Waveform_Viewers-Plotting_Large_Analog_Data

Discussion about this page on: Wiki page discuss: Waveform Viewers - Plotting Large Analog Data - Ubuntu Forums
A backup of the scripts discussed here @ sourceforge

Contents

Contents

Introduction

A situation often occurs, where the user ends up with some sort of a large dataset that needs to be visualised and analysed. Examples of this include:

- typical audio files
- data from statistical or mathematical analysis (using, say, R or scilab);
- simulation results data from electronic circuit simulators like ngspice;
- data downloaded from digital oscilloscopes (see, for instance, agiload)

Of course, there are many data formats, and approaches to visualisation and analysis; however, the data properties that are in focus here are:

- Each data point is a *real* number (or in other words, a *floating point* number)
- There may be many (million or more) data points in a dataset (*channel*)
- There may be several datasets (*channels*) that need to be viewed simultaneously, in respect to the *same* abscissa (x-axis)

Thus, what we'd like to have is an **interactive** viewer - one where we can zoom in, pan and focus data in details, and possibly annotate points of interest, interactively and *quickly*; the following quotes exemplify the typical expectations:

Anybody implement graph zoom with mouse wheel? - NI Discussion Forums

```
I need to implement a zoom feature with the middle mouse wheel (like Google Maps, Google Earth, etc) on a CNiGraph.  So if the mouse
```

Searching simple waveform viewer - Comp.Arch.Embedded - EmbeddedRelated.com

```
I think your own search failed because you used an overly restrictive
keyword.  The kind of program you're looking for is a plotting program.
  "Waveforms" are just one type of plottable data file.
...
No, I am not looking for realtime display (would be nice later on maybe),
but I search an 'interactive' off-line viewer. Load the file and have the
ability to zoom and scroll through the graph. The timebase of the graph
will be fairly long and I need to be able to zoom in to some details.
```

However, the problem is that, typically, we have to work with large datasets (millions of points) - or rather, the complete opposite of the sentiment expressed below:

Why doesn't Python release the memory when I delete a large object?

```
/.../ Do you really need a list containing 5 million integers? I never do ;-)
```

Typically, this is the domain of so-called "Waveform viewer" software; which is, however, dominated by proprietary (and sometimes rather expencive) offerings. What this article addresses, then, is a comparison between open-source tools that could be used for this purpose.

To illustrate the tools and their uses, here we focus on data which is in ASCII format, and simply represents a list of real values, one value per line (each line delimited with '\n') - as in:

```
-1.5
2.6
-3.2
...
```

First we will generate such example data using several Linux tools - and then we will proceed with its visualisation using different tools. The procedure described here was performed on Ubuntu 10.04, on a MSI Wind U135 netbook (1G RAM, 1GHz typically / 1.67GHz in 'performance mode', >20GB hard disk space).

Generating data

Let's say one dataset is 1 million points of real values, provided as ASCII list of one value per line, linefeed ('`\n`') delimited. Let's assume that these were obtained from an electric circuit simulation, or an oscilloscope measurement, at a sampling rate of `fs=25MHz` - meaning that the sampling period between each point is `Ts=1/25e6=4e-8` or 40ns. Thus, the entire span of 1 million points represents time series, encompassing a 40ms (`1e6*4e-8`) time period. Additionally, we will try to limit the number of decimal places in value strings (simulating, in some sense, the sampling resolution limit in typical acquisition hardware)

To make this more interesting, let's say we have two channels (datasets) - each representing an FM-like signal of the kind '`sin(x*sin(x))`' (*or rather, like "*$100 \cdot t \cdot \sin(\omega_b t) \cdot \sin\left(\omega_a t \cdot \sin(\omega_b t)\right)$*", so the amplitude changes as well*). Additionally, each channel has different frequencies (`fa1=3KHz, fb1=50Hz` vs. `fa2=4KHz, fb2=60Hz`) - and we should find the time between the xth (in channel 1) and yth (in channel 2) signal transition over an arbitrary level, positive (or rising) edge triggered.

- *Note that it is probably easier to solve these types of problems by hand or with a script; but sometimes we don't exactly know the values we are looking for - which is why a quick visual inspection of data (and a waveform viewer to do it with) is valuable.*
- The problem above is also not completely specified; it is just mentioned, in order to show the typical type of problems that require use of waveform viewers. For instance, it can be noticed that the FM-like signal above has regions where the frequency, quite visibly, slows down - typically, we would like to identify where these regions start or end; or what is the delay between the starts of the xth such region in first vs. second channel. The examples below will try to illustrate use of the software in similar spirit (but without any concrete goal).

Given the size of the final ASCII-text data files, the easiest way to browse them as text files would be to use `less` directly in the terminal (or even quicker, head or `tail`).

Bash and wcalc

In Ubuntu, `sudo apt-get install wcalc` should get you sorted for this step.

One of the first things one may think of in Linux, when faced with generating numeric ASCII data, is to use the `bash` and a calculator, such as `wcalc`, to generate this data. And that is indeed possible, as demonstrated below - although quite inefficient (in this case, at least).

Note that `wcalc` doesn't in itself have a concept of program loops, so we basically use `bash` to implement the loop (and call `wcalc` at each step). Below is a session log of `bash` commands; you can either:

- save them as a script (say `gendata.sh`) and process them using `./gendata.sh` (*note, this requires that you made the file executable `chmod +x gendata.sh` previously*); or
- you can paste/type them in the `bash` terminal console:

```
# clean data files first, as we append to them
# ignore nonexistant files with -f
rm -f *.wcalc.dat

# generate time axis - increasing steps of period
# 'seq' includes both start and end: 0, 1, 2, ... 999999, 1000000
# and also, generate data in same loop
fa1=3e3; fb1=50; fa2=4e3; fb2=60;
for t in $(seq 0 1 1000000)
do
        # recalculate time base
        t=$(wcalc -q "${t}*(1/25e6)")
        # calculate value - chan1 and chan2
        v1=$(echo "100*${t}*sin(2*pi*${fb1}*${t})*sin(2*pi*${fa1}*${t}*sin(2*pi*${fb1}*${t}))" | wcalc -q)
        v2=$(echo "100*${t}*sin(2*pi*${fb2}*${t})*sin(2*pi*${fa2}*${t}*sin(2*pi*${fb2}*${t}))" | wcalc -q)
        # output value, formatted to 5 decimal spaces
        # make sure locale is C, else possibly decimal point '.' may fail to parse
        v1f=$(export LC_ALL=C; printf "%.5f" "$v1")
        v2f=$(export LC_ALL=C; printf "%.5f" "$v2")
        echo $v1f | tee -a chan1.wcalc.dat
        echo $v2f | tee -a chan2.wcalc.dat
done
```

