# Advanced Guide to Universal ALICE 1.0:

## Objective:

This document serves as a supplemental guide to the more advanced functionality of Universal ALICE 1.0 software interface written for use various supported hardware. It is assumed that the reader has more than a passing familiarity with the Python programing language.

## Background:

Universal ALICE 1.0 is written in Python and includes the Numpy extension for numerical analysis. (<https://en.wikipedia.org/wiki/NumPy>) Numpy is the fundamental Python package for scientific computing. It contains among other things, a powerful array object along with a large library of high-level mathematical functions to operate on these arrays. Such as histograms, the Fourier transform, polynomial curve fitting and random number capabilities. ALICE provides the user access to these functions through a number of the interfaces such as mathematical operations on captured waveforms, including digital filters, arbitrary waveform generation, FFTs and windowing functions.

Customizing ALICE:

There are a number of variables that the user can use to customize the appearance of the user interface. These variables are located near the top of the Python program file. Alternatively, especially for users of the Windows executable version of ALICE, a file named alice\_init.ini can be created. It should be placed in the same directory with the alice-desktop-1.1.py source file or the directory where the program is started. The alice\_init.ini file is read, if found, when ALICE starts and before any of the windows are created. If no init file is found the internal default settings are used.

There are pairs of variables for each display window that set the size of the graphics drawing area in screen pixels. The default values are sized to optimally fill a screen with 1024X600 resolution. The menu buttons surrounding the graphics area need this much space to be properly displayed on most screens so using sizes smaller than the default may result in mangled menus.

GRW = 720 # Width of the Time grid

GRH = 390 # Height of the Time grid

GRWN = 720 # Width of the spectrum grid 720 default

GRHN = 390 # Height of the spectrum grid 390 default

GRWBP = 720 # Width of the Bode Plot grid 720 default

GRHBP = 390 # Height of the Bode Plot grid 390 default

GRWXY = 420 # Width of the XY grid 420 default

GRHXY = 390 # Height of the XY grid 390 default

GRWIA = 400 # Width of the Impedance grid 400 default

GRHIA = 400 # Height of the Impedance grid 400 default

The colors that are used to draw the various parts of the screen can be modified.

Color = "#rrggbb" rr=red gg=green bb=blue, Hexadecimal values 00 - ff

COLORframes = "#000080" # 50% blue

COLORcanvas = "#000000" # 100% black used for background color

COLORgrid = "#808080" # 50% Gray used for grid lines

COLORzeroline = "#0000ff" # 100% blue used for vertical and horizontal center grid lines

COLORtrace1 = "#00ff00" # 100% green CH A voltage trace

COLORtrace2 = "#ff8000" # 100% orange CH B voltage trace

COLORtrace3 = "#00ffff" # 100% cyan CH A current trace

COLORtrace4 = "#ffff00" # 100% yellow CH B current trace

COLORtrace5 = "#ff00ff" # 100% magenta Math trace

COLORtrace6 = "#ff0000" # 100% red

COLORtrace7 = "#8080ff" # 100% purple

COLORtraceR1 = "#008000" # 50% green CH A voltage snapshot trace

COLORtraceR2 = "#804000" # 50% orange CH B voltage snapshot trace

COLORtraceR3 = "#008080" # 50% cyan CH A current snapshot trace

COLORtraceR4 = "#808000" # 50% yellow CH B current snapshot trace

COLORtraceR5 = "#800080" # 50% magenta Math snapshot trace

COLORtraceR6 = "#800000" # 50% red

COLORtraceR7 = "#4040a0" # 70% purple

COLORtext = "#ffffff" # 100% white used for Text display

COLORtrigger = "#ff0000" # 100% red used for trigger point

COLORsignalband = "#ff0000" # 100% red

# Set variable for width of grid lines in pixels

GridWidth = 1

# default math equations, These initialize the entry spaces the first time each matching dialog pop-up window opens.

MathString = “(VBuffA + VBuffB - CHAOffset)“

MathXString = ”(VBuffA - CHAOffset)“

MathYString = ”(VBuffB - CHBOffset)“

UserAString = “MaxV1-VATop”

UserALabel = “OverShoot”

UserBString = “MinV2-VBBase”

UserBLabel = “UnderShoot”

MathAxis = “V-A” # can be one of the following “V-A”, “V-B”, V-C”, “V-D”

MathXAxis = “V-A”

MathYAxis = “V-B”

AWGAMathString = ”(VBuffA + VBuffB)/2”

AWGBMathString = “(VBuffA + VBuffB)/2”

FFTUserWindowString = “numpy.kaiser(SMPfft, 14) \* 3”

DigFilterAString = “numpy.sinc(numpy.linspace(-1, 1, 91))“

DigFilterBString = “numpy.sinc(numpy.linspace(-1, 1, 91))”

Inside the alice\_init.ini any of these variables can be set using the example format shown here:

GRW = 720

GRH = 390

GRWN = 720

GRHN = 390

GRWXY = 420

GRHXY = 390

GRWIA = 400

GRHIA = 400

The optional software interfaces can be enabled or disabled by setting the following variables to either 1 or 0 in the alice\_init.ini file.

EnableDigitalFilter

EnableCommandInterface

## Variables and Arrays:

ALICE uses a large number of variables and arrays to hold both the captured data as well as the results of calculations performed on the data. Below is a list of the variable and array names with explanations of their use and how they are calculated.

Endsample = Last sample in Buffer

hldn = number of samples from start of buffer to ignore based on Hold Off time setting.

