There's no such thing as a simple GUI

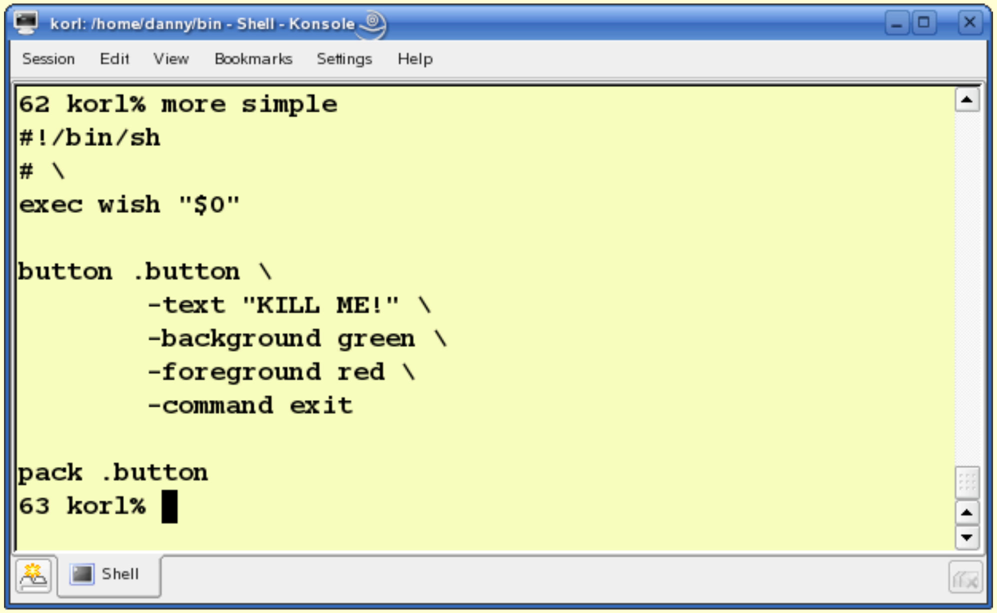
- A simple GUI
- Why do you need GUI's in the first place?
- Fundamental knowledge
- GUI design strategy
- Example
- Pitfalls

Danny Harvey
President
Boulder Real Time Technologies, Inc.

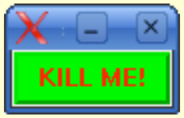
BRTT

June, 2006

A Simple GUI



```
62 korl% more simple
#!/bin/sh
# \
exec wish "$0"

button .button \
        -text "KILL ME!" \
        -background green \
        -foreground red \
        -command exit

pack .button
63 korl% 
```



Why do you need GUIs in the first place?

- Think HARD before you go down this road
- GUI programming should be considered as a task that requires advanced programming knowledge and skills
- Do you know why you need a proposed GUI?
- Have you considered less exotic alternatives?
- GOOD reasons for GUIs:
  - To show complex information that lends itself to graphical displays
  - To provide intuitive and highly choreographed user inputs – note that GUIs tend to constrain user interactions
- BAD reasons for GUIs:
  - To alleviate the user from typing (the developer will certainly be typing a lot more)
  - A vain attempt to put a more “sophisticated” or “sexier” front-end on some function that doesn’t really need it – note that a well designed generalized type-based user interface is usually going to provide much more function and flexibility than most GUI interfaces

Fundamental Knowledge

- GUI programming requires lots of knowledge regardless of Antelope; you will not be successful with this unless you do your homework
- The Antelope tools are no replacement for the fundamental knowledge needed by any GUI programmer
- The Antelope tools make it easy for an already experienced GUI programmer to interface with the various Antelope objects
- What "knowledge" does a GUI programmer need?
  - Good knowledge of the underlying programming or scripting language
  - A journeyman's understanding of the X-windows system; e.g. the X-server, the relationship between clients and the X-server, the event driven nature of X-windows interactions, fonts, colors, images, the various graphics primitives, scaling
  - Good knowledge of the particular widget package that is to be used. In Antelope we mainly use `tk`-based widgets, either in `tcl` or `perl`.

