment definition
- **invert_colors** – False to use black for high intensity rays, True to use white

If environment definition is provided, it is overlayed over this plot using default parameters for *arlpy.uwapm.plot_env()*.

Other keyword arguments applicable for *arlpy.plot.plot()* are also supported.

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d()
>>> rays = pm.compute_eigenrays(env)
>>> pm.plot_rays(rays, width=1000)
```

arlpy.uwapm.**plot_ssp**(*env*, ***kwargs*)

Plots the sound speed profile.

Parameters **env** – environment description

Other keyword arguments applicable for *arlpy.plot.plot()* are also supported.

If the sound speed profile is range-dependent, this function only plots the first profile.

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d(soundspeed=[[ 0, 1540], [10, 1530], [20, 1532], [25,␣
↪1533], [30, 1535]])
>>> pm.plot_ssp(env)
```

arlpy.uwapm.**plot_transmission_loss**(*tloss*, *env=None*, ***kwargs*)

Plots transmission loss.

Parameters

- **tloss** – complex transmission loss
- **env** – environment definition

If environment definition is provided, it is overlayed over this plot using default parameters for *arlpy.uwapm.plot_env()*.

Other keyword arguments applicable for *arlpy.plot.image()* are also supported.

```
>>> import arlpy.uwapm as pm
>>> import numpy as np
>>> env = pm.create_env2d(
        rx_depth=np.arange(0, 25),
        rx_range=np.arange(0, 1000),
        min_angle=-45,
        max_angle=45
    )
>>> tloss = pm.compute_transmission_loss(env)
>>> pm.plot_transmission_loss(tloss, width=1000)
```

arlpy.uwapm.**print_env**(*env*)

Display the environment in a human readable form.

Parameters **env** – environment definition

```
>>> import arlpy.uwapm as pm
>>> env = pm.create_env2d(depth=40, soundspeed=1540)
>>> pm.print_env(env)
```

1.8 Plotting utilities

Easy-to-use plotting utilities based on Bokeh.

arlpy.plot.**box**(*left=None*, *right=None*, *top=None*, *bottom=None*, *color='yellow'*, *alpha=0.1*, *hold=False*)

Add a highlight box to a plot.

Parameters

- **left** – x location of left of box
- **right** – x location of right of box
- **top** – y location of top of box
- **bottom** – y location of bottom of box
- **color** – text color (see Bokeh colors)
- **alpha** – transparency (0-1)
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> arlpy.plot.plot([0, 20], [0, 10], hold=True)
>>> arlpy.plot.box(left=5, right=10, top=8)
```

arlpy.plot.**color**(*n*)

Get a numbered color to cycle over a set of colors.

```
>>> import arlpy.plot
>>> arlpy.plot.color(0)
'blue'
>>> arlpy.plot.color(1)
'red'
>>> arlpy.plot.plot([0, 20], [0, 10], color=arlpy.plot.color(3))
```

arlpy.plot.**enable_javascript**(*b*)
Enable/disable Javascript.

Parameters **b** – True to use Javacript, False to avoid use of Javascript

Jupyterlab does not support Javascript output. To avoid error messages, Javascript can be disabled using this call. This removes an optimization to replace non-interactive plots with static images, but other than that does not affect functionality.

arlpy.plot.**figsize**(*x*, *y*)
Set the default figure size in pixels.

Parameters

- **x** – figure width
- **y** – figure height

class arlpy.plot.**figure**(*title=None*, *xlabel=None*, *ylabel=None*, *xlim=None*, *ylim=None*, *xtype='auto'*, *ytype='auto'*, *width=None*, *height=None*, *interactive=None*)
Create a new figure, and optionally automatically display it.

Parameters

- **title** – figure title
- **xlabel** – x-axis label
- **ylabel** – y-axis label
- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **xtype** – x-axis type ('auto', 'linear', 'log', etc)
- **ytype** – y-axis type ('auto', 'linear', 'log', etc)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **interactive** – enable interactive tools (pan, zoom, etc) for plot

This function can be used in standalone mode to create a figure:

```
>>> import arlpy.plot
>>> arlpy.plot.figure(title='Demo 1', width=500)
>>> arlpy.plot.plot([0,10], [0,10])
```

Or it can be used as a context manager to create, hold and display a figure:

