bitis

version 0.12.3

Fabrizio Pollastri

December 09, 2014

Contents

Contents 1

- Bitis, binary timed signals processing library 1
  - Introduction 1
  - Requirements 1
  - Installation 1
  - Code Repository 1
- Usage examples 1
  - Logic operations 1
  - Graphic and semigraphic plot 2
  - Correlation Function 4
  - Serial signal 5
  - Phase lockin 6
  - Modulation 8
- Module reference 11
  - Objects and methods 12
  - Functions 15
- BTS format 16
  - Definition 16
  - Python implementation 17
- Pre version 0.9.0 format 17
  - Pre 0.9.0 definition 17
  - Pre 0.9.0 python implementation 17
- Changes 17
  - Release 0.12.3 (released 9-Dec-2014) 17
    - Changes 17
    - Bugs fixed 17
    - Documentation 18
  - Release 0.12.2 (released 6-Dec-2014) 18
    - Bugs fixed 18
    - Documentation 18
  - Release 0.12.1 (released 3-Dec-2014) 18
    - New features 18
    - Bugs fixed 18
  - Release 0.12.0 (released 1-Dec-2014) 18
    - New features 18
    - Changes 18
    - Internals 18
    - Documentation 18
  - Release 0.11.2 (released 8-Oct-2014) 19
    - Bugs fixed 19

Release 0.11.1 (released 6-Oct-2014) 19
- Changes 19
- Bugs fixed 19
- Internals 19

Release 0.11.0 (released 1-Oct-2014) 19
- Features added 19
- Changes 19
- Bugs fixed 19
- Documentation 19

Release 0.10.0 (released 26-Sep-2014) 19
- Features added 19
- Changes 20
- Bugs fixed 20
- Documentation 20
- Internals 20

Release 0.9.0 (released 10-Sep-2014) 20
- Features added 20
- Changes 20
- Bugs fixed 21
- Internals 21

Release 0.8.0 (released 26-Aug-2014) 21
- Features added 21
- Changes 21
- Bugs fixed 21
- Internals 21

Release 0.7.1 (released 3-Feb-2014) 21
- Bugs fixed 22
- Internals 22

Release 0.7.0 (released 27-Jan-2014) 22
- Features added 22
- Incompatible changes 22

Release 0.6.0 (released 16-Dec-2013) 22
- Features added 22
- Incompatible changes 22
- Bugs fixed 22
- Internals 22

Release 0.5.0 (released 9-Dec-2013) 22
- Features added 23
- Incompatible changes 23
- Bugs fixed 23
- Internals 23

Release 0.4.0 (released 2-Dec-2013) 23

Features added 23
Incompatible changes 23
Bugs fixed 23
Internals 23
Release 0.3.0 (released 11-Nov-2013) 23
Features added 23
Release 0.2.0 (released 4-Nov-2013) 24
Features added 24
Release 0.1.0 (released 29-Oct-2013) 24
Index **25**

Contents

Bitis, binary timed signals processing library

Introduction

Bitis is a python module that implements a full set of operators over binary signals represented with BTS format. The BTS format is a computer memory representation of a binary signal that can have a very compact memory footprint when the signal has a low rate of change with respect to its sampling period.

For example, let see a typical case, a time reference signal having about one pulse per second and one microsecond of time resolution. The BTS format allows to completely discard the one million samples per second between each two pulses and allows to keep in memory only the signal change times: for each second, the time of the pulse front edge and the time of the trailing edge.

This is the documentation for version 0.12.3.

Since version 0.9.0, the BTS format has changed. The start and the end times of the signal are no more in the signal changes times sequence. Now, they are attributes of the signal object (Signal.start, Signal.end).`

At present, no effort is made for speed optimization and the employed algorithms are essentially procedural. The only goal is "make it work in some way" and understand what can be a decent set of objects/methods/functions.

BITIS is released under the GNU General Public License.

At present, version 0.12.3, BITIS is in alpha status. Any debugging aid is welcome.

For any question, suggestion, contribution contact the author Fabrizio Pollastri <f.pollastri_a_t_inrim.it>.

Requirements

To run the code, **Python 2.6 or later** must already be installed. The latest release is recommended. Python is available from http://www.python.org/.

When the Signal plotting method is used also Matplotlib is required. This also requires all dependencies of Matplotlib, like NumPy, etc.

Installation

1. Open a shell.
2. Get root privileges and install the package. Command:

```
pip install bitis
```

Code Repository

There is also a code repository at https://github.com/fabriziop/bitis .

Usage examples

Logic operations

This simple example shows some logic operations supported by the **BITIS** module.

```
1 import bitis as bt
2
3 ## Check the equation: a xor b = a and not b  or  not a and b
4
5 # create two random signals
6 a = bt.noise(0,0,100,period_mean=10,width_mean=3)
```

```
 7 b = bt.noise(-10,-10,90,period_mean=4,width_mean=2)
 8
 9 # direct xor
10 xor1 = a ^ b
11
12 # xor from equation
13 xor2 = a & ~b | ~a & b
14
15 # check results
16 if xor1 == xor2:
17     print 'Success!'
18 else:
19     print 'Failure!'
20
21 #### END
```

Graphic and semigraphic plot

The following example shows the plotting capabilitites of methods *plot* and *plotchar*. The method *plot* uses the matplotlib to produce graphic drawing of the given signal as a square/rectangular wave. The x axis represents the time, the y axis represents the logical levels. The method *plotchar* uses the box line drawing characters from unicode for drawing the best approximation of a graphic representation of the given signal. Below there are two representations of the same test signal.

```
 1 import bitis as bt
 2 import locale
 3 import matplotlib.pyplot as pl
 4 import sys
 5 import StringIO as SI
 6
 7 # init locale
 8 locale.setlocale(locale.LC_ALL,"")
 9
10 # a test signal
11 signal = bt.test()
12
13 # graphic plot
14 fig1 = pl.figure(1,figsize=(5,2))
15 pl.suptitle('BITIS: test signal graphic plot.')
16 pl.xlabel('time')
17 signal.plot()
18 pl.grid()
19
20 # save graphic plot to file
21 fig1.savefig('plot.png',format='png')
22
23 # sequence of semigraphic plots of increasing resolution
24 buf = SI.StringIO()
25 buf.write('BITIS: test signal semigraphic plot\n')
26 for width in range(1,77,5):
27     top, bot = signal.plotchar(width,max_flat=4)
28     buf.write('%3d ' % width + top + '\n')
29     buf.write('    ' + bot + '\n')
30 sys.stdout.write(buf.getvalue())
31
32 # save semigraphic plot to file
33 pfile = open('plot.txt','w')
34 pfile.write(buf.getvalue())
35 pfile.close()
36
```