Waveform calculated Vertical measurement constants:

Channel A Average voltage, DCV1 = numpy.mean(VBuffA[hldn:Endsample])

Channel A Minimum voltage, MinV1 = numpy.amin(VBuffA[hldn:Endsample])

Channel A Maximum voltage, MaxV1 = numpy.amax(VBuffA[hldn:Endsample])

Channel A Top voltage, VATop is the voltage of the most positive peak in the histogram

Channel A Base voltage, VABase is the voltage of the least positive peak in the histogram

Channel A RMS voltage, SV1 = numpy.sqrt(numpy.mean(numpy.square(VBuffA[hldn:Endsample])))

Channel B Average voltage, DCV2 = numpy.mean(VBuffB[hldn:Endsample])

Channel B Minimum voltage, MinV2 = numpy.amin(VBuffB[hldn:Endsample])

Channel B Maximum voltage, MaxV2 = numpy.amax(VBuffB[hldn:Endsample])

Channel B Top voltage, VBTop is the voltage of the most positive peak in the histogram

Channel B Base voltage, VBBase is the voltage of the least positive peak in the histogram

Channel B RMS voltage, SV2 = numpy.sqrt(numpy.mean(numpy.square(VBuffB[hldn:Endsample])))

Waveform calculated Horizontal measurement constants:

CHAHW is the channel A High Pulse Width

CHALW is the channel A Low Pulse Width

CHADCy is the channel A Duty Cycle

CHAperiod is the channel A Period

CHAfreq is the channel A Frequency

CHABphase is the channel A to channel B relative phase angle

CHBHW is the channel B High Pulse Width

CHBLW is the channel B Low Pulse Width

CHBDCy is the channel B Duty Cycle

CHBperiod is the channel B Period

CHBfreq is the channel B Frequency

Captured Data Waveform Buffers:

VBuffA is the Channel A voltage sample array ( in volts )

VBuffB is the Channel B voltage sample array ( in volts )

VmemoryA is the Channel A voltage memory array used for Trace Averaging

VmemoryB is the Channel B voltage memory array used for Trace Averaging

HBuffA contains the histogram of the channel A voltage waveform

HBuffB contains the histogram of the channel B voltage waveform

t is the time index ( 10 uSec per point )

SAMPLErate is the sampling rate, 100000 samples per Sec, or 10 uSec per sample

Vertical Position variables:

CHAOffset is the value in the channel A voltage position entry window

CHBOffset is the value in the channel B voltage position entry window

CHAIOffset is the value in the channel A current position entry window

CHBIOffset is the value in the channel B current position entry window

AWG waveform arrays:

AWGAwaveform is the Channel A AWG waveform memory array (used for non-built in waveforms)

AWGBwaveform is the Channel B AWG waveform memory array (used for non-built in waveforms)

The following example Python syntax allows setting the start and stop points to be used in an array:

AWGAwaveform[ start : stop ] where start and stop are integers.

Digital Filter coefficients:

DFiltACoef and DFiltBCoef

Use Example: VBuffA = numpy.convolve(VBuffA, DFiltACoef)

Frequency domain buffers:

FFTresultA contains the Channel A voltage magnitude FFT frequency bin results. To get the results in dB the following formula is used:

dbA = (10 \* math.log10(float(FFTresultA[n])) + 17) # gives amplitude in dBVolts where 0 dB = 1 Vrms

FFTresultB, Same for Channel B

FFTmemoryA is the Channel A FFT memory array used for Trace Averaging and Peak Hold modes

FFTmemoryB, Same for Channel B

SMPfft is the number of samples used when the FFT is calculated. It will always be a power of 2. And it will be the length of the FFT window function array.

Bode Plotter arrays:

FSweepAdB contains the Channel A voltage magnitude Bode plot frequency sweep results. To get the results in dB the following formula is used:

dbA = (10 \* math.log10(float(FSweepAdB[n])) + 17)

FSweepBdB, Same for Channel B

FSweepAPh

FSweepBPh, Same for Channel B

## Fourier series of cosines for a square wave:

One of the AWG waveforms that can be constructed is the Fourier series of cosines for a square wave.

The routine starts by first making the cosine wave at the fundamental frequency:

AWGAwaveform = numpy.cos(numpy.linspace(0, 2\*numpy.pi, SAMPLErate/AWGAFreqvalue))

It then loops over k ( only odd numbers ), the number of requested terms, calculating the harmonic and adding it to the waveform:

Harmonic = (math.sin(k\*numpy.pi/2)/k)\*(numpy.cos(numpy.linspace(0, k\*2\*numpy.pi, SAMPLErate/AWGAFreqvalue)))

AWGAwaveform = AWGAwaveform + Harmonic

After all the harmonic terms have been added the waveform is scaled and offset based on the entered Min and Max values.

## Fourier series of sines for a saw tooth wave:

Similar to the above built in AWG waveform Shape, ALICE contains a function to generate the Fourier series of sines for a saw tooth wave. The function can be run using the Math wave Shape by entering the function and passing the appropriate values or executed in the Command line tool or in a script.

The routine starts by first making the cosine wave at the fundamental frequency based on the Length value (number of samples in waveform):

Sn = Const \* numpy.sin(numpy.pi\*x)

It then loops over NumTerms, the number of requested terms, calculating the harmonic and adding it to the waveform. The final resulting waveform is then scaled by the Ampl value. Setting Ampl to a negative number will invert the waveform (direction of the saw tooth ramp).

Code:

def FourierSawTooth(Length, NumTerms, Ampl):

L = 1 # Length of the interval

x = numpy.linspace(0, 2, Length); # Create Length points on the interval [-3L, 3L]

Const = -2/numpy.pi # Constant factor in the expression for B\_n

Sn = Const \* numpy.sin(numpy.pi\*x) # Initialize vector sum series to zero

n = 2

while n <= NumTerms:

Const = -Const # Efficient way to implement alternating sign

Bn = Const/n # Coefficients inversely proportional to n

Fn = Bn \* numpy.sin(n\*numpy.pi\*x) # Calculate Fourier term

Sn = Sn + Fn # Add the term to Fourier sum

n = n + 1

Sn = Sn \* Ampl # Scale waveform by Ampl

return Sn

## Saving Array Data to Files:

ALICE supplies two ways to save the contents of any array to a file using the Command Line interface. These are in addition to the ( behind the scenes ) ways provided under the File drop down and other menus. The first is built into numpy. For example, to save the VBuffA array ( channel A voltage waveform buffer ) to a .csv file you would type in the Command Line interface:

numpy.savetxt(“my\_data.csv”, VBuffA, delimiter=",", fmt='%2.4f')

Where “my\_data.csv” is the name of the destination file, VBuffA is of course the data array to save, delimiter="," tells the function to use a , to separate the columns ( there won’t be multiple columns since most ALICE arrays are one dimensional ) and fmt='%2.4f' sets the format to 4 decimal places.