```
>>> import arlpy.plot
>>> with arlpy.plot.figure(title='Demo 2', width=500):
>>>     arlpy.plot.plot([0,10], [0,10], color='blue', legend='A')
>>>     arlpy.plot.plot([10,0], [0,10], marker='o', color='green', legend='B')
```

It can even be used as a context manager to work with Bokeh functions directly:

```
>>> import arlpy.plot
>>> with arlpy.plot.figure(title='Demo 3', width=500) as f:
>>>     f.line([0,10], [0,10], line_color='blue')
>>>     f.square([3,7], [4,5], line_color='green', fill_color='yellow', size=10)
```

arlpy.plot.**freqz**(*b*, *a=1*, *fs=2.0*, *worN=None*, *whole=False*, *degrees=True*, *style='solid'*, *thickness=1*, *title=None*, *xlabel='Frequency (Hz)'*, *xlim=None*, *ylim=None*, *width=None*, *height=None*, *hold=False*, *interactive=None*)

Plot frequency response of a filter.

This is a convenience function to plot frequency response, and internally uses `scipy.signal.freqz()` to estimate the response. For further details, see the documentation for `scipy.signal.freqz()`.

Parameters

- **b** – numerator of a linear filter
- **a** – denominator of a linear filter
- **fs** – sampling rate in Hz (optional, normalized frequency if not specified)
- **worN** – see `scipy.signal.freqz()`
- **whole** – see `scipy.signal.freqz()`
- **degrees** – True to display phase in degrees, False for radians
- **style** – line style ('solid', 'dashed', 'dotted', 'dotdash', 'dashdot')
- **thickness** – line width in pixels
- **title** – figure title
- **xlabel** – x-axis label
- **ylabel1** – y-axis label for magnitude
- **ylabel2** – y-axis label for phase
- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **interactive** – enable interactive tools (pan, zoom, etc) for plot
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy
>>> arlpy.plot.freqz([1,1,1,1,1], fs=120000);
```

arlpy.plot.**gcf**()

Get the current figure.

Returns handle to the current figure

arlpy.plot.**hlines**(*y*, *color='gray'*, *style='dashed'*, *thickness=1*, *hold=False*)

Draw horizontal lines on a plot.

Parameters

- **y** – y location of lines
- **color** – line color (see Bokeh colors)
- **style** – line style ('solid', 'dashed', 'dotted', 'dotdash', 'dashdot')
- **thickness** – line width in pixels
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> arlpy.plot.plot([0, 20], [0, 10], hold=True)
>>> arlpy.plot.hlines(3, color='red', style='dotted')
```

arlpy.plot.**hold**(*enable=True*)

Combine multiple plots into one.

Parameters **enable** – True to hold plot, False to release hold

Returns old state of hold if enable is True

```
>>> import arlpy.plot
>>> oh = arlpy.plot.hold()
>>> arlpy.plot.plot([0,10], [0,10], color='blue', legend='A')
>>> arlpy.plot.plot([10,0], [0,10], marker='o', color='green', legend='B')
>>> arlpy.plot.hold(oh)
```

arlpy.plot.**image**(*img*, *x=None*, *y=None*, *colormap='Plasma256'*, *clim=None*, *clabel=None*, *title=None*, *xlabel=None*, *ylabel=None*, *xlim=None*, *ylim=None*, *xtype='auto'*, *ytype='auto'*, *width=None*, *height=None*, *hold=False*, *interactive=None*)

Plot a heatmap of 2D scalar data.

Parameters

- **img** – 2D image data
- **x** – x-axis range for image data (min, max)
- **y** – y-axis range for image data (min, max)
- **colormap** – named color palette or Bokeh ColorMapper (see Bokeh palettes)
- **clim** – color axis limits (min, max)
- **clabel** – color axis label
- **title** – figure title
- **xlabel** – x-axis label
- **ylabel** – y-axis label
- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **xtype** – x-axis type ('auto', 'linear', 'log', etc)
- **ytype** – y-axis type ('auto', 'linear', 'log', etc)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **interactive** – enable interactive tools (pan, zoom, etc) for plot
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> import numpy as np
>>> arlpy.plot.image(np.random.normal(size=(100,100)), colormap='Inferno256')
```

arlpy.plot.**interactive**(*b*)

Set default interactivity for plots.

Parameters **b** – True to enable interactivity, False to disable it

arlpy.plot.**iqplot**(*data, marker='.', color=None, labels=None, filled=False, size=None, title=None, xlabel=None, ylabel=None, xlim=[-2, 2], ylim=[-2, 2], width=None, height=None, hold=False, interactive=None*)

Plot signal points.