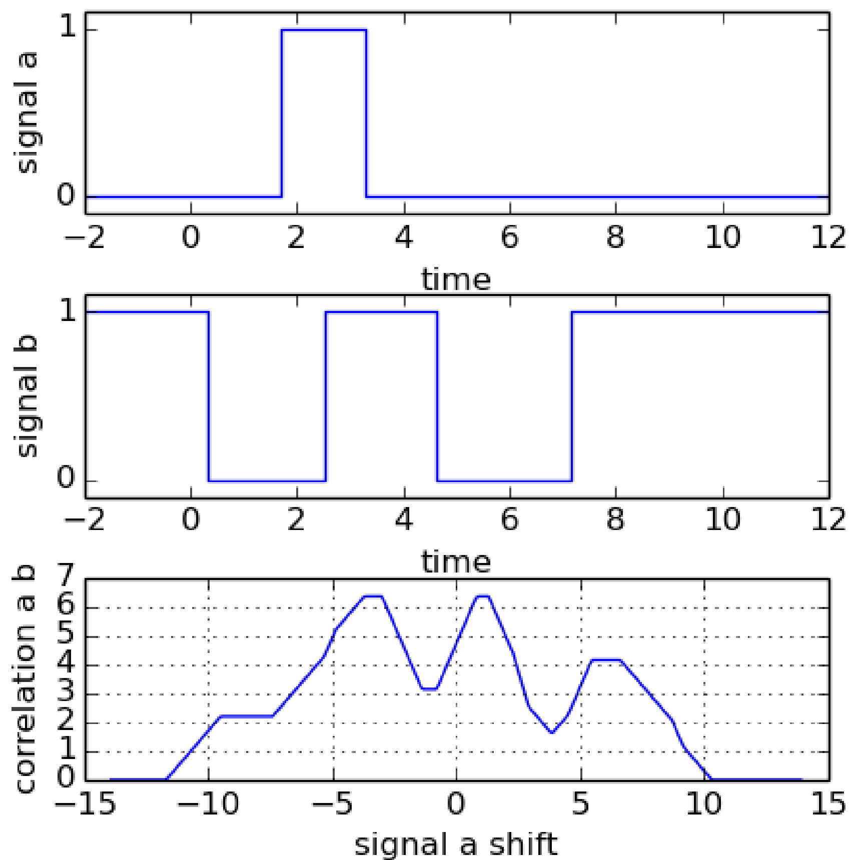
In this example, *plotchar* is called with the argument max_flat=4 . This means that a signal constant level elapsing more than 4 characters is compressed (in time) to be of length 4 characters. This characters drop is marked by the 'x' chararacter that can be seen in the last four semigraphic plots. When this happens, it is important to keep in mind that the x axis time scale is no more uniform.

Correlation Function

The following example shows the plotting of two random signals and their correlation function.

```
1  import bitis as bt
2  import random
3
4  import matplotlib.pyplot as pl
5
6  # make repeatable random sequences
7  random.seed(1)
8
9  # create random signals
10 in_a = bt.noise(-2,-2,12,period_mean=6,width_mean=3)
11 in_b = bt.noise(-2,-2,12,period_mean=4,width_mean=2)
12
13 # compute correlation
14 corr_ab = in_a.correlation(in_b,step_size=0.1)
15
16 # start plotting
17 fig1 = pl.figure(1,figsize=(5,5))
18 pl.suptitle('BITIS: correlation of two signals.')
19
20 # plot signal a
21 pl.subplot(3,1,1)
22 pl.xlim(-2,12)
23 pl.ylabel('signal a')
24 pl.xlabel('time')
25 in_a.plot()
26
27 # plot signal b
28 pl.subplot(3,1,2)
29 pl.xlim(-2,12)
30 pl.ylabel('signal b')
31 pl.xlabel('time')
32 in_b.plot()
33
34 # plot correlation function
35 pl.subplot(3,1,3)
36 pl.grid()
37 corr, shift = corr_ab
38 pl.plot(shift,corr)
39 pl.ylabel('correlation a b')
40 pl.xlabel('signal a shift')
41 pl.subplots_adjust(hspace=0.4)
42
43 # save plot to file
44 fig1.savefig('correlation.png',format='png',)
45
46
47 if __name__ == '__main__':
48     pl.show()
49
50 #### END
```

This is the plotting result.



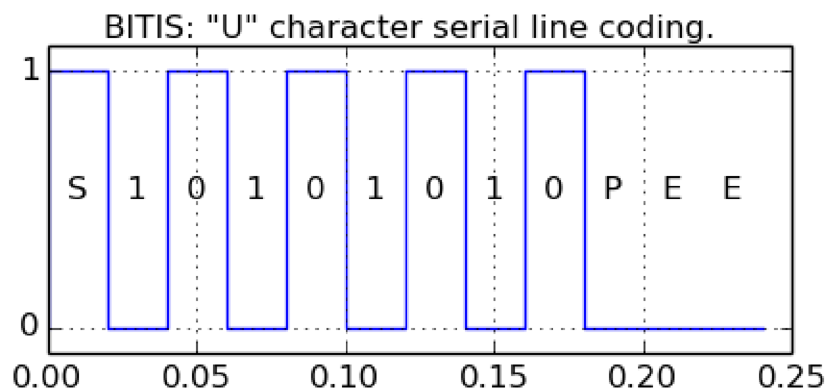
Serial signal

The following example shows the signal of an asynchronous serial interface coding the ASCII character "U" with 8 character bits, odd parity, 2 stop bits and 50 baud tranmitting speed.

```
 1 import bitis as bt
 2 import matplotlib.pyplot as pl
 3
 4 CHAR_BITS = 8
 5 PARITY = 'odd'
 6 STOP_BITS = 2
 7 BAUD = 50
 8 TSCALE = 1.
 9
10 chars = ['U']
11 timings = [0]
12
13 fig1 = pl.figure(1,figsize=(5,2))
14 pl.suptitle('BITIS: "U" character serial line coding.')
15 pl.xlabel('time')
16 bt.serial_tx(chars,timings,char_bits=CHAR_BITS,parity=PARITY,
17         stop_bits=STOP_BITS,baud=BAUD).plot()
18 bit_time = TSCALE / BAUD
19 pl.text(bit_time/2,0.5,'S',ha='center')
20 mask = 1
21 for c in range(CHAR_BITS):
22     if ord(chars[0]) & mask:
23         char = '1'
24     else:
25         char = '0'
```

```
26     pl.text((c + 1.5) * bit_time,0.5,char,ha='center')
27     mask <<= 1
28 pl.text((c + 2.5) * bit_time,0.5,'P',ha='center')
29 pl.text((c + 3.5) * bit_time,0.5,'E',ha='center')
30 pl.text((c + 4.5) * bit_time,0.5,'E',ha='center')
31 pl.grid()
32
33 # save plot to file
34 fig1.savefig('serial_tx.png',format='png')
35
36
37 if __name__ == '__main__':
38     pl.show()
39
40 #### END
```

This is the plotting result. The x axis units are milliseconds.