For saving more than one array to same file Use numpy.transpose():

numpy.savetxt(“my\_data.csv”, numpy.transpose([VBuffA,VBuffB]), delimiter=",", fmt='%2.4f')

The second is a wrapper function around the Python wave package. For example, to save the VBuffA array (channel A voltage waveform buffer) to a mono .wav file ( at 100 KSPS ) you would type in the Command Line interface:

Write\_WAV(VBuffA, 2, “my\_data.wav”)

Where “my\_data.wav” is the name of the destination file, VBuffA is of course the data array to save, and the 2 tells the program to save two copies of the array data to the file for example. This is handy to make longer versions of the relatively short buffer lengths, used in ALICE, that can be listened to by playing back the .wav file.

## Special Built-in Functions:

ALICE contains a number of special functions that can be used to measure, generate and manipulate waveform array data samples. These functions can be used to generate AWG waveforms using the Math wave Shape by entering the function and passing the appropriate values or executed in the Command line tool or in a script.

### SinePower generator

The SinePower function generates a single cycle of the the wave shape with Length number of samples. The Phase variable can be an angle from 0 to 360 degrees. The wave will always be centered on +2.5V and amplitude will be +/- 1.0 V around that when the Ampl variable is set to 1.0. The “Power” variable is an exponent parameter for the SinePower waveform. If the power value is greater than 0, the sine function outputs: sin(x)^{100/{100 - power}}. If the power value is less than 0, the function becomes sin(x)^{{100+power}/100}. The wave will be a pure sine wave with Symmetry set to 0.0 and be a pure square wave with Symmetry set to 100. With Symmetry set to -99.9 the shape will be a pair of narrow positive and negative pulses.

Code:

def SinePower(Length, Power, Phase, Ampl):

# Generate a Sine Power Pulse waveform of length samples with Symmetry, Phase and Ampl

OutArray = []

t = 1.0E-5 # 10 uSec

frequency = 1.0/(t\*Length) # Freq of one cycle

exponent\_setting = numpy.clip(Power, -99.999999999, 100.000) / 100.0

if exponent\_setting >= 0:

exponent = (1.0 - exponent\_setting)

else:

exponent = 1.0 / (1.0 + exponent\_setting)

#

Len = 0

while Len < Length:

x = t \* Len \* frequency + Phase / 360.0

plain\_old\_sine = numpy.sin(x \* 2 \* numpy.pi)

# In the SinePower wave function, the 'Power' value is used

# to indicate and exponent between 1.0 and 0.0.

y = numpy.copysign(numpy.abs(plain\_old\_sine) \*\* exponent, plain\_old\_sine)

OutArray.append(Ampl \* y)

Len = Len + 1

#

OutArray = numpy.array(OutArray) + 2.5 # Center wavefrom on 2.5 V

return(OutArray)

#

### Schroeder Multi-sine generator

Multi-sine signals are often used in frequency response measurements of a network or system.

In 1970, Schroeder published a method for reducing the crest factor of multi-sine signals with flat amplitude spectra and equally spaced frequency components by choosing the phases φk such that φk=−k(k−1)π/k. The typical crest factor of a Schroeder multi-sine with flat amplitude spectra and uniformly spaced frequency components is approximately 1.6.

The function returns an array of sample points of length Length, scaled by Ampl. The lowest, first tone has a period of Length samples. NrTones integer multiples of the first tomes are generated.

def SchroederPhase(Length, NrTones, Ampl):

# Generate a Schroeder Phase (Chirp) of Length samples and having NrTones

OutArray = []

OutArray = Ampl\*numpy.cos(numpy.linspace(0, 2\*numpy.pi, Length)) # the fundamental

k = 2

while k <= NrTones:

# Add all harmonics up to NrTones

Harmonic = Ampl\*numpy.cos(numpy.linspace(0, k\*2\*numpy.pi, Length)+(numpy.pi\*k\*k/NrTones))

OutArray = OutArray + Harmonic

k = k + 1

OutArray = OutArray + 2.5 # Center wavefrom on 2.5 V

return(OutArray)

#

## Generating Test Noise waveforms

### Time-series From Half-spectrum:

It is often useful to be able to produce not only flat, but arbitrary noise spectrum profiles - flat “bands” of noise, “pink noise”, “noise mountains” emulating peaking in some amplifiers. The Generate Time-series From Half-spectrum code block starts with a desired noise spectral density (which can be generated manually or from simulation), the sample rate of the time series, and produces a time series of voltage values that can be then played back through the AWG.