Parameters

- **data** – complex baseband signal points
- **marker** – point markers ('.', 'o', 's', '*', 'x', '+', 'd', '^')
- **color** – marker/text color (see Bokeh colors)
- **labels** – label for each signal point, or True to auto-generate labels
- **filled** – filled markers or outlined ones
- **size** – marker/text size (e.g. 5, '8pt')
- **title** – figure title
- **xlabel** – x-axis label
- **ylabel** – y-axis label
- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **interactive** – enable interactive tools (pan, zoom, etc) for plot
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy
>>> import arlpy.plot
>>> arlpy.plot.iqplot(arlpy.comms.psk(8))
>>> arlpy.plot.iqplot(arlpy.comms.qam(16), color='red', marker='x')
>>> arlpy.plot.iqplot(arlpy.comms.psk(4), labels=['00', '01', '11', '10'])
```

class arlpy.plot.**many_figures**(*figsize=None*)

Create a grid of many figures.

Parameters **figsize** – default size of figure in grid as (width, height)

```
>>> import arlpy.plot
>>> with arlpy.plot.many_figures(figsize=(300,200)):
>>>     arlpy.plot.plot([0,10], [0,10])
>>>     arlpy.plot.plot([0,10], [0,10])
>>>     arlpy.plot.next_row()
>>>     arlpy.plot.next_column()
>>>     arlpy.plot.plot([0,10], [0,10])
```

arlpy.plot.**next_column**()

Move to the next column in a grid of many figures.

arlpy.plot.**next_row**()

Move to the next row in a grid of many figures.

arlpy.plot.**plot**(*x*, *y=None*, *fs=None*, *maxpts=10000*, *pooling=None*, *color=None*, *style='solid'*, *thickness=1*, *marker=None*, *filled=False*, *size=6*, *mskip=0*, *title=None*, *xlabel=None*, *ylabel=None*, *xlim=None*, *ylim=None*, *xtype='auto'*, *ytype='auto'*, *width=None*, *height=None*, *legend=None*, *hold=False*, *interactive=None*)

Plot a line graph or time series.

Parameters

- **x** – x data or time series data (if y is None)
- **y** – y data or None (if time series)
- **fs** – sampling rate for time series data
- **maxpts** – maximum number of points to plot (downsampled if more points provided)
- **pooling** – pooling for downsampling (None, 'max', 'min', 'mean', 'median')
- **color** – line color (see Bokeh colors)
- **style** – line style ('solid', 'dashed', 'dotted', 'dotdash', 'dashdot', None)
- **thickness** – line width in pixels
- **marker** – point markers ('.', 'o', 's', '*', 'x', '+', 'd', '^')
- **filled** – filled markers or outlined ones
- **size** – marker size
- **mskip** – number of points to skip marking (to avoid too many markers)
- **title** – figure title
- **xlabel** – x-axis label
- **ylabel** – y-axis label
- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **xtype** – x-axis type ('auto', 'linear', 'log', etc)
- **ytype** – y-axis type ('auto', 'linear', 'log', etc)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **legend** – legend text
- **interactive** – enable interactive tools (pan, zoom, etc) for plot
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> import numpy as np
>>> arlpy.plot.plot([0,10], [1,-1], color='blue', marker='o', filled=True, legend=
↪'A', hold=True)
>>> arlpy.plot.plot(np.random.normal(size=1000), fs=100, color='green', legend='B
↪')
```

arlpy.plot.**psd**(*x*, *fs=2*, *nfft=512*, *noverlap=None*, *window='hann'*, *color=None*, *style='solid'*, *thickness=1*, *marker=None*, *filled=False*, *size=6*, *title=None*, *xlabel='Frequency (Hz)'*, *ylabel='Power spectral density (dB/Hz)'*, *xlim=None*, *ylim=None*, *width=None*, *height=None*, *legend=None*, *hold=False*, *interactive=None*)

Plot power spectral density of a given time series signal.