While this will generate the requested data values - **note** that for 1000000 points, there will be some 10 seconds delay before output starts being generated (try a value up to 1000 to see results faster)! Even worse, regardless of the actual ammount of requested points - and even if we do _not_ save to disk (*i.e. the `tee` commands above are removed*) - these commands seem to use the harddisk at each step regardless, which slows down the process; in some 10 seconds, you might get only up to 50-100 values in one file; which means for a million points, you'd have to wait 10000 seconds - or 166 minutes - which is more than a couple of hours, before the files are fully generated!

Using `scilab` or `python` to generate ASCII value data files will bring down the generation time in tens of seconds, and these methods are discussed below.

Scilab

In Ubuntu, `sudo apt-get install scilab` should get you sorted for this step.

Below is a session log of `scilab` commands; you can either:

- save them as a script (say `gendata.sci`) and process them using `scilab -nw -f gendata.sci` (*note, this will still enter the console, put an `exit` command at end of script to exit automatically*); or
- you can paste/type them in the `scilab` command line console (*use `scilab -nw` to start it*):
  - *Note that `scilab -nw` simply starts "without specialized Scilab Window", and so the console goes directly in the Terminal; however, in this mode, `scilab` you must press Enter to recognize key presses - important for stopping printout of large data!*

```
// increase stacksize - else scilab may run out of memory
stacksize max

// generate time axis - increasing steps of period
// form implicit vector using colon symbol:
// from start 0, using step 1, until you reach 1000000
// semicolon suppresses output
// the ' transposes, so we get 1-column vector
t=[0:1:1000000]';

// printout size - should be '1000001.    1.' (rows, columns)
size(t)

// printout first and last element - should be '0, 1000000'
printf("%d, %d\n", t(1), t(1000001))

// scale this vector (array) with the period duration of fs=25MHz
t=t*(1/25e6);

// printout first and last element - should be '0.000000, 0.040000'
printf("%f, %f\n", t(1), t(1000001))

// generate data according to equations
// careful to use element-by-element vector multiplication (.*) when needed
fa1=3e3; fb1=50; fa2=4e3; fb2=60;
chan1= 100*t.*sin(2*%pi*fb1*t).*sin(2*%pi*fa1*t.*sin(2*%pi*fb1*t));
chan2= 100*t.*sin(2*%pi*fb2*t).*sin(2*%pi*fa2*t.*sin(2*%pi*fb2*t));

// printout size - same as t
size(chan1)

// save data as files - use 'write' for formatted text file
// note, if chan1 was a single-row, many-column vector,
//   'write(%io(2),chan1)' would fail with 'Incorrect file or format'!
// also, 'write' fails if file exists
// format specifier - fortran (http://www.ichec.ie/support/tutorials/fortran.pdf)
// f0.5: 0 is total places expand (but leading 0 is dropped); 5 is decimal places limit
// these two commands may take some 15 sec each to execute..
write('chan1.sci.dat',chan1,'(f0.5)')
write('chan2.sci.dat',chan2,'(f0.5)')
```

These commands should generate two ASCII files of around 7MB each:

```
$ for ix in {-h,-b} ; do du $ix *.sci.dat | column; done
7,2M    chan1.sci.dat   7,2M    chan2.sci.dat
7485619 chan1.sci.dat   7488995 chan2.sci.dat
```

Python with numpy

In Ubuntu, `sudo apt-get install python-numpy` (contains numpy) should get you sorted for this step.

Below is a session log of `python` commands; you can either:

- save them as a script (say `gendata.py`) and process them using `python gendata.py`; or
- you can paste/type them in the `python` command line console (*use `python` to start it*):

```
import numpy

# generate time axis - increasing steps of period
# use arrange - from min to max using step
# up to 1000001 so we match scilab example
t = numpy.arange(0,1000001,1)

# printout size - should be 1000001
print len(t)

# printout first and last element - should be '0, 1000000'
print "%d, %d" % (t[0], t[len(t)-1])

# scale this array with the period duration of fs=25MHz
t=t*(1/25e6)

# printout first and last element - should be '0.000000, 0.040000'
print "%f, %f" % (t[0], t[len(t)-1])
```

```
# generate data according to equations
# no need to be careful for element-by-element vector multiplication (.*) here;
# here we work with 'arrays', not mathematical 'vector' concepts
# (so, only element-by-element multiplication is possible by default) :
fa1=3e3; fb1=50; fa2=4e3; fb2=60; pi=numpy.pi;
chan1= 100*t*numpy.sin(2*pi*fb1*t)*numpy.sin(2*pi*fa1*t*numpy.sin(2*pi*fb1*t));
chan2= 100*t*numpy.sin(2*pi*fb2*t)*numpy.sin(2*pi*fa2*t*numpy.sin(2*pi*fb2*t));

# printout size - same as t
print len(chan1)

# save data as files
# format specifier - C
# "%.5f": total places expand (but leading 0 is not dropped); 5 is decimal places limit
# these two commands may take some 25 sec each to execute..
numpy.savetxt("chan1.py.dat",chan1,fmt="%.5f")
numpy.savetxt("chan2.py.dat",chan2,fmt="%.5f")
```