Code:

#

# Generate Time-series From Half-spectrum code block

# takes: a desired noise spectral density array (freq)

# the sample rate of the time series (fs),

# returns a time series of voltage samples that can be sent to the AWG

#

# DC in first element.

# Output length is 2x input length

def time\_points\_from\_freq(freq, fs=1, density=False):

N=len(freq)

rnd\_ph\_pos = (numpy.ones(N-1, dtype=numpy.complex)\*

numpy.exp(1j\*numpy.random.uniform

(0.0,2.0\*numpy.pi, N-1)))

rnd\_ph\_neg = numpy.flip(numpy.conjugate(rnd\_ph\_pos))

rnd\_ph\_full = numpy.concatenate(([1],rnd\_ph\_pos,[1], rnd\_ph\_neg))

r\_s\_full = numpy.concatenate((freq, numpy.roll(numpy.flip(freq), 1)))

r\_spectrum\_rnd\_ph = r\_s\_full \* rnd\_ph\_full

r\_time\_full = numpy.fft.ifft(r\_spectrum\_rnd\_ph)

# print("RMS imaginary component: ",

# np.std(np.imag(r\_time\_full)),

# " Should be close to nothing")

if (density == True):

#Note that this N is "predivided" by 2

r\_time\_full \*= N\*numpy.sqrt(fs/(N))

return(numpy.real(r\_time\_full))

Some examples that use time\_points\_from\_freq

#

def TimeSeriesNoise(n, Fsample, mag, b=4):

# Build Noise Time-series

# n = number of Freq Bins

# b = number of noise bands

# Fsample is Sample Rate

# generates four "bands" of mag V/rootHz noise

mag = mag \* 0.707106 # scale by 1/sqrt 2 for RMS

width = int(n/(4 \* b))

i = 1

aband = numpy.ones(width)

zband = numpy.zeros(width)

bands = numpy.concatenate((aband, zband))

while i < b:

bands = numpy.concatenate((bands, aband, zband))

i = i + 1

bands = bands\*mag

bands[0] = 0.0 # Set DC bin content to zero

return time\_points\_from\_freq(bands, fs=Fsample, density=True)

#

#

# Generate Time samples for single frequency Bin

# Uses IFFT

#

def TimeSeriesSingleTone(n, BinNum, Fsample, mag):

# Build Single tone Time-series

# n = number of Freq Bins

# BinNum = FFT Bin number

# Fsample is Sample Rate

# mag is tone amplitude

bands = numpy.zeros(n)

bands[BinNum] = 1

bands = bands \* (mag/2.0)

return time\_points\_from\_freq(bands, fs=Fsample, density=True)

### Colored Noise Generators

def PinkNoise(N, mag):

# Pink noise.

# Pink noise has equal power in bands that are proportionally wide.

# Power spectral density decreases with 3 dB per octave.

# N Length of sample array, mag magnitude scaling factor

x = numpy.random.normal(0.0, 1, N).astype(numpy.float32) # white Noise

X = numpy.fft.rfft(x) / N

S = numpy.sqrt(numpy.arange(X.size)+1.0) # +1 to avoid divide by zero

y = numpy.fft.irfft(X/S).real[:N] # extremely tiny value 1e-9 without normalization

z = numpy.ndarray = mag

y = y \* numpy.sqrt((numpy.abs(z)\*\*2).mean() / (numpy.abs(y)\*\*2).mean())

return y

def BlueNoise(N, mag):

# Blue noise.

# Power increases with 6 dB per octave.

# Power spectral density increases with 3 dB per octave.

# N Length of sample array, mag magnitude scaling factor

x = numpy.random.normal(0.0, 1, N).astype(numpy.float32) # white Noise

X = numpy.fft.rfft(x) / N

S = numpy.sqrt(numpy.arange(X.size)) # Filter

y = numpy.fft.irfft(X\*S).real[:N]

z = numpy.ndarray = mag

y = y \* numpy.sqrt((numpy.abs(z)\*\*2).mean() / (numpy.abs(y)\*\*2).mean())

return y

def BrownNoise(N, mag):

# Brown noise.

# Power decreases with -3 dB per octave.

# Power spectral density decreases with 6 dB per octave.

# N Length of sample array, mag magnitude scaling factor

x = numpy.random.normal(0.0, 1, N).astype(numpy.float32) # white Noise

X = numpy.fft.rfft(x) / N

S = numpy.arange(X.size)+1 # Filter

y = numpy.fft.irfft(X/S).real[:N]

z = numpy.ndarray = mag

y = y \* numpy.sqrt((numpy.abs(z)\*\*2).mean() / (numpy.abs(y)\*\*2).mean())

return y

def VioletNoise(N, mag):

# Violet noise.

# Power increases with +9 dB per octave.

# Power density increases with +6 dB per octave.

# N Length of sample array, mag magnitude scaling factor

x = numpy.random.normal(0.0, 1, N).astype(numpy.float32) # white Noise

X = numpy.fft.rfft(x) / N

S = numpy.arange(X.size) # Filter

y = numpy.fft.irfft(X\*S).real[0:N]

z = numpy.ndarray = mag

y = y \* numpy.sqrt((numpy.abs(z)\*\*2).mean() / (numpy.abs(y)\*\*2).mean())

return y

### Digital single pole RC filter

This function is used mainly for compensating the response of any analog input resistor divider networks placed ahead of the analog inputs but can be used for any other filtering needs such as filtering AWG waveforms. The “RC” time constant is passed in uSec and the Gain can be either positive for high pass function or negative for low pass function.

Code:

## Digital RC filter function for input divider frequency compensation

# TC1 is in micro seconds

def Digital\_RC\_High\_Pass( InBuff, TC1, Gain ):

global SAMPLErate, Two\_X\_Sample

OutBuff = []

n = len(InBuff)

if Two\_X\_Sample.get() == 0:

Delta = 1.0/SAMPLErate

else: # adjust for sligh difference in 2X sample mode?

Delta = 0.88/SAMPLErate

TC = TC1 \* 1.0E-6

Alpha = TC / (TC + Delta)

OutBuff.append(0.0) # initialize first output sample

i = 1

while i < n:

OutBuff.append( Alpha \* (OutBuff[i-1] + InBuff[i] - InBuff[i-1]) )

i += 1

OutBuff = numpy.array(OutBuff)

OutBuff = InBuff + (OutBuff \* Gain)

return OutBuff

### Digital Filter Coefficient Generators

Higher order SINC filters can be generated by convolving SINC1 filters. For example, convolving two SINC1 filters (with a rectangular impulse response in time) will result in a SINC2 response, with a triangular impulse response.