Parameters

- **x** – time series signal
- **fs** – sampling rate
- **nfft** – segment size (see scipy.signal.welch)
- **noverlap** – overlap size (see scipy.signal.welch)
- **window** – window to use (see scipy.signal.welch)
- **color** – line color (see Bokeh colors)
- **style** – line style ('solid', 'dashed', 'dotted', 'dotdash', 'dashdot')
- **thickness** – line width in pixels
- **marker** – point markers ('.', 'o', 's', '*', 'x', '+', 'd', '^')
- **filled** – filled markers or outlined ones
- **size** – marker size
- **title** – figure title
- **xlabel** – x-axis label
- **ylabel** – y-axis label
- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **legend** – legend text
- **interactive** – enable interactive tools (pan, zoom, etc) for plot
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> import numpy as np
>>> arlpy.plot.psd(np.random.normal(size=(10000)), fs=10000)
```

arlpy.plot.**scatter**(*x*, *y*, *marker='.'*, *filled=False*, *size=6*, *color=None*, *title=None*, *xlabel=None*, *ylabel=None*, *xlim=None*, *ylim=None*, *xtype='auto'*, *ytype='auto'*, *width=None*, *height=None*, *legend=None*, *hold=False*, *interactive=None*)

Plot a scatter plot.

Parameters

- **x** – x data
- **y** – y data
- **color** – marker color (see Bokeh colors)
- **marker** – point markers ('.', 'o', 's', '*', 'x', '+', 'd', '^')

- **filled** – filled markers or outlined ones
- **size** – marker size
- **title** – figure title
- **xlabel** – x-axis label
- **ylabel** – y-axis label
- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **xtype** – x-axis type ('auto', 'linear', 'log', etc)
- **ytype** – y-axis type ('auto', 'linear', 'log', etc)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **legend** – legend text
- **interactive** – enable interactive tools (pan, zoom, etc) for plot
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> import numpy as np
>>> arlpy.plot.scatter(np.random.normal(size=100), np.random.normal(size=100),␣
↪color='blue', marker='o')
```

arlpy.plot.**set_colors**(*c*)

Provide a list of named colors to cycle over.

```
>>> import arlpy.plot
>>> arlpy.plot.set_colors(['red', 'blue', 'green', 'black'])
>>> arlpy.plot.color(2)
'green'
```

arlpy.plot.**specgram**(*x*, *fs=2*, *nfft=None*, *noverlap=None*, *colormap='Plasma256'*, *clim=None*, *clabel='dB'*, *title=None*, *xlabel='Time (s)'*, *ylabel='Frequency (Hz)'*, *xlim=None*, *ylim=None*, *width=None*, *height=None*, *hold=False*, *interactive=None*)

Plot spectrogram of a given time series signal.

Parameters

- **x** – time series signal
- **fs** – sampling rate
- **nfft** – FFT size (see scipy.signal.spectrogram)
- **noverlap** – overlap size (see scipy.signal.spectrogram)
- **colormap** – named color palette or Bokeh ColorMapper (see Bokeh palettes)
- **clim** – color axis limits (min, max), or dynamic range with respect to maximum
- **clabel** – color axis label
- **title** – figure title
- **xlabel** – x-axis label
- **ylabel** – y-axis label

- **xlim** – x-axis limits (min, max)
- **ylim** – y-axis limits (min, max)
- **width** – figure width in pixels
- **height** – figure height in pixels
- **interactive** – enable interactive tools (pan, zoom, etc) for plot
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> import numpy as np
>>> arlpy.plot.specgram(np.random.normal(size=(10000)), fs=10000, clim=30)
```

arlpy.plot.**text**(*x*, *y*, *s*, *color='gray'*, *size='8pt'*, *hold=False*)
Add text annotation to a plot.

Parameters

- **x** – x location of left of text
- **y** – y location of bottom of text
- **s** – text to add
- **color** – text color (see Bokeh colors)
- **size** – text size (e.g. '12pt', '3em')
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> arlpy.plot.plot([0, 20], [0, 10], hold=True)
>>> arlpy.plot.text(7, 3, 'demo', color='orange')
```

arlpy.plot.**theme**(*name*)
Set color theme.

Parameters **name** – name of theme

```
>>> import arlpy.plot
>>> arlpy.plot.theme('dark')
```

arlpy.plot.**use_static_images**(*b=True*)
Use static images instead of dynamic HTML/Javascript in Jupyter notebook.

Parameters **b** – True to use static images, False to use HTML/Javascript

Static images are useful when the notebook is to be exported as a markdown, LaTeX or PDF document, since dynamic HTML/Javascript is not rendered in these formats. When static images are used, all interactive functionality is disabled.

To use static images, you must have the following packages installed: selenium, pillow, phantomjs.

arlpy.plot.**vlines**(*x*, *color='gray'*, *style='dashed'*, *thickness=1*, *hold=False*)
Draw vertical lines on a plot.