These commands should generate two ASCII files of around 8MB each:

```
$ for ix in {-h,-b} ; do du $ix *.py.dat | column; done
8,1M    chan1.py.dat    8,1M    chan2.py.dat
8485619 chan1.py.dat    8488995 chan2.py.dat
```

wrap-up

We've seen that we can use different tools to generate ASCII real data, however, only `scilab` and `python`, in this case at least, perform reasonably fast. At the moment, we could choose one or the other as the data set to work with - there shouldn't be (m)any differences between the two datasets:

```
# using diff will find difference in each line due to missing 0 in *sci.dat
# so, use 'column' to visually inspect a snippet:
$ tail --lines=100 chan1.py.dat | head --lines 10 | column ; tail --lines=100 chan1.sci.dat | head --lines 10 | colu

0.00401 0.00394 0.00388 0.00381 0.00374 0.00367 0.00361 0.00354 0.00347 0.00341
.00401  .00394  .00388  .00381  .00374  .00367  .00361  .00354  .00347  .00341
```

In this case, I prefer the output from `python` simply because it includes the leading zero, so let's choose the `python` datasets as the one we're going to use in the rest of this tutorial:

```
cp chan1.py.dat chan1.dat
cp chan2.py.dat chan2.dat
```

Viewing data

In this section, we will try to visualise the ASCII data files `chan1.dat` and `chan2.dat` (created previously) using different tools. As the data files could, in principle, be obtained in a different way - the first idea would be usually to use the default GNU software for the purpose - so we'll start with an outline of that.

kst

Added later, so without an example - but check out Kst (or `kst-plot`) for KDE; if you have CPU problems when running in under Gnome (e.g. in 11.04 Natty), try the Windows build of e.g. version 2.0.7 in `wine`: it is rather fast when browsing (zooming), can handle big data, and has equations to pre-process data within the application.

gwave

In Ubuntu, `sudo apt-get install gwave` should get you sorted for this step.

The Gwave (homepage) software, which also goes by the name of gnuwave (not to be confused with Google's offering known as 'Google Wave'), is a waveform viewer, that "*can read "raw" files from spice2G6, spice3F5 or ngspice, and a tabular ASCII format suitable for use with GnuCAP or homegrown tools.*"

While this description looks promising - unfortunately, our single-colimn, raw ASCII data in `chan1.dat` and `chan2.dat` can not be read directly by gwave That is because gwave expects at least two columns of data in ASCII data files, the first representing the time base - and also, preferably the first row should contain data labels. Also, there is no manual entry for gwave - nor a 'help' command line option; so the best place to look for documentation is the source code (starting with the README and src/gwave.c).

We can quickly try to add a label to a data file, and then load it in gwave:

```
$ echo "$(echo "chan1val"; cat chan1.dat)" > chan1-label.dat
# we could have also done inline file replace with 'sed':
# $ sed -i '1i chan1val' chan1.dat
# in order to add this first line, however that
#  takes approx just as long as the above echo/cat combo

$ gwave chan1.dat
WARNING: (gnome gtk): ...
[wavefile_read]: <<ERROR>> independent variable is not nondecreasing at row 1; ival=0.00201 last_ival=0.00202
```

... however, we can see that gwave tries to interpret the first column as a timebase, and so demands that it has non-decreasing values.

That means, we should now, basically: load the data; generate corresponding time-base; and then export the new timebase and the old values as two-column ASCII data ☹ And, as we can guess, this may be inefficient doing directly in `bash` - so we should start up a math package again.
In `scilab`, we could do:

```
// read2col.sci
// increase stacksize - else scilab may run out of memory
stacksize max

// read the saved ASCII data
chan1=read("chan1.dat",-1,1);
chan2=read("chan2.dat",-1,1);

tsz=size(chan1) // should be "1000001.    1."

// recreate time array
t=(1/25e6)*[0:1:tsz(1)-1]';

// write out concatenated data columns
write('chan1t.dat', [t chan1], '(e12.5'' ''f0.5)')
write('chan2t.dat', [t chan2], '(e12.5'' ''f0.5)')

// gives big files:
// $ for ix in {-h,-b} ; do du $ix *t.dat | column; done
// 20M  chan1t.dat      20M     chan2t.dat
// 20802955     chan1t.dat      20813281        chan2t.dat
```

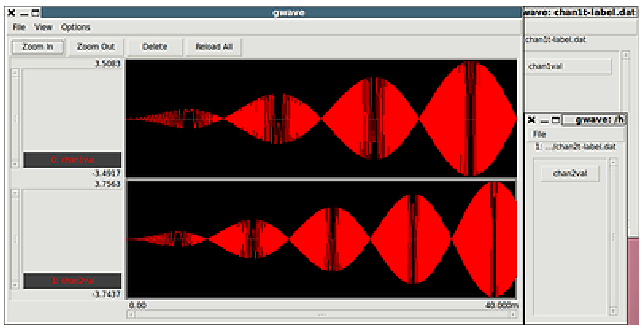
or in `python`, we could do:

```
# read2col.py
import numpy

# read the saved ASCII data
chan1=numpy.loadtxt("chan1.dat")
chan2=numpy.loadtxt("chan2.dat")

tsz=len(chan1)
print tsz # should be "1000001"

# recreate time array
t=(1/25e6)*numpy.arange(0,tsz,1)

# write out concatenated data columns
# note: http://stackoverflow.com/questions/1544948/python-numpy-savetxt
# test: numpy.savetxt(sys.stdout,ct1,fmt="%.5e %.5f")
ct1=numpy.array(zip(t, chan1), dtype=[('t', 'd'), ('ch1', 'd')])
ct2=numpy.array(zip(t, chan2), dtype=[('t', 'd'), ('ch2', 'd')])

numpy.savetxt("chan1t.dat",ct1,fmt="%.5e %.5f")
numpy.savetxt("chan2t.dat",ct2,fmt="%.5e %.5f")

#~ gives big files:
#~ $ for ix in {-h,-b} ; do du $ix *t.dat | column; done
#~ 20M  chan1t.dat      20M     chan2t.dat
#~ 20485631     chan1t.dat      20489007        chan2t.dat
```