Code:

#

# Higher order SINC filters can be generated by convolving first order Box Car filters

def BuildRejectFilter(Order, Freject, Fsample):

# Order can be 1, 2, 3 or 4

# Fsample = 100000

# Calculate SINC1 oversample ratios for Freject

osr = int(Fsample/Freject) #

# osr60 = int(Fsample/60) # 60 Hz example

# Create "boxcar" SINC1 filter

sinc1 = numpy.ones(osr)

# sinc1\_60 = np.ones(osr60)

# Calculate higher order filters

sinc2 = numpy.convolve(sinc1, sinc1)

sinc3 = numpy.convolve(sinc2, sinc1)

sinc4 = numpy.convolve(sinc2, sinc2)

fosr = float(Fsample/Freject)

if Order == 1:

return sinc1/fosr

elif Order == 2:

return sinc2/fosr

elif Order == 3:

return sinc3/fosr

elif Order == 4:

return sinc4/fosr

else:

return sinc1/fosr

# Here's the SINC4-ish filter

# with three zeros at 50Hz, one at 60Hz.

# filt\_50\_60\_rej = np.convolve(sinc3\_50, sinc1\_60)

### Rearranging the order of samples:

Sometimes there is a need to rearrange the data samples in an array. The length of the array is not changed just the order of the samples. These two functions, Wrap and Unwrap, are inverses of each other.

Code:

def Wrap(InArray, WrFactor):

# Build new array by skipping WrFactor samples and wrapping back around

# [1,2,3,4,5,6} becomes [1,3,5,2,4,6]

# effectively multiplies the frequency content by WrFactor

OutArray = []

OutArray = numpy.array(OutArray)

InArray = numpy.array(InArray)

EndIndex = len(InArray)

StartIndex = 0

while StartIndex < WrFactor:

OutArray = numpy.concatenate((OutArray, InArray[StartIndex:EndIndex:WrFactor]), axis=0)

StartIndex = StartIndex + 1

return OutArray

#

Code:

def UnWrap(InArray, WrFactor):

# Build new array by splitting arrray into WrFactor sections and interleaving samples from each section

# [1,2,3,4,5,6} becomes [1,4,2,5,3,6]

# effectively divided the frequency content by WrFactor

OutArray = []

InArray = numpy.array(InArray)

EndIndex = int(len(InArray)/WrFactor)

StartIndex = 0

while StartIndex < EndIndex:

LoopIndex = 0

while LoopIndex < WrFactor:

OutArray.append(InArray[StartIndex+LoopIndex])

LoopIndex = LoopIndex + 1

StartIndex = StartIndex + 1

OutArray = numpy.array(OutArray)

return OutArray

#

The following function opens a .wav audio file and writes the passed array of samples to it.

Code:

def Write\_WAV(data, repeat, filename):

global SAMPLErate

# write data array to mono .wav file 100KSPS

# copy buffer repeat times in output file

# Use : Write\_WAV(VBuffB, 2, "write\_wave\_1.wav")

wavfile = wave.open(filename, "w")

nchannels = 1

sampwidth = 2

framerate = SAMPLErate

amplitude = 32766

nframes = len(data)

comptype = "NONE"

compname = "not compressed"

wavfile.setparams((nchannels,

sampwidth,

framerate,

nframes,

comptype,

compname))

# Normalize data

ArrN = numpy.array(data)

ArrN /= numpy.max(numpy.abs(data))

frames = []

for s in ArrN:

mul = int(s \* amplitude)

# print "s: %f mul: %d" % (s, mul)

frames.append(struct.pack('h', mul))

print( len(frames))

frames = ''.join(frames)

print( len(frames))

for x in xrange(0, repeat):

print( x )

wavfile.writeframes(frames)

wavfile.close()

#

### Calculation of Relative Phase

There are a number of possible ways to measure the phase difference between two signals. The main method used by the ALICE measurements is by finding “zero-crossing” points in each waveform and calculating the phase from the time samples. A second method, used in the Phase Analyzer tool ( and Bode plotter) is the FFT. The following function calculates the relative based on this formula:

However, the results are incorrect if either of the two signals has a DC offset. The function removes any DC content. Also the result is only from 0 to +180º. It lacks quadrature calculation for a definitive + or - sign for the phase, which requires a bit more calculation.

This method is strictly speaking only for use with sine wave signals of the same frequency but results for triangle waves is reasonably close and for square waves is only correct at 0 and 180º.

Code:

# Function to calculate relative phase angle between two sine waves of the same frequency

# Removes any DC content

def Sine\_Phase():

global DCV1, DCV2, VBuffA, VBuffB

sum1 = 0.0

sum2 = 0.0

sum12 = 0.0

i = 0

n = len(VBuffA)

while i < n:

sum1 += (VBuffA[i]-DCV1)\*(VBuffA[i]-DCV1)

sum2 += (VBuffB[i]-DCV2)\*(VBuffB[i]-DCV2)

sum12 += (VBuffA[i]-DCV1)\*(VBuffB[i]-DCV2)

i += 1

return math.acos(sum12/math.sqrt(sum1\*sum2))\*180.0/numpy.pi

#

## Curve Fitting:

The numpy library contains a polynomial fitting function. The following exponential curve fitting function is contained in ALICE.

Code:

# Fit the function y = A \* exp(B \* x) to the data arrays xs and ys

# returns (A, B)

# From: https://mathworld.wolfram.com/LeastSquaresFittingExponential.html

def fit\_exp(xs, ys):

S\_x2\_y = 0.0

S\_y\_lny = 0.0

S\_x\_y = 0.0

S\_x\_y\_lny = 0.0

S\_y = 0.0

for (x,y) in zip(xs, ys):

S\_x2\_y += x \* x \* y

S\_y\_lny += y \* numpy.log(y)

S\_x\_y += x \* y

S\_x\_y\_lny += x \* y \* numpy.log(y)

S\_y += y

#end

a = (S\_x2\_y \* S\_y\_lny - S\_x\_y \* S\_x\_y\_lny) / (S\_y \* S\_x2\_y - S\_x\_y \* S\_x\_y)

b = (S\_y \* S\_x\_y\_lny - S\_x\_y \* S\_y\_lny) / (S\_y \* S\_x2\_y - S\_x\_y \* S\_x\_y)

return (numpy.exp(a), b)

#

As an example use case the following simple resistor and diode circuit shown in figure 1 is offered.