Parameters

- **x** – x location of lines
- **color** – line color (see Bokeh colors)
- **style** – line style ('solid', 'dashed', 'dotted', 'dotdash', 'dashdot')

- **thickness** – line width in pixels
- **hold** – if set to True, output is not plotted immediately, but combined with the next plot

```
>>> import arlpy.plot
>>> arlpy.plot.plot([0, 20], [0, 10], hold=True)
>>> arlpy.plot.vlines([7, 12])
```

1.9 Common utilities

Common utility functions.

arlpy.utils.**broadcastable_to**(*x*, *shape*, *axis=None*)
Convert 1D array to be broadcastable along specified axis.

Parameters

- **x** – array to broadcast
- **shape** – shape to broadcast to
- **axis** – axis to broadcast along

Reshapes the array to the minimum dimensions such that it can be broadcasted to the given shape along the specified axis. If an axis is not specified, the first axis that matches the size of x is used.

```
>>> import arlpy
>>> import numpy as np
>>> arlpy.utils.broadcastable_to(np.array([1,2,3]), (5,3,2), 1)
array([[1],
       [2],
       [3]])
```

arlpy.utils.**db2mag**(*x*)
Convert dB quantity to magnitude.

arlpy.utils.**db2pow**(*x*)
Convert dB quantity to power.

arlpy.utils.**linspace2d**(*start0*, *stop0*, *num0*, *start1*, *stop1*, *num1*)
Generate linearly spaced coordinates in 2D space.

Parameters

- **start0** – first value on axis 0
- **stop0** – last value on axis 0
- **num0** – number of values on axis 0
- **start1** – first value on axis 1
- **stop1** – last value on axis 1
- **num1** – number of values on axis 1

```
>>> from arlpy import bf
>>> bf.linspace2d(0, 1, 2, 0, 1, 3)
[[ 0. ,  0. ],
 [ 0. ,  0.5],
 [ 0. ,  1. ],
```

(continues on next page)

(continued from previous page)

```
 [ 1. ,  0. ],
 [ 1. ,  0.5],
 [ 1. ,  1. ]]
```

arlpy.utils.**linspace3d**(*start0*, *stop0*, *num0*, *start1*, *stop1*, *num1*, *start2*, *stop2*, *num2*)

Generate linearly spaced coordinates in 2D space.

Parameters

- **start0** – first value on axis 0
- **stop0** – last value on axis 0
- **num0** – number of values on axis 0
- **start1** – first value on axis 1
- **stop1** – last value on axis 1
- **num1** – number of values on axis 1
- **start2** – first value on axis 2
- **stop2** – last value on axis 2
- **num2** – number of values on axis 2

```
>>> from arlpy import bf
>>> bf.linspace3d(0, 1, 2, 0, 1, 3, 0, 0, 1)
[[ 0. ,  0. , 0. ],
 [ 0. ,  0.5, 0. ],
 [ 0. ,  1. , 0. ],
 [ 1. ,  0. , 0. ],
 [ 1. ,  0.5, 0. ],
 [ 1. ,  1. , 0. ]]
```

arlpy.utils.**mag2db**(*x*)

Convert magnitude quantity to dB.

arlpy.utils.**pow2db**(*x*)

Convert power quantity to dB.

arlpy.utils.**progress**(*n*, *width=50*)

Display progress bar for long running operations.

Parameters

- **n** – total number of steps to completion
- **width** – width of the progress bar (only for the text version)

```
>>> import arlpy
>>> progress = arlpy.utils.progress(100)
>>> for j in range(100):
        next(progress)
```

arlpy.utils.**rotation_matrix**(*alpha*, *beta*, *gamma*)

Generates a 3D rotation matrix.

Parameters

- **alpha** – rotation angle around x-axis

- **beta** – rotation angle around y-axis
- **gamma** – rotation angle around z-axis

Rotation is applied around x, y and z axis in that order.

CHAPTER 2

Special-purpose modules

The following modules are specific to tools available at the ARL and may not be of general interest to others:

2.1 Digital Towed Array

DTLA support toolbox.

`arlpy.dtla.`**`check`**(*filename*)
Check if a file is likely to be a valid DTLA datafile.

`arlpy.dtla.`**`get_channels`**(*filename=None*)
Get the number of available data channels.

`arlpy.dtla.`**`get_data`**(*filename*, *channel=None*, *start=0*, *length=None*, *detrend='linear'*)
Load selected data from DTLA recording.