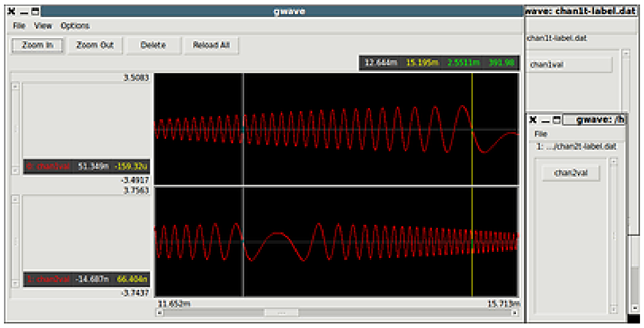
Python will be quite a bit slower; still, I like the output better, so I'm going to use those files here. Let's not forget to add the labels in first row once the 2-column files are generated (more waiting!):

```
echo "$(echo "t chan1val"; cat chan1t.dat)" > chan1t-label.dat
echo "$(echo "t chan2val"; cat chan2t.dat)" > chan2t-label.dat
```

Finally, we can call gwave `chan1t-label.dat` and it will be loaded; then we have to drag the '`chan1val`' button from the window to the plot, and we'll finally get a plot; then the second channel can be loaded:



While the zoom action requires that the buttons are clicked - at least the zooms are synchronised, and they don't take that long; and the scrollbar controls both subplots simultaneously. Also there are two cursors - the first (white) is placed by left mouse click, the second (yellow) is placed by middle mouse click, both are draggable (but remember to hold the correct mouse button!), and their difference in time domain is automatically plotted:



And this is it, in respect to gwave facilities; there don't seem to be ways to use scripts to analyse data, or add other kinds of labels.

gtkwave

In Ubuntu, `sudo apt-get install gtkwave` should get you sorted for this step.

GTKWave is another "wave viewer", however, there is something significantly different about it from `gwave` - namely, `gtkwave` is primarily intended to visualise *digital logic* signals (such as those produced by VHDL or Verilog simulation output). As such, `gtkwave` cannot by default work with our analog ASCII data - in spite of the homepage having a screenshot featuring analog waveforms.

References to such analog waveforms in `gtkwave` can be found in, say, "Re: [Ghdl-discuss] analog signals : ghw versus VCD" - and such discussion point that the analog waveform displayed **has** to be, in fact, defined in the digital domain. In other words, in HDL (hardware description languages) often a smallest primitive is `wire` or `signal`, that can carry a single bit information (1 or 0). However, "buses" - or *parallel* groups - of `signals` or `wires` can also be defined together, and as such, a given 'bus' will be able to carry integer information, in a range defined by the number of `wire/signals` contained in it. It is these kind of buses that can be used for, say, counter circuit outputs - and it is their output that can be visualised in `gtkwave` as an analog waveform. Confirmation for this can also be found in:

- help needed in sine generation of vhdl code.
- IP Core Cordic sine generator

Thus, our only hope for our ASCII text data, is to somehow convert the real (floating point) values to integer values - which implies, simply, multiplication with a large enough number, if the number of decimals are limited. Then, finding the largest integer would specify how many bits are required to represent it - and that will determine the number of bits in the 'bus', that we will have to define in the digital domain, in order to represent our analog signal.

`gtkwave` (and seemingly other waveform viewers oriented primarily to the digital domain) can work with the `.vcd` (Value change dump) file format, which is open and text only. Converters have been written for this format (see "la2vcd: Logic Analyzer to VCD File Converter"), but apparently not for ASCII data. One of the major problems could be that the binary representation of a bus value **has** to include all bits, as in `b00000001`; however note:

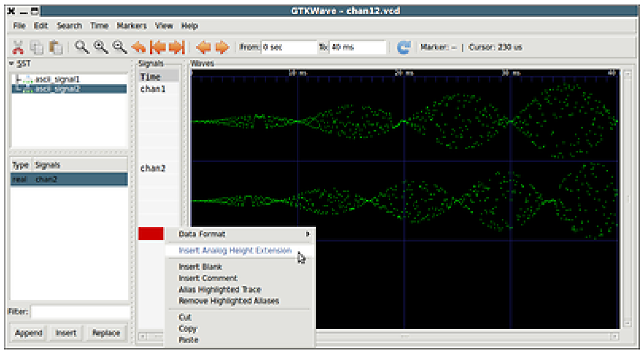
> Value Change Dump VCD
>
> ```
> Value Changes
> The value change section is a listing of variables with their value. If the variable is a scalar (1 bit) then the value change format
> ```

which means that, in principle, our real data should still be usable. However, the `vcd` format also needs a timestamp before each value change, so the best approach may be to write a C program.

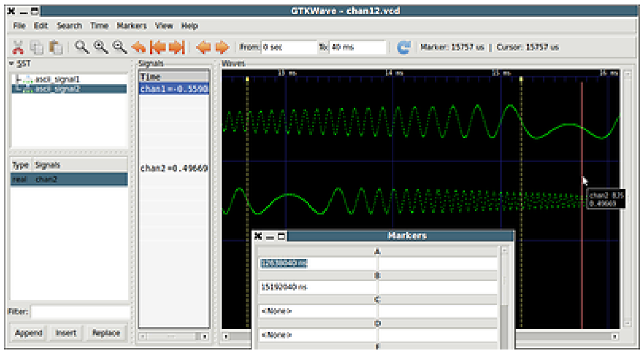
For this, I put together a small program called asc2vcd.c; which can help us convert our ASCII data files in the following manner using the command line:

```
# convert each ASCII data file to VCD,
# make sure they have unique marks
./asc2vcd chan1.dat 40ns chan1 ! ascii_signal1 > chan1.vcd
./asc2vcd chan2.dat 40ns chan2 % ascii_signal2 > chan2.vcd

# concatenate individual .vcd files one after another,
# gtkwave will be able to read both signals from this,
#  as long as marks are different...
cat chan1.vcd chan2.vcd > chan12.vcd

# open concatenated file in gtkwave
gtkwave chan12.vcd

#~ large files:
#~ $ for ix in {-h,-b} ; do du $ix *.vcd | column; done
#~ 37M  chan12.vcd      19M     chan1.vcd       19M     chan2.vcd
#~ 38752750     chan12.vcd      19374687        chan1.vcd       19378063        chan2.vcd
```