Figure 1, Diode test circuit

The voltage across a small signal diode is measured using scope channel B and the diode current is measured using the AWG A channel current as the channel A voltage is swept using a triangle wave. The data samples in the VBuffB and IBuffA arrays is used to fit an exponential. In the following ALICE script, the data is analyzed and then plotted using the built-in pyplot functions from the Matplotlib library.

Note that this script uses the User Entry Widgets on the X-Y plotting screen to display the fit values for IS and N. You will need to have the User Entries enabled in the alice\_init.ini file: EnableUserEntries = 1

Alternatively these lines can be commented out and the values simply printed to the Console screen.

xs = VBuffB[50:500] # diode voltage

ys = IBuffA[50:500] # diode current

ys = ys / 1000.0 # convert mA to Amps

ys = numpy.absolute(ys) # make all y values positive for taking ln

ys = ys - numpy.amin(ys) + 2.2e-9 # add offset and Is guess

# Function to fit data I\_d(x) + I\_s = I\_s \* exp(x/(n\*Vt))

# note that I\_d + I\_s is approx I\_d since I\_s is small

(A, B) = fit\_exp(xs, ys)

# some constants

# Saturation current I\_s = 1.0e-9

# Ideality factor n = 2

# Thermal voltage, KT/q

Vt = 0.0259

# guess values for Is and n

# Iguess(x) = 5.0e-9\*(exp(x/(2\*Vt))-1.0)

#print( "{Is} A = ", A, "B = ", B )

Fit\_N = 1.0/(Vt\*B) # Fit n with Vt at 25 C

Is\_String = ' {0:.2e} '.format(A)

AmA = A \* 1000.0

User3Entry.delete(0,END)

User3Entry.insert(5, Is\_String)

N\_String = ' {0:.2f} '.format(Fit\_N)

User4Entry.delete(0,END)

User4Entry.insert(5, N\_String)

#print ("Vt = ", Vt, "Fit Vt = ", Fit\_Vt)

plt.figure()

plt.plot(VBuffB[0:500], IBuffA[0:500], 'g', label='Raw Data')

plt.plot(VBuffB[0:500], [AmA \* (numpy.exp(B\*x)-1) for x in VBuffB[0:500]], 'b', label='Fit')

#plt.plot(xs, [2.2e-9 \* (numpy.exp(x/(2.0\*0.0259))-1) for x in xs], 'r', label='Guess')

plt.title('Exponential Diode Fit')

plt.xlabel('Volts')

plt.ylabel('Amps')

plt.legend(loc='best')

plt.tight\_layout()

plt.show(block=False)

The following screenshots show the results:

## Using the Numpy library

ALICE includes the Numpy numerical library of array creation and manipulation functions. The reader is directed to the [[https://docs.scipy.org/doc/numpy/reference/index.html |numpy documentation]] for complete details on these functions. Here we will point out some of the more useful functions for creating and manipulating waveform sample arrays. Numpy contains many more than can be covered here. However, be sure to only use functions that return 1 dimensional arrays.

In these example we use AWGAwaveform as the array variable but any of the ALICE internal waveform arrays can be of course used.

### Array Creation:

numpy.ones(length) Return a new array of given length filled with ones.

numpy.zeros(length) Return a new array of given length filled with zeros.

numpy.full(length, fill\_value) Return a new array of given length, filled with fill\_value.

numpy.linspace(start\_value, stop\_value, num=length) Return a new array of given length of evenly spaced numbers between start\_value and stop\_value.

numpy.logspace(start\_value, stop\_value, num=length, base=log\_base) Return a new array of given length of numbers spaced evenly on a log scale. The base of the log can be optionally specified such as 10 or 2 etc.

### Arithmetic functions:

numpy.square(x) Return the element-wise square of the input.

numpy.sqrt(x) Return the positive square-root of an array, element-wise.

numpy.exp(x) Calculate the exponential of all elements in the input array.

numpy.log(x) Return the Natural logarithm, element-wise.

numpy.log10(x) Return the base 10 logarithm of the input array, element-wise.

### Trigonometric functions:

numpy.sin(x) Trigonometric Sine, element-wise.

numpy.cos(x) Cosine element-wise.

To create one cycle of a sine wave 400 samples long you will first create an array of values from 0 to 2\*pi and then send it to the sine function like this.

numpy.sin(numpy.linspace(0, 2\*numpy.pi, 400))

The waveform values will be from -1 to 1 so additionally you will need to scale and or offset the values to be between 0 than 5 for the AWG. In this example we create the sine wave centered on 2.5 V with a P-P of 4 V.

(numpy.sin(numpy.linspace(0, 2\*numpy.pi, 400)) \* 2) + 2.5

numpy.sinc(x) Return the sinc function.

Much like the trig functions the input to the sinc function is a linear spaced array of points.

numpy.sinc(numpy.linspace(-4, 4, 400)) will product 4 “cycles” 400 samples long.

The values will be between -1 to 1 so additionally you will need to scale and or offset the values to be between 0 than 5 for the AWG. In this example we create the sinc pulse centered on 2.5 V with a peak value of 4.5 V.

(numpy.sinc(numpy.linspace(-4, 4, 400)) \* 2)+2.5

### Rearranging Sample points:

numpy.roll(AWGAwaveform, shift) Roll array elements by shift points. This will in effect change the relative timing delay or phase of the waveform.