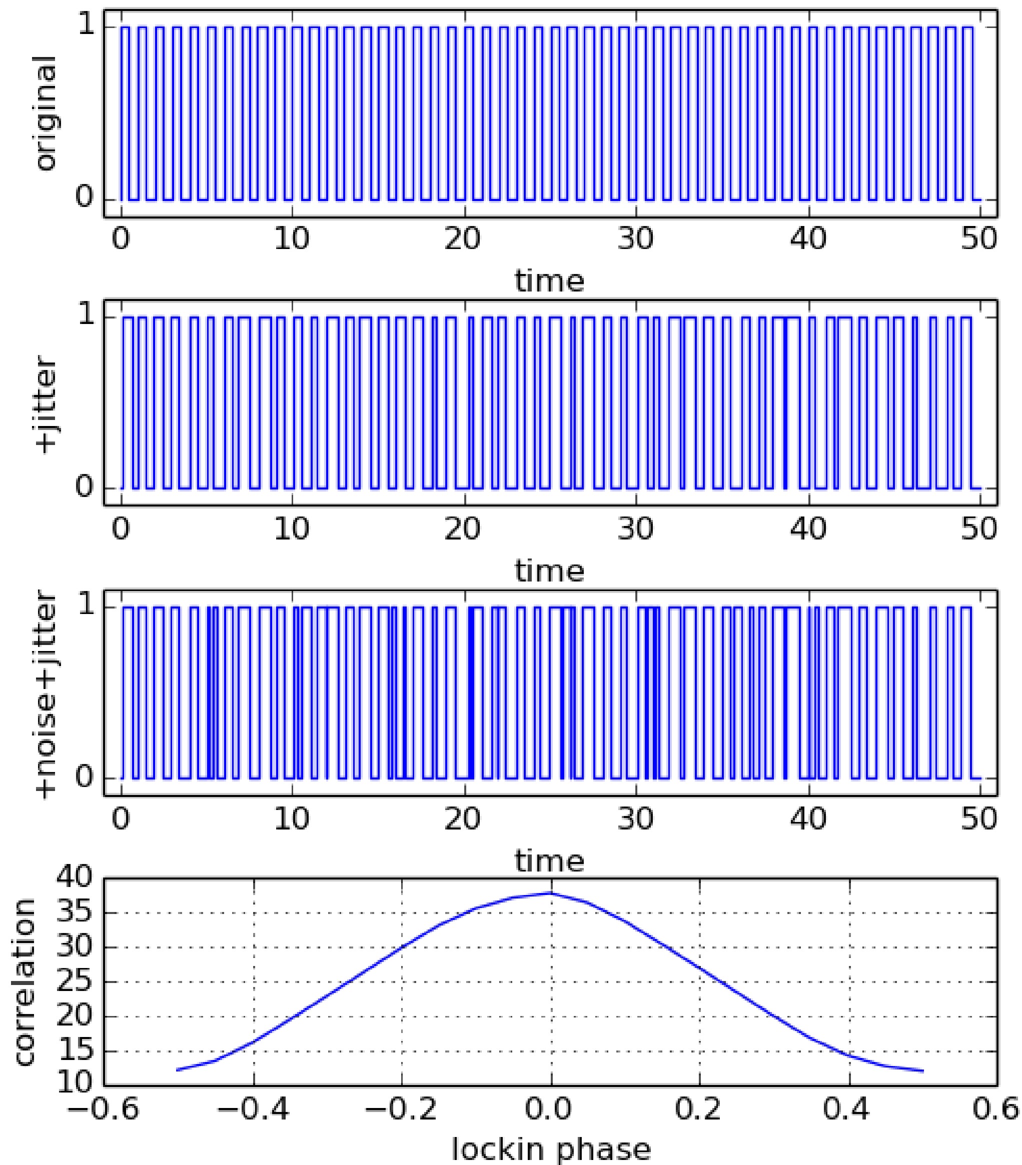
Phase lockin

The following example demonstrate a phase recovery from a disturbed periodic signal whose undisturbed original is known. The original signal is a square wave of 50 cycles @1Hz, 50 % duty cycle. A gaussian jitter is added to the original signal change times and the result is xored with noise pulses to simulate transmission line disturbances.

```
 1 import bitis as bt
 2 import random
 3
 4 import matplotlib.pyplot as pl
 5
 6 # make repeatable random sequences
 7 random.seed(1)
 8
 9 # generate the original signal: square wave, 50 cycles @1Hz, 50% duty cycle.
10 original = bt.square(0.,0.,50.,1.,0.5)
11
12 # add jitter to original signal
13 jittered = original.clone()
14 jittered.jitter(0.1)
15
16 # add noise by xor
17 jittered_noised = jittered ^ bt.noise(0,0,50,period_mean=5,width_mean=0.5)
18
19 # compute correlation between original and disturbed signal
20 corr, shift = original.correlation(jittered_noised,step_size=0.05,
21     skip=49.45,width=1.05)
22
23 # start plotting
24 fig1 = pl.figure(1,figsize=(6,7))
```

```
25 pl.suptitle('BITIS: lockin to a noisy signal.')
26
27 # plot original signal
28 pl.subplot(4,1,1)
29 pl.xlim(-1,51)
30 pl.ylabel('original')
31 pl.xlabel('time')
32 original.plot()
33
34 # plot signal with jitter
35 pl.subplot(4,1,2)
36 pl.xlim(-1,51)
37 pl.ylabel('+jitter')
38 pl.xlabel('time')
39 jittered.plot()
40
41 # plot signal with jitter and noise
42 pl.subplot(4,1,3)
43 pl.xlim(-1,51)
44 pl.ylabel('+noise+jitter')
45 pl.xlabel('time')
46 jittered_noised.plot()
47
48 # plot correlation function
49 pl.subplot(4,1,4)
50 pl.grid()
51 pl.plot(shift,corr)
52 pl.ylabel('correlation')
53 pl.xlabel('lockin phase')
54 pl.subplots_adjust(hspace=0.4)
55
56 # save plot to file
57 fig1.savefig('lockin.png',format='png')
58
59
60 if __name__ == '__main__':
61     pl.show()
62
63 #### END
```

The plot shows the original, the disturbed signal and the correlation among them, correlation that reaches a maximum when the original has that same phase of the disturbed original.



Modulation

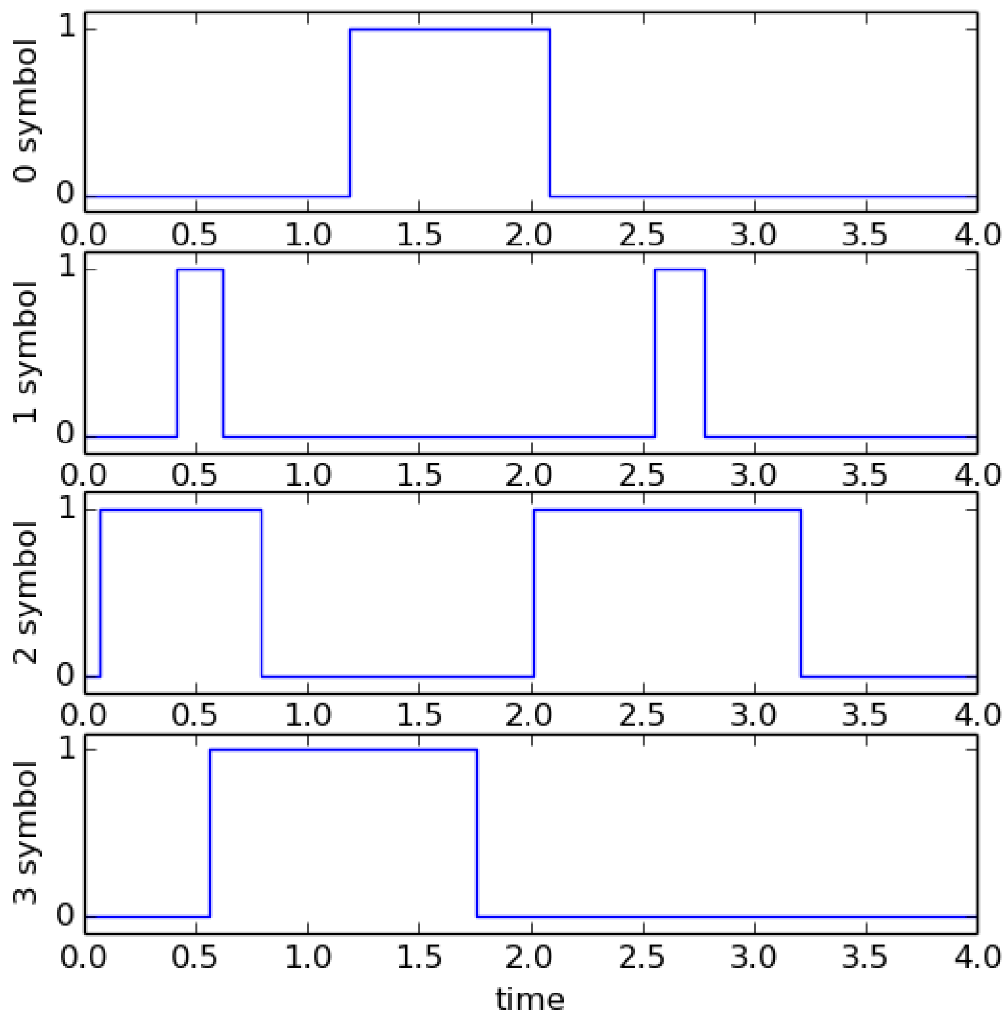
The following example shows the generation of a modulated signal, given a random code and a set of symbols. The modulated signal is obtained concatenating in time the symbol corresponding to a code value. Then the modulated signal is demodulated by maximal correlation symbol estimation. As a byproduct, the signal/symbols correlation matrix is obtained as shown below. The example does not take into account any signal alteration by noise.

```
1 import bitis as bt
2 import random
3 import matplotlib.pyplot as pl
4 from mpl_toolkits.mplot3d import Axes3D
5 import numpy as np
6
7 SYMBOLS_NUM = 4
8 SYMBOL_ELAPSE = 4.
9 SYMBOL_PULSES_MEAN_PERIOD = 1.
10 SYMBOL_PULSES_MEAN_WIDTH = 0.5
11 CODE = [3,0,1,2]
```