Once `gtkwave` has opened the files, click on "`ascii_signal1`" in the "SST" window; from below, drag the signal "`chan1`" and drop it in the second, "Time" column; a "`chan1`" label will appear in it. Right-click this label, and choose "Data Format" / "Analog" / "Step", and start clicking on "Zoom Out" until you can see something (*starts happening when approx 1ms is in range*). Then, again right-click on "`chan1`" label in "Time" column, and choose "Insert Analog Height Extension" - repeat this 3 to 5 times, until you get a satisfactory display. Then, repeat the right-click again and choose "Data Format" / "Analog" / "Resizing" / "All Data". Finally, repeat the same for "`ascii_signal2`" and "`chan2`"; and zoom out until both are in focus. The final result should be something like this:



Additionally, Alt+mousewheel zooms in and out, just mousewheel pans data, and Ctrl+mousewheel does sort of a fine scroll. Right-click drag is also a way to zoom in a region; left-click places a cursor. Instead of a second cursor, you can use "Markers" / "Drop named marker" to add a marker at the current cursor location, while to delete all markers, use "Markers" / "Collect all named markers". Choose "View/Show Mouseover" for an info box while clicking; and use "Markers" / "Show/change marker data" to raise a window related to 'Markers', as shown below:



In all, zoom is quite fast for our data (maybe even too fast ☻), and channels can be easily compared. Finally, once your both channels are loaded in `gtkwave`, you may want to re-export the `.vcd` file again (this should help with proper vcd format, and also decrease the size, of the original `chan12.vcd`- make sure you choose a different filename if re-exporting).

Audacity

In Ubuntu, `sudo apt-get install audacity` should get you sorted for this step.

Audaci... what??! Isn't this an audio program? Well, yes it is - which is why it has a pretty nice and fast waveform plotting engine ☻

Before we can use `audacity`, we should note that by default, it can only import binary data - so our ASCII data will, again, need conversion. However, that is again not too difficult - since `audacity` supports import of 'raw' data; which may be just a binary stream of, say, floating point numbers.

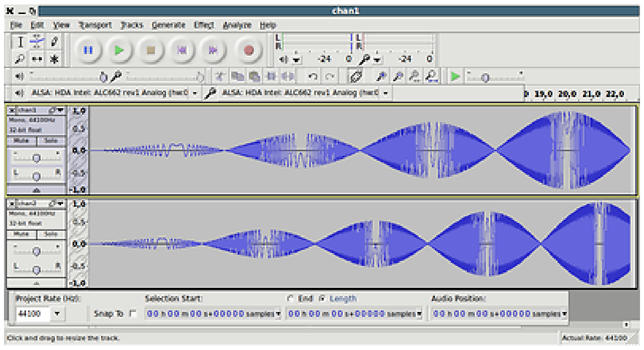
- Furthermore, it is important to note that audio software typically deals with floating point values from -1 to 1; signals exceding this range are typically clipped, although `audacity` can display up to -2 to 2 range. As our signal here fits in the range [-4:4], that means we'd have to scale it (multiply) by 1/4 to have it fit in Audacity's default [-1:1] range.

- Note also that `audacity` can be set to sampling rates of up to 100KHz; we can not set it to our desired sampling rate of 25 MHz. Therefore, the timebase will be necesarilly wrong for our type of signal.

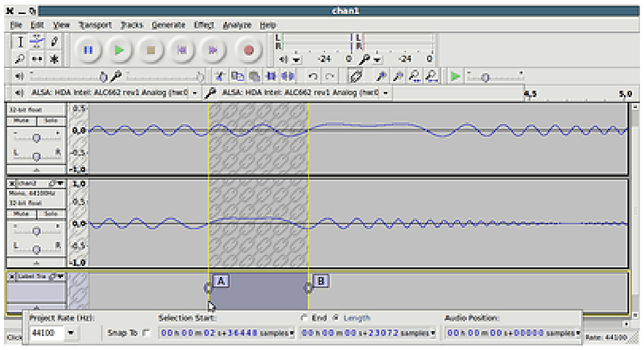
For this, I put together a small program called asc2raw.c; which can help us convert our ASCII data files to a raw binary stream of floats, in the following manner using the command line:

```
# convert each ASCII data file to raw stream of floats
# scalefactor is 1/4 = 0.25
LC_ALL=C ./asc2raw chan1.dat 0.25 > chan1.raw
LC_ALL=C ./asc2raw chan2.dat 0.25 > chan2.raw

#~ files are not that big now:
#~ $ for ix in {-h,-b} ; do du $ix *.raw | column; done
#~ 3,9M chan1.raw       3,9M    chan2.raw
#~ 4000004      chan1.raw       4000004 chan2.raw
```

Then, open `audacity`, and choose "File" / "Import" / "Raw Data..." and choose, say, `chan1.raw`. In the subsequent "Import Raw Data" dialog, `audacity`will automatically recognize that the file carries 32-bit float, little-endian data; leave the 'Mono' setting, and you may just as well leave the sampling rate at the default 44100 Hz - it will be wrong in terms of what our signal should represent; but it will allow for soundcard reproduction, and so with this we'll be able to **sonify** our signal (*well, at least in principle - our signal here is way too bass-y to be reproduced on netbook speakers, for instance*). You can then repeat the same procedure for `chan2.raw`.



Ctrl+wheel zooms in/out; Shift+wheel pans the signal in `audacity`. There is one cursor, and at its location, labels can be placed (in a separate "label" track) using "Tracks" / "Add label at selection"; then making a selection in the "label" track snaps to the labels, and extends accross tracks.