### Functions to extend waveform:

numpy.concatenate((AWGAwaveform, AWGBwaveform,…)) Join a sequence of arrays.

numpy.repeat(AWGAwaveform, repeats) Repeat elements of an array. This will effectively lower the sample rate of the waveform. If repeat is 2 the frequency of the new waveform will be ½ what the original was.

The pad function adds samples to the beginning and end of the array.

numpy.pad(AWGAwaveform, (100, 100), 'edge')

numpy.pad(AWGAwaveform, (100,100), 'maximum')

The first argument is the array variable, next is a list of the number of points to add. In the case of our one dimensional waveforms this is just two values for the beginning and end of the array. The third argument tells the function what values to use to extend the array. How the array is extended can be one of the following:

‘constant’ - Pads with a constant value.

‘edge’ - Pads with the edge values of array.

‘linear\_ramp’ - Pads with the linear ramp between end\_value and the array edge value.

‘maximum’ - Pads with the maximum value of all or part of the vector along each axis.

‘mean’ - Pads with the mean value of all or part of the vector along the axis.

‘median’ - Pads with the median value of all or part of the vector along the axis.

‘minimum’ - Pads with the minimum value of all or part of the vector along the axis.

‘reflect’ - Pads with the reflection of the vector mirrored on the first and last values of the vector along the axis.

‘symmetric’ - Pads with the reflection of the vector mirrored along the edge of the array.

‘wrap’ - Pads with the wrap of the vector along the axis. The first values are used to pad the end and the end values are used to pad the beginning.

### Window functions:

numpy.bartlett(length) Return the Bartlett window.

numpy.blackman(length) Return the Blackman window.

numpy.hamming(length) Return the Hamming window.

numpy.hanning(length) Return the Hanning window.

numpy.kaiser(length, beta) Return the Kaiser window

### Special functions:

numpy.convolve(a, v) Returns the discrete, linear convolution of two one-dimensional sequences.

This is used primarily for digital filtering of waveform data arrays.

numpy.polyfit(x, y, deg) Fit a polynomial p(x) = p[0] \* x\*\*deg + ... + p[deg] of degree deg to points (x, y). Returns a vector of coefficients p that minimizes the squared error.

numpy.poly1d(p) A convenience class, used to encapsulate “natural” operations on polynomials so that said operations may take on their customary form in code.

The following example shows how to use polyfit and poly1d to fit a 5th order polynomial to the voltage characteristics of diode and plot the polynomial over the plot of the measured data points.

First construct the simple resistor and diode circuit shown in the figure.

CB-In

R1 1 kΩ

GND

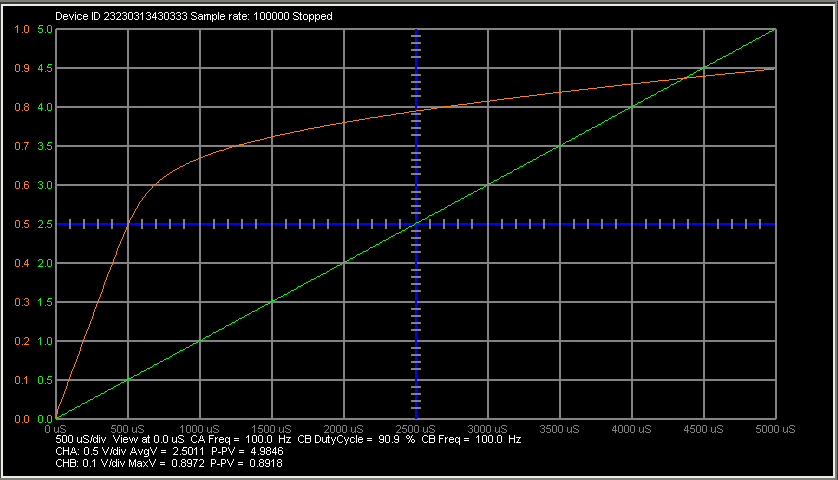
CA-In

D1 1N914

Figure, Diode test circuit

Set Channel A AWG Min value to 0V and Max value to 5V. Set the Mode to SVMI and the Shape to triangle. Set the Freq to 100 Hz. Set Channel B mode to Hi-Z to measure the voltage across the diode.

Set the Horz Time scale to 0.5mSec/Div. Hit Run, wait for a few seconds to capture some data then hit Stop. This should display the rising half of the triangle wave on Channel A from 0 to 5 V ( green trace ). The width of the grid will be 500 sample points (5 mSec at 10 uSec/sample). Channel B should display the voltage across the diode going from 0 to about 0.8 V (orange trace). You may want to change the vertical scale to 0.1 V/div and position to 0.5 for CH-B to display the waveform from 0 to 1 V. You should now have something like this:



Plot of Diode Voltage

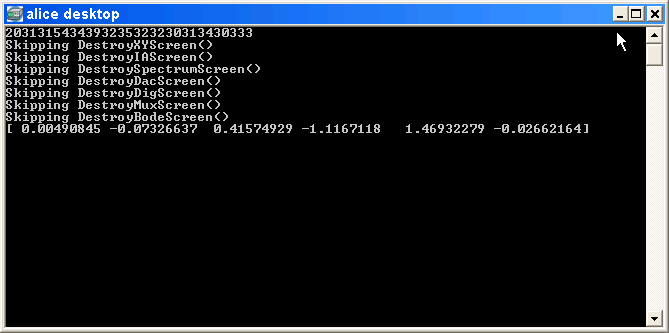
Open the Command Line interface ( with the program stopped ). We want to fit a polynomial to the first 500 samples where CH-A ramps from 0 to 5 V. Type the following line into the entry space and hit return.

global Zpoly; Zpoly = numpy.polyfit(VBuffA[0:499], VBuffB[0:499], 5)

To check the terms of the polynomial type the following line into the entry space and hit return.

print Zpoly

In the ALICE desktop console window you should see something like this.