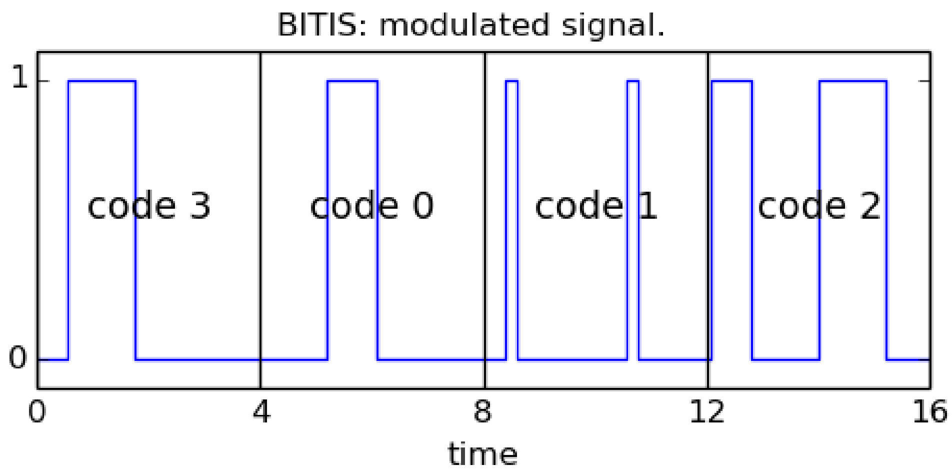
```
12
13 # make repeatable random sequences
14 random.seed(1)
15
16 # generate symbols as random signals
17 symbols = []
18 for i in range(SYMBOLS_NUM):
19     symbol = bt.noise(0.,SYMBOL_ELAPSE,period_mean=
20         SYMBOL_PULSES_MEAN_PERIOD,width_mean=SYMBOL_PULSES_MEAN_WIDTH)
21     # ensure same start and end levels equal to 0
22     while symbol.slevel or symbol.slevel != symbol.end_level():
23         symbol = bt.noise(0.,SYMBOL_ELAPSE,period_mean=
24             SYMBOL_PULSES_MEAN_PERIOD,width_mean=SYMBOL_PULSES_MEAN_WIDTH)
25     symbols.append(symbol)
26
27 # modulate
28 mod = bt.code2mod(CODE,symbols)
29
30 # demodulate
31 decode, corr, corrs = bt.mod2code(mod,symbols)
32
33 # plot symbols
34 fig1 = pl.figure(1,figsize=(6,6))
35 pl.suptitle('BITIS: modulation symbols.')
36 for i in range(len(symbols)):
37     pl.subplot(4,1,i+1)
38     pl.xlim(0,SYMBOL_ELAPSE)
39     pl.ylabel('%d symbol' % i)
40     pl.xlabel('time')
41     symbols[i].plot()
42
43 # plot modulated signal
44 fig2 = pl.figure(2,figsize=(6,2.5))
45 fig2.subplots_adjust(top=0.9,bottom=0.2)
46 pl.suptitle('BITIS: modulated signal.')
47 pl.xlim(0,SYMBOL_ELAPSE*len(CODE))
48 pl.xlabel('time')
49 pl.xticks(np.arange(0,17,4))
50 pl.grid(axis='x',linestyle='-',linewidth=1)
51 mod.plot()
52 for c in range(len(CODE)):
53     pl.text(c*4+2 ,0.5,'code %d' % CODE[c],ha='center',size=14)
54
55 # plot correlation matrix of modulated signal
56 fig3 = pl.figure(3,figsize=(6,6))
57 pl.suptitle('BITIS: correlation matrix of modulated signal.')
58 ax = fig3.gca(projection='3d')
59 x, y = np.mgrid[0:len(CODE),0:SYMBOLS_NUM] - 0.1
60 x = x.flatten()
61 y = y.flatten()
62 z = np.zeros_like(x)
63 pl.xlabel('code time')
64 pl.ylabel('symbol')
65 ax.set_zlabel('correlation')
66 dz = np.array(corrs).flatten()
67 pl.xticks(np.arange(4))
68 pl.yticks(np.arange(4))
69 cz = ['g']*len(z)
70 for i in range(len(cz)):
71     if dz[i] > 0.9:
```

```
72          cz[i] = 'r'
73 ax.bar3d(x,y,z,0.2,0.2,dz,color=cz)
74
75 # save plots to files
76 fig1.savefig('modem1.png',format='png')
77 fig2.savefig('modem2.png',format='png')
78 fig3.savefig('modem3.png',format='png')
79
80
81 if __name__ == '__main__':
82     pl.show()
83
84 #### END
```