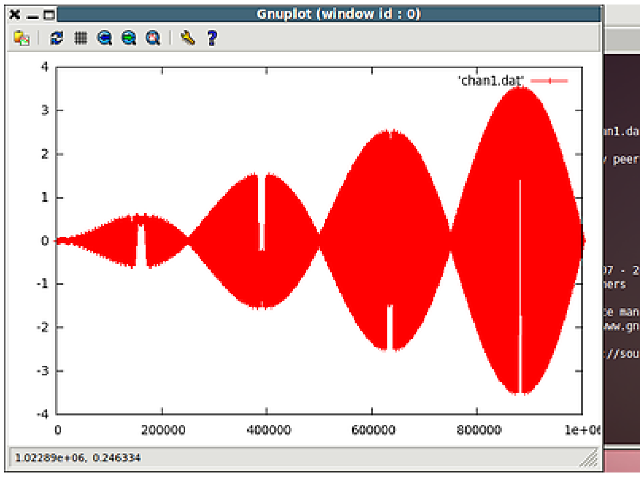


Gnuplot

Another known software, `gnuplot` is mentioned here simply because it offers some level of interaction with its plot - such as right click zooms. For instance, we can do something like this in a terminal shell:

```
$ gnuplot

        G N U P L O T
        Version 4.2 patchlevel 6
...
...
Terminal type set to 'wxt'
gnuplot> plot 'chan1.dat' with linespoints
```

This will raise a `gnuplot` window - however, it is slowwwwww for reading...



The zoom is also extremely slow for data of this size, and can lead to a complete freeze of the application...

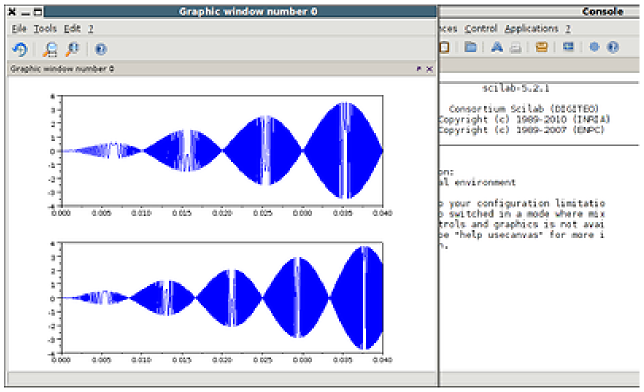
Scilab

`scilab` can be used to generate the ASCII data - and it can be also used to view it. Consider the following command sequence:

```
// readdata.sci
// increase stacksize - else scilab may run out of memory
stacksize max

// read the saved ASCII data
// -1,1 describes file format: rows=-1, columns 1
// "...=-1 if you do not know the numbers of rows"
```

```
chan1=read("chan1.dat",-1,1);
chan2=read("chan2.dat",-1,1);

size(chan1) // should be "1000001.    1."

// recreate time array
t=(1/25e6)*[0:1:1000000]';

// display - 'plot' is Matlab oriented command,
//  can also use plot2d, plot2d2, ... which are possibly a tad quicker?

// in matrix of rowXcolumn of 2X1, select subplot 1
subplot(211)

// plot chan1
plot(t, chan1) // default blue
//plot2d(t, chan1) // default black

// in matrix of rowXcolumn of 2X1, select subplot 2
subplot(212)

// plot chan2
plot(t, chan2) // default blue
//plot2d(t, chan2) // default black
```

Executing these commands, should result with a plot like:



```
         scilab-5.2.1

  Consortium Scilab (DIGITEO)
Copyright (c) 1989-2010 (INRIA)
Copyright (c) 1989-2007 (ENPC)

on:
al environment

o your configuration limitatio
o switched in a mode where mix
trols and graphics is not avai
se "help usecanvas" for more i
n.
```

While the speed is not all that bad in this case - the axes are not synchronised (so zoom in one subplot doesn't zoom the corresponding region in the other subplot); it is not necesarilly easy to navigate using the given zoom options; and there don't seem to be options to add annotations or cursors interactively (*though it is possible similar options are available from code*).

- However, note that with a dataset like this, calling something like '`plot(t,chan1,'o')`' in `scilab`, would append a *marker* (a small circle) to each and every datapoint - and so, when zoomed out to view entire waveform, `scilab` will also attempt to render 2 million additional points, and freeze!
- Note also that **Xcos** (*old `scicos`, the graphical dynamics environment of `scilab`*) uses the same graphics engine as `scilab` to display its data.

Python - matplotlib

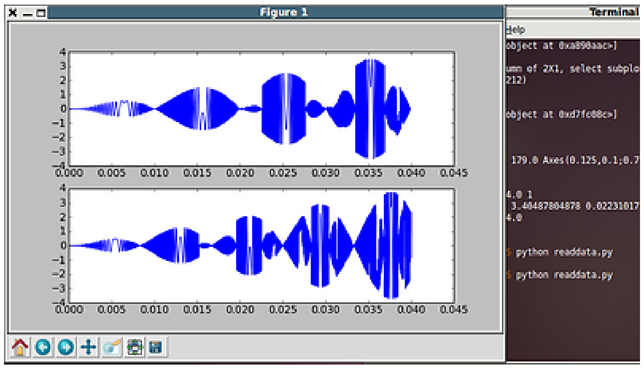
In Ubuntu, `sudo apt-get install python-matplotlib` (contains pylab) should get you sorted for this step.