Parameters

- **`filename`** – name of the datafile
- **`channel`** – list of channels to read (base 0, None to read all channels)
- **`start`** – sample index to start from
- **`length`** – number of samples to read (None means read all available samples)
- **`detrend`** – processing to be applied to each channel to remove offset/bias (supported values: `'linear'`, `'constant'`, `None`)

`arlpy.dtla.`**`get_data_length`**(*filename*)
Get the length of the datafile in samples.

`arlpy.dtla.`**`get_sampling_rate`**(*filename=None*)
Get the sampling rate in Hz.

2.2 ROMANIS

ROMANIS support toolbox.

arlpy.romanis.**get_channels**(*dirname=None*)
Get the number of available data channels.

Parameters **dirname** – directory of the datafile

Returns number of channels

arlpy.romanis.**get_data**(*dirname*, *start=0*, *length=None*, *calib=None*, *sensor=None*)
Load selected data from ROMANIS recording.

Parameters

- **dirname** – directory of the datafile
- **start** – start time (seconds, default is 0)
- **length** – data length (seconds, None means read all)
- **calib** – directory of the calibration file (None means no calibration)
- **sensor** – sensor number to load (None means load all, 0-507)
- **returns** – 2-D array of the selected data

arlpy.romanis.**get_data_length**(*dirname*)
Get the length of the datafile in seconds.

Parameters **dirname** – directory of the datafile

Returns data length in seconds

arlpy.romanis.**get_sampling_rate**(*dirname=None*)
Get the sampling rate in Hz.

Parameters **dirname** – directory of the datafile

Returns sampling rate in Hz

2.3 High frequency data acquisition system

HiDAQ support toolbox.

arlpy.hidaq.**check**(*filename*)
Check if a file is likely to be a valid HiDAQ datafile.

arlpy.hidaq.**get_channels**(*filename=None*)
Get the number of available data channels.

arlpy.hidaq.**get_data**(*filename*, *channel=None*, *start=0*, *length=None*, *detrend=None*)
Load selected data from HiDAQ recording.

Parameters

- **filename** – name of the datafile
- **channel** – list of channels to read (base 0, None to read all channels)
- **start** – sample index to start from
- **length** – number of samples to read (None means read all available samples)

- **detrend** – processing to be applied to each channel to remove offset/bias (supported values: `'linear'`, `'constant'`, `None`)

`arlpy.hidaq.`**`get_data_length`**(*filename*)
Get the length of the datafile in samples.

`arlpy.hidaq.`**`get_sampling_rate`**(*filename=None*)
Get the sampling rate in Hz.

2.4 Unet modem & network stack

UNET support toolbox.

`arlpy.unet.`**`get_signal`**(*signals*, *n*, *order='F'*)
Gets the specified signal from the list of signals.

Parameters

- **signals** – table of signals returned by get_signals()
- **n** – signal index or signal selector
- **order** – ordering for multi-channel signals ('C' or 'F', see numpy.reshape())

Returns signal array

```
>>> import arlpy.unet
>>> s = arlpy.unet.get_signals('signals-0.txt')
>>> x = arlpy.unet.get_signal(s, 2)
>>> y = arlpy.unet.get_signal(s, s.rxtime == 123374675)
```

`arlpy.unet.`**`get_signals`**(*filename*)
Get a list of signals in a signals file.

Parameters **filename** – name of signals file with RxBasebandSignalNtfs

Returns table of signals

`arlpy.unet.`**`read_signals`**(*filename*, *callback*, *filter=None*, *order='F'*)
Read a signals file and call callback for each signal.

The callback function is called for each signal with a dictionary containing header information and the extracted signal.

If a filter function is specified, it is called for each signal header. The function should return True if the signal should be extracted, False otherwise.

Parameters

- **filename** – name of signals file with RxBasebandSignalNtfs
- **callback** – callback to call with each signal
- **filter** – callback to decide if a signal is extracted, or None
- **order** – ordering for multi-channel signals ('C' or 'F', see numpy.reshape())

```
>>> import arlpy.unet
>>> arlpy.unet.read_signals('signals-0.txt', lambda hdr, x: print(hdr, x.shape))
>>> arlpy.unet.read_signals('signals-0.txt', lambda hdr, x: print(hdr, x.shape),␣
↪lambda hdr: hdr['fc']==0)
```

Python Module Index

a

Index

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

Z