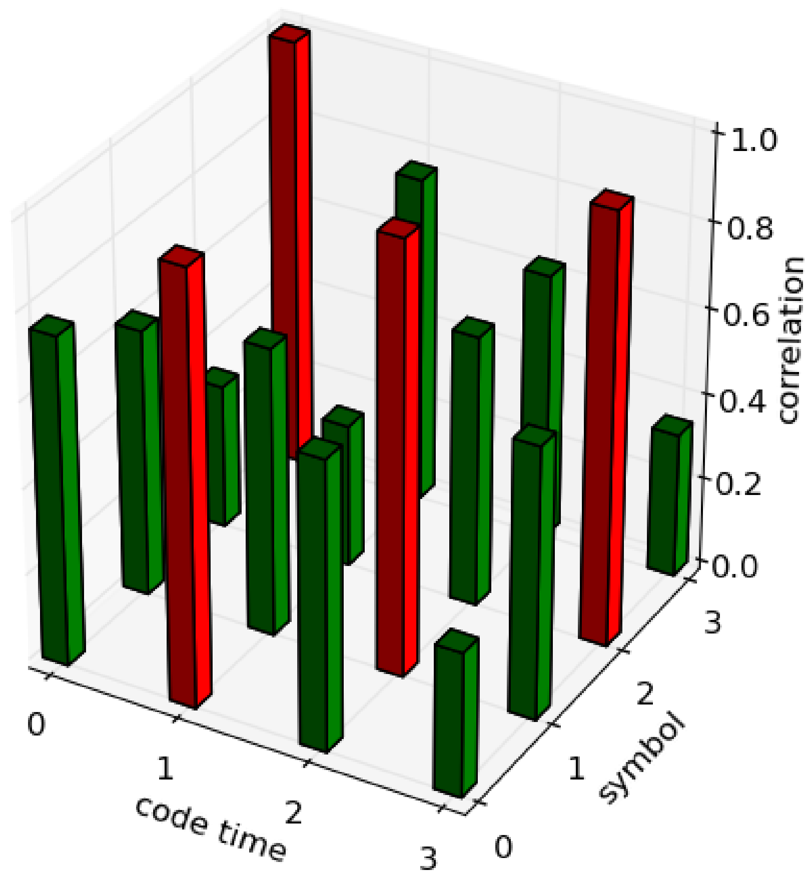
The plot shows the set of four random symbols. Each symbol has an elapse time of 4 seconds.



The plot shows the modulated signal with the boundaries between the symbols. For each symbol time, the code value is shown.



The plot shows the symbol correlation matrix of the modulated signal. The correlation value reaches the maximum where the correlating symbol is equal to the modulating symbol. For each code time, the symbol with maximum correlation (+1) is marked in red.



Module reference

The **Bitis** modules defines one class realizing a BTS signal with a set of methods implementing several unary or binary operators over the signal. There are also some functions for data interface.

Objects and methods

class `bitis.Signal` (*start=None*, *edges=None*, *end=None*, *slevel=0*, *tscale=1.0*)

Implements the concept of "Binary Timed Signal": a memory representation of a binary signal as sequence of the times of signal edges (signal changes). *start* sets the signal start time. *edges* can be used to initialize the signal edges sequence, it must be a list of times (integers or floats). May be empty. The signal level before the first change is specified by *slevel*. Also a time scale factor can be specified by *tscale*, at present not used.

__add__ (*other*)

Concatenate (join) other to self.

__and__ (*other*)

Compute the logic *and* of two given signal objects: *self* and *other*. Return a signal object with the and of the two input signals. Can be used as the bitwise and operator as in the following example (signal a,b,c are instances of the Signal class):

```
signal_c = signal_a & signal_b
```

__eq__ (*other*)

Equality test between two signals. Return *True* if the two signals are equal. Otherwise, return *False*. Can be used as the equality operator as in the following example (signal a,b are instances of the Signal class):

```
if signal_a == signal_b:
    print 'signal a and b are equal'
```

__invert__ (*inplace=False*)

Compute the logic *not* of the given signal object: *self*. If *inplace* is false, return the result as a new signal object. Otherwise, return the result as *self*. Can be used as the bitwise not operator as in the following example (signal a,b are instances of the Signal class):

```
signal_b = ~ signal_a
```

__len__ ()

Return the length of the change times sequence.

__ne__ (*other*)

Inequality test between two signals. Return *True* if the two signals are not equal. Otherwise, return *False*. Can be used as the inequality operator as in the following example (signal a,b are instances of the Signal class):

```
if signal_a != signal_b:
    print 'signal a and b are different'
```

__nonzero__ ()

Return true if the signal is not void, return false otherwise.

__or__ (*other*)

Compute the logic *or* of two given signal objects: *self* and *other*. Return a signal object with the or of the two input signals. Can be used as the bitwise or operator as in the following example (signal a,b,c are instances of the Signal class):

```
signal_c = signal_a | signal-b
```

__xor__ (*other*)

Compute the logic *xor* of two given signal objects: *self* and *signal*. Return a signal object with the xor of the two input signals. Can be used as the bitwise xor operator as in the following example (signal a,b,c are instances of the Signal class):

```
signal_c = signal_a ^ signal_b
```

`append` **(*`other`*)**
Return the *self* signal modified by appending *other* signal to it. If there is a time gap between the signals, fill it. The start time of *other* must be greater or equal to end time of *self*. The end level of *self* must be equal to the start level of *other*. Otherwise, no append is done.

`chop` **(*`period`*, *`origin=None`*, *`max_chops=1000`*)**
Divide the signal into several time contiguous signals with the same elapse time equal to *period*. The dividing times sequence starts at *origin* and has an element every *period* time, except for the last element. It has end time = period * max_chops, if max_chops is reached. Otherwise, it has the end time of the chopped signal. Return a list with the chopped signals. If *origin* is before the signal start, it is moved forward by an integer times of period, until it falls into the signal domain. If *origin* is none, it is set to self start time by default. If *origin* is after the signal end, no chop occours, an empty list is returned. If *self* is void, return an empty chop list.

`clone` **()**
Return a deep copy with the same attributes/values of signal object.

`clone_into` **(*`other`*)**
Return a full copy of signal object into *other*. Each other attribute is assigned a deep copy of the value of the same attribute in signal object. Return *other*.

`correlation` **(*`other`*, *`mask=None`*, *`step_size=1.0`*, *`skip=0`*, *`width=None`*, *`normalize=False`*)**
Return the correlation function of two given signal objects: *self* and *other*.

> **mask**: signal or None, same elapse of *other*. Compute correlation only where *mask* == 1. If None, compute correlation on the whole intersection of *self* and *other*.
>
> **step_size**: positive float, the time pitch of the correlation function.
>
> **skip**: positive float, the time elapse at the start of the correlation function not to be computed.
>
> **width**: is the time elapse where to compute the correlation function. If None, compute the correlation function for each time shift of *self* that has an intersection with *other*, having an elapse time >= *step_size*.
>
> **normalize**: boolean, controls the values of the correlation function. If True, values are normalized in the range -1 +1. If False, values are absolute: the integral of xor between shifted *self* and *other* signals.

Return pattern **(** *corr, shift* **)**

> **corr**: list of floats. The values of the correlation function.
>
> **shift**: list of floats. The time shift applied to *self* to slide it over *other* signal for each value of *corr*.

`elapse` **()**
Ruturn the signal elapse time: end time - start time. If *self* is void, return zero.

`end_level` **()**
Return the logic level at the end of a signal object.

`integral` **(*`level=1`*, *`normalize=False`*)**
Return the integral of a signal object: the elapsed time of all periods in which the signal is at the level specified by *level*. Output can be absolute (*normalize=False*) or can be normalized (*normalize=True*): absolute integral averaged over the whole signal domain. The summation is operated on the signal domain only. If *self* is void, return none.

`jitter` **(*`stddev=0`*)**
Add a gaussian jitter to the change times of *self* signal object with the given standard deviation *stddev* and zero mean. Signal start and end times are unchanged.