As `python` can be used to generate the ASCII data - it can be also used to view it. Consider the following command sequence:

```
# readdata.py
import numpy
from matplotlib.pyplot import figure, show

# read the saved ASCII data
chan1=numpy.loadtxt("chan1.dat")
chan2=numpy.loadtxt("chan2.dat")

tsz=len(chan1)
print tsz # should be "1000001"

# recreate time array
t=(1/25e6)*numpy.arange(0,tsz,1)

# display
# create figure
fig = figure(1,figsize=(8,5))

# in matrix of rowXcolumn of 2X1, select subplot 1
ax1 = fig.add_subplot(211)
# plot chan1
ax1.plot(t,chan1)
```

```
# in matrix of rowXcolumn of 2X1, select subplot 2
ax2 = fig.add_subplot(212)
# plot chan1
ax2.plot(t,chan2)

# show figure
fig.show() # for python shell
show() # for script
```

Executing these commands, should result with a plot like:



```
object at 0xa890aac>]
umn of 2X1, select subplo
212)
object at 0xd7fc08c>]
 179.0 Axes(0.125,0.1;0.7
4.0 1
 3.40487804878 0.02231017
4.0
$ python readdata.py
$ python readdata.py
```

There is a default navigation toolbar, that allows panning and zooming; however, it is kind of sluggish, and the two axes are not synchronized. Apart from this, there are no direct ways for labeling, such as cursors - however, those facilities are accessible through script using `python` with `matplotlib`, as will be shown further on in mwfview.

Python - chaco

In Ubuntu, `sudo apt-get install python-chaco` should get you sorted for this step.

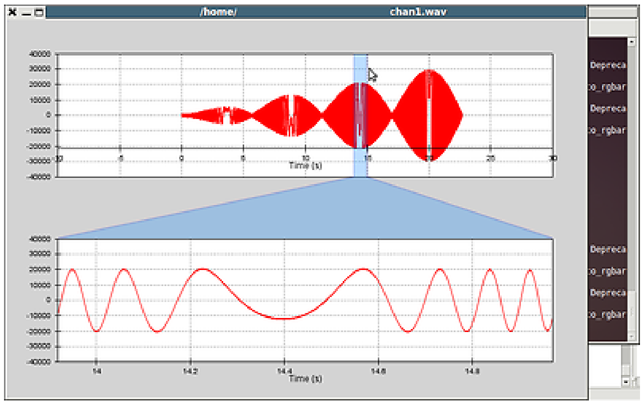
Chaco is a "*Python plotting application toolkit that facilitates writing plotting applications at all levels of complexity*", which is interesting because "*...it supports fast vector graphics rendering for interactive data analysis (read: **fast live updating plots**) and custom plot construction...*" ('NumericAndScientific/Plotting - PythonInfo Wiki').

I haven't had much time to dive into `chaco`, however, there is an interesting example in Chaco :: Enthought, Inc. - Screen Shots (Gallery), called "Zooming Plot Inspector" (zoom_plot.py).

This example, `zoom_plot.py` is meant to visualise audio `.wav` files - so you can use the audacity procedure in order to export the `raw` version of our data to a `.wav` file. In the `audacity` "Export settings" choose "WAV ... signed 16 bit PCM", and save, say, as `chan1.wav`. Then, in terminal, execute:

```
# zoom_plot.py is installed as part of python-chaco
# if copied in whatever directory, it may not find its libraries
# so execute it from its default location
cd /usr/share/doc/python-chaco/examples/zoomed_plot

# change line "data = numpy.fromstring(s, numpy.dtype('int8'))"
# to "numpy.dtype('int16'))" in /usr/share/doc/python-chaco/examples/zoomed_plot/wav_to_numeric.py
sudo nano wav_to_numeric.py

# change 'fname' line to 'fname = "/path/to/chan1.wav"'
# in /usr/share/doc/python-chaco/examples/zoomed_plot/zoom_plot.py
sudo nano zoom_plot.py

# execute zoom_plot.py - chan1.wat should be loaded
python zoom_plot.py
```

Executing these commands, should result with a plot like:

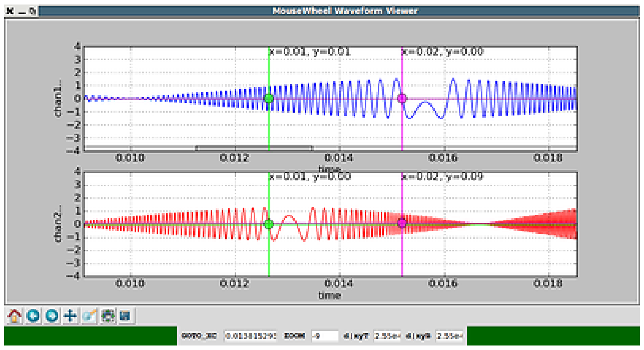


While the zooming widget is certainly cool, lack of possibility to define timebase, and wrong render of y-axis values, limit the use of `zoom_plot.py`; additionally, for a dataset of this size, the rendering of the plot doesn't seem all *that* much faster than `matplotlib`.

Python - mwfview.py

Well, as I was originally frustrated with lack of mouse-wheel zooming capabilities in viewers, I thought I'd look at `matplotlib` a bit closer, and maybe add some mouse-wheel events for zoom. The result is, what I'd call "MouseWheel Waveform Viewer" (creative, no?), or mwfview.py.

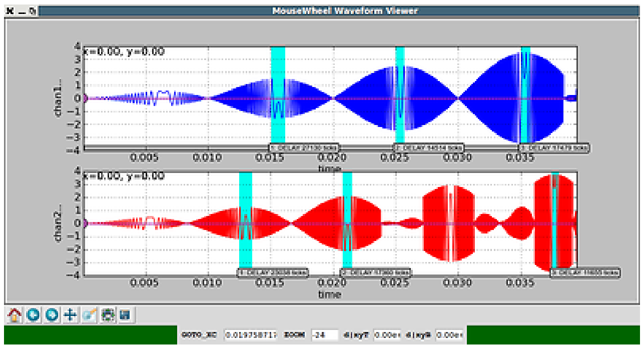
The data filenames are written inside the script (as well as any other settings), and the script is ran by `python mwfview.py`. There's two cursors, mousewheel zoom and textfield navigation:



.. and all that, at not much shabbier display speed than default `matplotlib`. In addition, if you uncomment the following lines:

```
# analysis
abmT = AnalysisTransitions(ax_sub1, t, chan1)
abmB = AnalysisTransitions(ax_sub2, t, chan2)
```