ALICE console showing polynomial terms

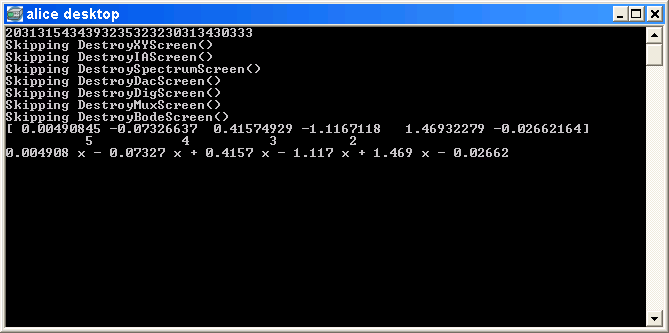
We can use poly1d to make an object that makes this easy to plot on the screen. Type the following line into the entry space and hit return.

global ZBuff; ZBuff = numpy.poly1d(Zpoly)

Again to check the results type the following line into the entry space and hit return.

print ZBuff

In the ALICE desktop console window you should now see something like this.

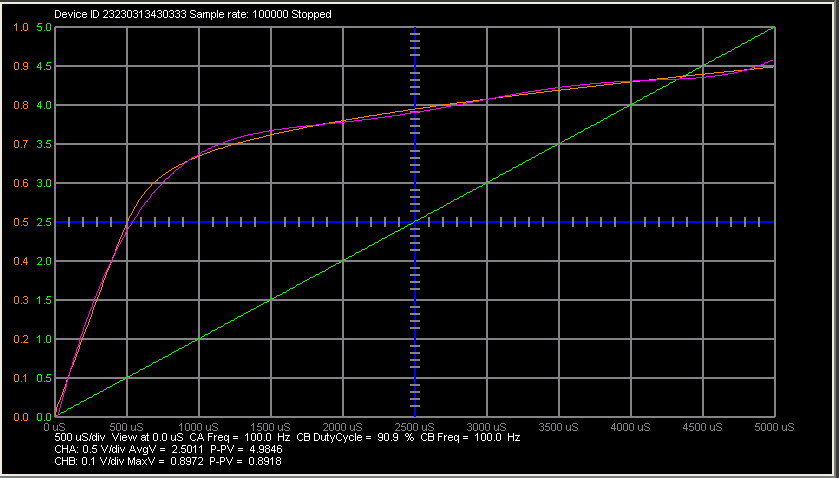


ALICE console showing polynomial equation

To plot the polynomial on the screen we will use the Math waveform feature. From the Math drop down menu select Math Axis and set it to V-B to use the same axis as the diode voltage plot. From the Math drop down menu select Enter Formula and enter the following:

ZBuff(VBuffA[t])-CHBOffset

This plots the value of the polynomial evaluated at each point in VBuffA as the time index t goes from 0 to 499 ( 5 mSec ). Be sure to note that () are used for ZBuff and not [] because it is a function and not an array like VBuffA. You should now see something like this on the display. The magenta Math plot is the polynomial.



Plot of measured data and polynomial

### Random functions:

Many of the random number functions return arrays of random numbers. Here are a few examples:

numpy.random.standard\_normal(8000)+2.5 will return a 8,000 sample array of random numbers with a normal distribution, standard deviation = 1, centered on 2.5.

numpy.random.uniform(1,4,10000) will return a 10,000 sample array of random numbers with a uniform distribution between 1 and 4.

numpy.random.triangular(1, 2.5, 4, 10000) will return a 10,000 sample array of random numbers with a triangular distribution between 1 and 4, centered on 2.5.

## Appendix:

For completeness, here are a few of the more obscure arrays used in ALICE:

These trace “lines” are 2d X-Y arrays in screen pixels.

T1Vline = [] # Voltage Trace line channel A

T2Vline = [] # Voltage Trace line channel B

T1Iline = [] # Current Trace line channel A

T2Iline = [] # Current Trace line channel B

TMAVline = [] # Voltage Trace line MUX channel A

TMBVline = [] # Voltage Trace line MUX channel B

TMCVline = [] # Voltage Trace line MUX channel C

TMDVline = [] # Voltage Trace line MUX channel D

TMBRline = [] # V reference Trace line MUX channel B

TMCRline = [] # V reference line MUX channel C

TXYline = [] # XY Trace line

TXYRline = [] # XY reference trace line

Tmathline = [] # Math trace line

T1VRline = [] # V reference Trace line channel A

T2VRline = [] # V reference Trace line channel B

T1IRline = [] # I reference Trace line channel A

T2IRline = [] # I reference Trace line channel B

TMRline = [] # Math reference Trace line

T1Fline = [] # Frequency Trace line channel A

T2Fline = [] # Frequency Trace line channel B

T1Pline = [] # Phase angle Trace line channel A - B

T2Pline = [] # Phase angle Trace line channel B - A

T1FRline = [] # F reference Trace line channel A

T2FRline = [] # F reference Trace line channel B

T1PRline = [] # Phase reference Trace line channel A - B

T2PRline = [] # Phase reference Trace line channel B - A

TFMline = [] # Frequency Math Trace

TFRMline = [] # Frequency reference Math Trace

TAFline = [] # Bode Freq Trace line channel A

TBFline = [] # Bode Freq Trace line channel B

TAPline = [] # Bode Phase angle Trace line channel A - B

TBPline = [] # Bode Phase angle Trace line channel B - A

TAFRline = [] # Bode F reference Trace line channel A

TBFRline = [] # Bode F reference Trace line channel B

TAPRline = [] # Bode Phase reference Trace line channel A - B

TBPRline = [] # Bode Phase reference Trace line channel B - A

TBPMline = [] # Bode Frequency Math Trace

TBPRMline = [] # Bode Frequency reference Math Trace

**For Further Reading:**

**Return to Lab Activity Table of Contents.**