`join` **(*`other`*, *`inplace=False`*)**
Join two signals (*self* and *other*) in one signal. End time of *self* must be less or equal to start time of *other*. If there is a time gap between the joining signals, fill it. Return a signal object with the join result. If *inplace* is false, a new signal object is returned. If *inplace* is true, the join result is put into *self* and *self* is returned. Signals with different levels at *self* end and at *other* start cannot be joined (join return a void signal).

`level` **(*`time`*, *`tpos=0`*)**

Return the logic level and the number of edges of a signal object at a given time (*time*) and a given edge position where to start searching (*tpos*). The edge position is the number of signal edges before *time*. When *time* is equal to an edge time that edge is considered before *time*. *time* must be in the signal time domain, otherwise None is returned. *tpos* must point to an edge before *time*.

`newer` **(*`split`*, *`inplace=False`*)**
Split *self* into two signals at time *split* and return the part after split time. If *split* is equal to a signal change time, the change is put into the return signal. If *split* is at or before signal start, return *self*. If *split* is at or after signal end, return the void signal. If *inplace* is false, a new signal object is returned. If *inplace* is true, *self* is changed to the newer part and returned. If *self* is void, the void signal is returned.

`older` **(*`split`*, *`inplace=False`*)**
Split *self* into two signals at time *split* and return the part before split time. If *split* is equal to a signal change time, the change is not put into the return signal. If *split* is at or before signal start, return the void signal. If *split* is at or after signal end, return *self*. If *inplace* is false, a new signal object is returned. If *inplace* is true, *self* is changed to the older part and returned. If *self* is void, the void signal is returned.

`phase` **(*`other`*, *`mask`*, *`resolutions`*, *`period=None`*)**
Find the phase between *self* and *other*. Phase is the time shift that applied to *self* gives the maximum correlation: *self* (t + phase) * *other* (t) is maximum (* means correlation). For faster computation, the phase can be computed by progressive smaller resolutions.

mask: signal or None, same elapse of *other*. Compute correlation only where *mask* == 1. If None, compute correlation on the whole intersection of *self* and *other*.

resolutions: tuple or list of positive float, at least one element, sequence of resolutions from coarser to finest, the time step used in the computation of correlation.

period: None or positive float. If None, phase is computed as absolute time shift. If float, phase is the time shift with respect to nearest integer multiple of *period*, its range is - *period*/2. <= phase < + *period*/2..

Return pattern **(** *phase, corr_phase, corrs, shifts* **)**

phase: float, the computed phase.

corr_phase: float, the correlation function value at phase shift.

corrs: list of lists of floats, positive. For each resolution value specified in *resolutions*, the unnormalized values of the correlation function.

shifts: list of lists of floats. The time shift values corresponding to the correlation function values in *corrs*.

`plot` **(*`*args`*, *`**kargs`*)**
Graphic plot of signal *self* as square wave. Requires Matplotlib. **args* and ***kargs* are passed on to matplotlib functions.

`plotchar` **(*`charnum`*, *`origin=None`*, *`end=None`*, *`max_flat=None`*)**
Semigraphic plot of signal *self* with unicode line drawing characters (U+25xx). Require locale setting.

charnum: integer, the maximum length of the string of the rendering characters.

origin: float, the rendering start time. If None, *start* is set to the signal start time.

end: float, the rendering end time. If None, *end* is set to the signal end time.

max_flat: integer, the maximum number of consecutive horizontal line characters. When reached, no more horizontal chars are added and a lower case 'x' char is put in the middle of this sequence to mark the character drop. If None, compression is disbled.

Return pattern **(** *topchars*, *botchars* **)**

topchars: utf-8 encoded string. The top row of unicode characters rendering the semigraphic plot.

botchars: the same as *topchars*, but for the bottom row.

`reverse` **(*`inplace=False`*)**
Reverse the signal change times sequence: last change becomes the first and viceversa. Time intervals between edges are preserved. If *inplace* is false, return the result as a new signal object. Otherwise, return the result as *self*.

shift (***offset***, ***inplace=False***)
Add *offset* to signal start and end times and to each signal change time. If *inplace* is false, return the result as a new signal object. Otherwise, return the result as *self*.

split (***split***, ***inplace=False***)
Split *self* into two signals at time *split*.
Return pattern **(** *older, newer* **)**

older: signal, the part of *self* from start time to *split* time.

newer: signal, the part of *self* from *split* time to signal end.
If *split* is equal to a signal change time, the change is put into the *newer* part. If *split* is at or before signal start, *older* is the void signal and *newer* is *self*. If *split* is at or after signal end, *older* is *self* and *newer* is the void signal. If *inplace* is false, *newer* is returned as a new signal object. If *inplace* is true, *self* is changed to *newer* and returned as *newer*. If *self* is void, both *older* and *newer* are the void signal.

stream (***other***, ***elapse***, ***buf_step=1.0***)
Append *other* signal to *self* signal. If self signal elapse time becomes greater than *elapse*, delete from the older part of self until its elapse time is less or equal than *elapse*. The elapsed time fo the deleted part is forced to a integer multiple of *buf_step*.

validate **()**
Validate signal attributes. Complete type and value checking of signal object attributes. If a check fails, an exception is raised.

Functions

bitis.**bin2pwm** (*bincode*, *elapse_0*, *elapse_1*, *period*, *active=1*, *origin=0*, *tscale=1.0*)
Convert a binary code into a pulse width modulation signal in BTS format. Return a Signal class object. *bincode* is a tuple or a list of tuples: (*bit_length*, *bits*). *bit_length* is an integer with the number of bits. *bits* is an integer or a long integer with the binary code. First bit is the LSB, last bit is the MSB. *period* is the period of pwm pulses. *elapse_0* is the elapse time of a pulse coding a 0 bit. *elapse_1*, the same for a 1 bit. *active* is the active pulse level. *origin* is the time of the leading edge of the first signal pulse.

bitis.**pwm2bin** (*pwm*, *elapse_0*, *elapse_1*, *period=None*, *active=1*, *origin=0*, *threshold=0.2*)
Convert a pulse width modulation signal in BTS format to binary code. Return a tuple: see *bincode* in **bin2pwm**. *pwm* is the signal to decode. For the other arguments see **bin2pwm**. If *period* is not defined, conversion is done by testing only the active pulse level elapse against a threshold computed as mean of elapse_0 and elapse_1. No check is done on pulse period and decoding consider every pulse. If period is set to the modulation pulse period, conversion is done by synchronous symbols correlation. The signal is chopped with the given *period*, starting from *origin*, start of symbol periods, until signal end. Each signal chop, corresponding to one symbol time, is correlated with both models of 0 and 1 pulses. The better value above *threshold* is taken as result.

bitis.**code2mod** (*code*, *symbols*, *origin=0*, *tscale=1.0*)
Modulate a code sequence into a modulation signal in BTS format. For each number in code, the symbol in symbols with index equal to number is appended to the modulation signal. Return a Signal class object. *code* is list of integer. *symbols* is a list of signal objects, one for each coding symbol. *origin* is the start time of the first coded symbol.

bitis.**mod2code** (*mod*, *symbols*, *mask=None*, *origin=None*, *tscale=1.0*)
Demodulate a modulation signal in BTS format by maximal correlation symbol estimation. Return the demodulated code sequence (list of int), the corresponding normalized correlation, list of float) of all symbols and the time where the demodulation ends. *symbols* is a list of signal objects, one for each coding symbol. The symbols start time is assumed as phase difference with respect to the signal start time. *mask* is a signal objects. Symbol correlation is computed only where mask = 1. *origin* is the start time of the first coded symbol. If not defined, it is set to start time of *mod*. All symbols must have the same elapse time that is the symbol period. The same holds for mask.

bitis.**serial_tx** (*chars*, *times*, *char_bits=8*, *parity='off'*, *stop_bits=2*, *baud=50*, *tscale=1.0*)
Simulate a serial asynchronous transmitting interface. Return a BTS signal with the serial line pulses coding a given list of characters, according to the following serial parameters. The list of *chars* is the input to the serial transmitter. *times* is the list of the start bit rising edge time of each char in *chars*. If times are too fast with respect to the current baud rate, a char fifo behavoiur is activated. *char_bits* is the character size in bits (5,6,7,8). *parity* is

the parity bit even, odd or off (parity absent). *stop_bits* is the number of stop bits (1,2). *baud* is the serial line speed, any positive value is allowed. The serial line is assumed active high.