... then, an analysis will be performed, where positive transitions over 1 (1 is chosen simply to limit the number of results) are identified; and those with delays of particular duration are marked - as on the image below (see the script source for details):



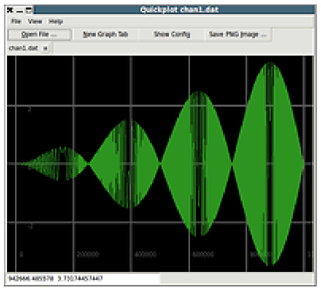
Quickplot

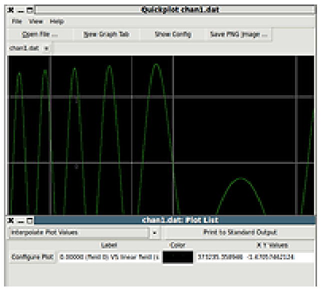
In Ubuntu, `sudo apt-get install quickplot` should get you sorted for this step.

Quickplot (found through this page): "*The difference between this 2D plotter and most 2D plotters is that the primary purpose of Quickplot is to help you quickly interact with your data*".

Note that `quickplot` data parsing depends on the locale (type `locale` in your terminal to see your current settings); in particular failure occurs if the decimal point separator character of the locale (say ',') is different from the one in the data (say '.'). The following command should ensure that period '.' is the decimal point separator (as per the 'C' locale) - and that the minimum memory is used for plotting:

```
LC_ALL=C quickplot chan1.dat -L --no-lines --point-size=0
```



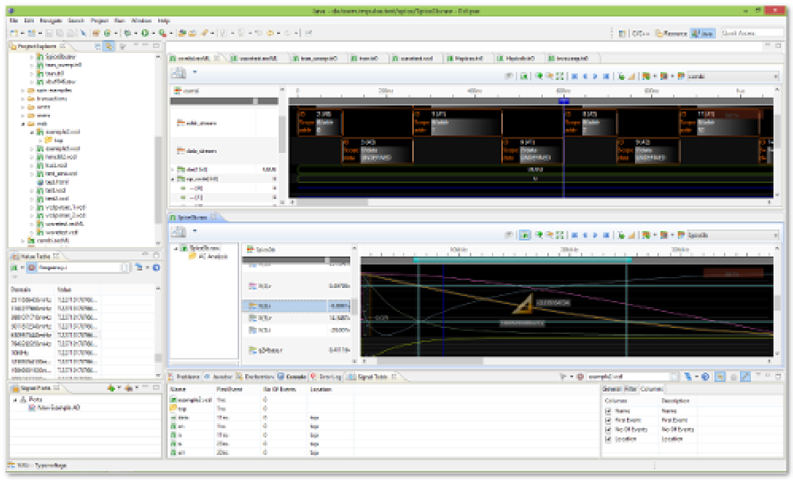


However, it still takes a while to load the file - and, unfortunately, every time a portion of the window needs to be redrawn (*if it had been covered by another window*), causes long and tiring redraws, with data of this size (however, zooming in itself doesn't cause these delays). It also seems there is a single cursor, whose setting raises an info window.

Impulse

Impulse is a plugin for the eclipse IDE. Its quite useful if eclipse is used for simulation development. Impulse has similar roots like gtkWave (VHDL, Verilog amd embedded systems), but has a good support for analog data like xy cusors, auto-scaling, log10 axes, ... It supports logic data, analog, text, logs, transactions and events. Beside waveforms, data can be displayed in tables (Value Table). Data can be read from files (spice, tabular data,..) or from pipes, sockets and process streams. Multiple inputs can be combined.

In Ubuntu, `sudo apt-get install eclipse` installs the ide. After you started eclipse , go to "Install new Software" and use `http://update.toem.de` as update site. More informations at toem.



Honorable mentions

- Crispy Plotter - Windows only open-source, installs fine under wine, looks very good interaction wise - but doesn't seem to be able to read waveforms..
- WaveMan - "*Downloading - Coming soon. The program is not yet ready for release.*" Screen Shots
- Plotmtv - "*The application plotmtv is a fast multi-purpose plotting program for visualization of scientific data in an X11-window environment. Each plot comes with a simple but functional Graphical User Interface which allows users to zoom in or pan to areas of interest on the plot or to toggle between 2D and 3D plots... The program reads in data in the MTVDAT format and plots each dataset in the data-file in turn.*" Seems old, packages in Ubuntu found from dapper to jaunty.
- Via "Puppy Linux Discussion Forum: View topic - Oscilloscope, strip chart or other waveform viewer? (solved)":
    - baudline signal analyzer - FFT spectrogram ; dual license, costly source for purchase, else free binary download; *awesome* spectrogram visualisation from live soundcard input; there should be waveform loading facilities as well
    - xoscope for Linux: "*xoscope for Linux is a digital oscilloscope that uses a sound card (via /dev/dsp or EsounD) and/or Radio Shack ProbeScope as the signal input.*"; `sudo apt-get install xoscope`; you can store and recall channel snippets, but no dedicated facilities to import and view data; seems old
    - Osqoop (EIG: Osqoop (l'oscilloscope libre)) - not in `apt`, but "*Ubuntu packages are provided for Karmik and Lucid*" (Osqoop - Home - Open wiki) in ppa (though the lucid one "Failed to build"; use the karmic one); it captures from soundcard - you can then "freeze" the "display", and "export visible data" as ASCII file - however there are no facilities to import data

Conclusion

As we can see, there are many tools, intended for different purposes, that can be used for waveform viewing. My personal favorite, though, has become `python`'s `matplotlib`, since it allows for scripting of both user interaction with the graph, and for analysis of signals (*as discussed in mwfview above*) - while still keeping a relatively decent graphic rendering performance, even for two channels of a million points each.

Additional

- chipforge.org: Free Tools for free Hardware
- Sound & MIDI Software For Linux - DSP Software
- analog waveform plotter for gnucap – an alternative to gwave « Geek Went Freak!