`bitis.serial_rx (sline, char_bits=8, parity='off', stop_bits=2, baud=50)`

Simulate a serial asynchronous receiving interface. Return a list of the received characters, a list of their start times and a list of their status: 0 = ok, 1 = parity error. *sline* is a BTS signal with the serial line pulses coding the characters to be received. For the keyword arguments see **serial_tx**. The serial line pulses are sampled at the given baud rate like a real asynchronous serial interface.

`bitis.noise (start, origin, end, period_mean=1, period_stddev=1, width_mean=1, width_stddev=1, active='random')`

Return a signal object with random pulses. *start* is the noise signal start. *origin* is the time of the first pulse trailing edge. *end* is the signal end time. Pulses period and width follow a gaussian distribution: *period_mean* and *period_stddev* are the given mean and standard deviation of pulses period, *width_mean* and *width_stddev* are the given mean and standard deviation of the pulse width at 1 level. *active* is the active pulse level, can be 0,1,'random'.

`bitis.square (start, origin, end, period, width, active=1)`

Return a signal object with a square wave with constant period and constant duty cycle. *start* is the start time. *origin* is the time of the first pulse trailing edge. *end* is the signal end time. *period* is the pulse period. *width* is the pulse width at active level.

`bitis.test ()`

Return a signal object with a test signal. The signal has a sequence of primes as edges timing.

BTS format

Definition

The scope of this memo is to describe the BTS, Binary Timed Signal. A format for compact storage of binary signals in computer memory. Binary signals are signal that can have only two logic levels/states, zero or one, true or false.

The *BTS* format is composed by 5 signal elements.

1. The start time, integer or float.
2. The edges times, sequence of integers or floats.
3. The end time, integer or float.
4. The start level, integer or boolean.
5. The time scale, integer or float.

Start time

The start time of the signal. Before this time, the signal is not defined. When it is none, the signal is considered void.

Edges times

This sequence stores all the times where the signal changes its level from 0 to 1 or viceversa. The edge times sequence may be empty: in this case the signal is constant. The sequence must be sorted in ascending order. All elements must have different times. The first element must be greater or equal to the start time. The last element must be lower or equal to the end time.

End time

The end time of the signal. After this time, the signal is not defined. The end time must be greater than the start time. May be none when start is none.

Start level

If the edges times sequence has 1 or more items, the start level value specifies the signal level from the signal start time to the first edge time. If the edges times sequence is empty, the signal has a constant level that is equal to the start level value.

Time scale

An arbitrary unit of time can be chosen to express the values of times. The time scale value is the ratio: 1 second / arbitrary time unit.

Python implementation

BITIS implements the *BTS* format with the *Signal* class. Each BTS signal is an instance of this class. The five elements of the BTS format are the five attributes (*start, edges, end, slevel, tscale*) of the *Signal* class. The sequence *edges* is realized as list of integers or floats.

Pre version 0.9.0 format

Pre 0.9.0 definition

The *BTS* format is composed by 3 elements.

1. The change times.
2. The start level.
3. The time scale.

Change times

This sequence stores all the times where the signal changes its level from 0 to 1 or viceversa. The first and the last sequence items have a different meaning: they are respectively the start time and the end time of the signal. The signal start and end are the boundaries of the signal domain. Outside this interval, the signal is to be itended as not defined. The change times sequence may be empty: in this case the signal must be threated as empty or null. The sequence may have 2 items: in this case the signal has a constant level along all its domain and there are no level changes. The sequence may have 3 or more items: in this case the signal has 1 or more level changes. A sequence with only one item is not allowed. The sequence must be sorted in ascending order.

Start level

If the change times sequence has 3 or more items, the start level value specifies the signal level from the signal start time to the first change time. If the change times sequence has 2 items, the signal has a constant level that is equal to the start level value.

Time scale

An arbitrary unit of time can be chosen to express the values of change times. The time scale value is the ratio: 1 second / arbitrary time unit.

Pre 0.9.0 python implementation

BITIS implements the *BTS* format with the *Signal* class. Each BTS signal is an instance of this class. The three elements of the BTS format are the three attributes (*times, slevel, tscale*) of the *Signal* class. The sequence *times* is realized as list of integers or floats.

Changes

Release 0.12.3 (released 9-Dec-2014)

Changes

- Methods older and newer: rewrite of boundary conditions processing.

Bugs fixed

- Method split: wrong return signal when split at or after self end.
- Method older: wrong end time of return signal.

Documentation

- Methods split, newer and older: new documentation and layout.

Release 0.12.2 (released 6-Dec-2014)

Bugs fixed

- Method correlation: fix wrong computation of correlation function when there is a mask.
- Method phase: fix initial search width not set to the whole shift range.

Documentation

- Method correlation: new documentation layout.

Release 0.12.1 (released 3-Dec-2014)

New features

- Method phase: add correlation value at phase shift.

Bugs fixed

- Method phase: fix incomplete refactoring of other identifier.

Release 0.12.0 (released 1-Dec-2014)

New features

- Method shift: now skips computations for zero offset.
- New method phase: computation of phase among two signals.

Changes

- Method split: now manage a split time outside signal domain returning the proper void signal.
- Method correlation: dropped step_left and step_right arguments, substituted skip and width.
- Method plotchar: dropped period argument.
- Method mod2code: now symbol start time is the phase with respect to the sig start time.
- Methods noise and square: now require an origin argument.

Internals

- Method correlation: refactoring for new arguments skip and width.
- Method correlation: augumented test.
- New method phase: add test.

Documentation

- Started better layout for function/methods arguments and return patterns.

Release 0.11.2 (released 8-Oct-2014)

Bugs fixed

- Method plotchar: missing last non flat char after flat chars.

Release 0.11.1 (released 6-Oct-2014)

Changes

- Method plotchar: now argument max_flat deault is None, was 100 .

Bugs fixed

- Method level: now for time < start return (None,0) .
- Method plotchar: last flat lost when signal end < plot end.
- Method stream: now the newest part is self, was a new allocated signal.

Internals

- Method plotchar refactored.
- New test for methods level and plotchar.

Release 0.11.0 (released 1-Oct-2014)

Features added

- Method plotchar: semigraphic signal plot with line drawing characters.

Changes

- Method level: now return None, len(signal) when time is beyond signal end.
- Method elapse: return zero when signal is void, before was none.

Bugs fixed

- Method serial_tx: returned void signal when chars had len == 1.
- Method serial_rx: dead lock when last sample time was before end time and after last ege time.

Documentation

- Add example plot.

Release 0.10.0 (released 26-Sep-2014)

Features added

- Method validate: a consistency checker for signal attributes.
- Method code2mod: code to symbols signal modulator.
- Method mod2code: demodulator by maximum correlation symbol estimation.

- Example "modulation".
- New method end_level: return the ending level of a signal.
- New method older: return the older part of a signal with respect to a given time.
- New method newer: return the newer part of a signal with respect to a given time.

Changes

- Method test changed to function.
- Signal instancing now validate signal attributes.
- Now, instancing of Signal() generates a void signal.
- Changed return of method split when split time falls outside signal domain.
- Now method serial_tx generate a serial signal with start=origin.

Bugs fixed

- Method chop: wrong chop when split falls on signal end.
- Method __add__: added inplace=false to join call.
- Method level: wrong level returned.
- Method join: changed start and end calls with corresponding attributes.
- Method serial_rx: corrected wrong char start detection and level tests.
- Method noise: missing return argument, the noise signal itself.
- Method append: now update correctly the end time of the result.

Documentation

- Added the rules of BTS format.

Internals

- Rewrite of void signal handling through all methods and functions.
- New test for methods code2mod and mod2code.
- Refactored method split with method level.
- Added random inplace to spit/join test.
- New test for methods older and newer.
- Method append: now implemented with a call to split.

Release 0.9.0 (released 10-Sep-2014)

Features added

- New method level: return the signal level and edge position at a given time.
- Methods shift, reverse, __invert__ now can work inplace: result into self signal.
- New method __nonzero__: return true if the signal is not empty.

Changes

- All methods and objects changed to work with the new BTS format (v2).

- Removed methods: start, end.

Bugs fixed

- Fix method reverse: now works when signal start != 0.
- Fix method split when split time falls on signal start or end and after last edge.
- Fix method chop.
- Fix methods __eq__ and __ne__: now work when operands are None.
- Fix function serial_rx. Now work with constant (no edges) signals. Eliminated spurious status generation.

Internals

- Method _intersect now returns as last edge position the position plus one.
- Added tests for inplace/noinplace testing.

Release 0.8.0 (released 26-Aug-2014)

Features added

- New method chop: divide a signal in a sequence of contiguous signal of given period.
- Method correlation now has a mask argument: if mask signal is not none, the correlation is computed only where mask=1.
- Method join now has an inplace arguments. When true, no new signal is generated for the join result. Self signal is used instead.
- Method pwm2bin now can convert by synchronouos symbols correlation.
- Method split now has an inplace argument. When true, no new signal is generated for the newer signal part. Self signal is used instead.
- Method split, when splitting on a signal change time, now assigns the change to the start of the newer signal part.

Changes

- Methods start, end, elapsed now return None when the signal time changes sequence is empty.
- Method bin2pwm now signal start=origin and signal end is not extended.

Bugs fixed

- Fix method correlation stepping limits for defaults.
- Fix method split splitting on a change time: now correct end of older part and correct start of newer part are generated. start of newer were generated.
- Fix method serial_rx bit time computation: use floats.

Internals

- Added test for method chop.
- Added test for new the convertion mode (sync symb corr) of method pwm2bin.

Release 0.7.1 (released 3-Feb-2014)

Bugs fixed

- Fix inequality test: missing __ne__ method.

Internals

- Optimized "and" and "or" operator for constant signals.

Release 0.7.0 (released 27-Jan-2014)

Features added

- Add buf_step to method stream.
- Add return self to in place working method clone_into.

Incompatible changes

- Change step_start, step_num with step_left, step_right in method correlation.
- Change correlation unittest from a graphic one to procedural only.

Release 0.6.0 (released 16-Dec-2013)

Features added

- Add method clone_into.
- Add method concatenate: add operator.
- Add method stream.
- Add method elapse returning the signal elapse time.
- Add example to demonstrate phase recovery from a noisy signal (lockin).
- Add examples, module reference, bts format, change log to doc pages.
- Add unittest for stream.

Incompatible changes

- Change start level with active argument in noise method.

Bugs fixed

- Fix method append: make it return the signal with the append result.
- Fix shift in correlation method.
- Fix time shift computation in correlaton method: was delayed by 1 step size.

Internals

- Change method append: check arguments with assert.
- Refactor method split.

Release 0.5.0 (released 9-Dec-2013)

Features added

- Embed y limits setting into plot method.
- Add method square for signal generation of a periodc square wave.
- Add a more fine control in correlation function computation.
- Add signal append method.
- Add method start, return signal start time.
- Add method end, return signal end time.
- Add method len, return signal change times sequence length.

Incompatible changes

- Change start times computation in bin2pwn, serial_tx to minimize time elapse from start to first change.

Bugs fixed

- Fix 0.4.0 release changelog: missing changes.

Internals

- Change noise from method to function.
- Change examples for changed noise method.

Release 0.4.0 (released 2-Dec-2013)

Features added

- Add signal split method.
- Add two signals join method.
- Add unittest for split and join.
- Add float times capability to BTS signals.

Incompatible changes

- Uniformate pwm2bin arguments to bin2pwm methods.
- Add tscale=1. argument in bin2pwm.
- Change to tscale=1. argument in serial_tx.

Bugs fixed

- Fix slevel setup, signal start and end in bin2pwm.

Internals

- Rewrite jitter method.

Release 0.3.0 (released 11-Nov-2013)

Features added

- Add async serial transmitter (bits.serial_tx method) from chars to BTS serial line signal.
- Add async serial receiver (bitis.serial_rx method) from BTS serial line to chars.
- Add async serial transmitter example: serial_tx.py.
- Add unittest for async serial tx and rx.
- Modified plot method: only 0,1 ticks on y axis.

Release 0.2.0 (released 4-Nov-2013)

Features added

- Add PWM coder and decoder between a BTS signal (PWM) and a binary code.
- New correlation example.

Release 0.1.0 (released 29-Oct-2013)

- First release.

Index

_

__add__() (bitis.Signal method)
__and__() (bitis.Signal method)
__eq__() (bitis.Signal method)
__invert__() (bitis.Signal method)
__len__() (bitis.Signal method)
__ne__() (bitis.Signal method)
__nonzero__() (bitis.Signal method)
__or__() (bitis.Signal method)
__xor__() (bitis.Signal method)

A

append() (bitis.Signal method)

B

bin2pwm() (in module bitis)

C

chop() (bitis.Signal method)
clone() (bitis.Signal method)
clone_into() (bitis.Signal method)
code2mod() (in module bitis)
correlation() (bitis.Signal method)

E

elapse() (bitis.Signal method)
end_level() (bitis.Signal method)

I

integral() (bitis.Signal method)

J

jitter() (bitis.Signal method)
join() (bitis.Signal method)

L

level() (bitis.Signal method)

M

mod2code() (in module bitis)

N

newer() (bitis.Signal method)
noise() (in module bitis)

O

older() (bitis.Signal method)

P

phase() (bitis.Signal method)
plot() (bitis.Signal method)
plotchar() (bitis.Signal method)
pwm2bin() (in module bitis)

R

reverse() (bitis.Signal method)

S

serial_rx() (in module bitis)
serial_tx() (in module bitis)
shift() (bitis.Signal method)
Signal (class in bitis)
split() (bitis.Signal method)
square() (in module bitis)
stream() (bitis.Signal method)

T

test() (in module bitis)

V

validate() (bitis.Signal method)