GUI Design Strategy

- Clearly define the problem you are trying to solve – it may surprise you to find that this step may either eliminate the development task entirely or point it in a direction that does not require a GUI
- "storyboard" the GUI – make drawings of what it should look like and *exactly* how the user would interact (i.e. what particular widgets will be used, how information will be displayed, process flow, etc.)
- Try to dissect the overall problem into three logical parts; 1) user GUI front-end, 2) internal data engine and 3) a graphical display back-end that will show whatever information you need to show
- Don't try to do the whole problem in one monolithic chunk; it is fine to be running separate programs and scripts that talk to each other in some fashion – this also helps in prototyping and debugging
- Start off with the bare minimum GUI functionality – you can always add more walking menus and dialogs later if you really need them
- Be patient – don't expect to come up with your "final" solution quickly; GUIs tend to be perpetual and incremental works-in-progress – accept that fact and you will be a happier person

GUI Design Strategy

- Choose your language for the GUI and display parts
  - I will skip over **`C`**, **`C++`** and **`java`**
  - Antelope contains fairly standard **`perl/tk`** extensions; this provides the highest performance scripting approach with the sophistication of the **`perl`** language. Downsides are lack of Antelope **`perl/tk`** graphical extensons and **`perl`**’s hyper-paranoid security limitations on normal **`tk`** IPC.
  - Antelope contains fairly standard **`tcl/tk`** extensions plus special Antelope graphics extensions, like **`brttplot`**; this provides the highest graphics functionality scripting approach with the simplicity of the **`tcl`** language and ease of fully duplex **`tk`**-based IPC. Downsides are potential performance problems and limitations in overall complexity due to simplistic nature of **`tcl`** language.

GUI Design Strategy

- Figure out how to glue the major pieces together
  - Internal data engine can be Datascope, ORB or standalone analysis programs, like **`dbwfmeas`**
  - IPC can be implemented through combinations of database manipulations, external parameter files, command line arguments and the use of **`tksend`** to pass messages between processes
  - A good approach is to modularize design by using small stand-alone display libraries and scripts

Example - `dbnoise`

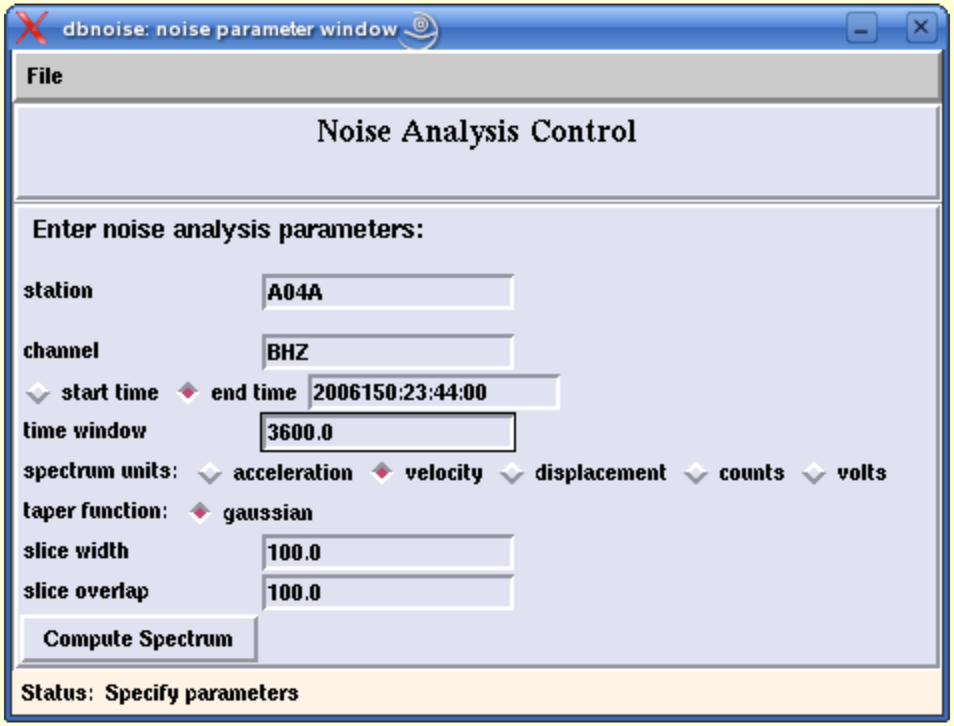
- Written as a **`tcl/tk`** script to provide a GUI to the **`dbwfmeas`** program for specifying parameters for noise spectra computations, execute the computations and display the resulting spectra – we consider this to be a simple GUI (420 lines of code)
- **`dbwfmeas`** is designed as a graphic-less high performance computation engine that reads data from a database, computes stuff and puts its computations into a database; all of the computational parameters are specified through a normal Antelope parameter file

Example - `dbnoise`

- Start by going through the process manually of setting up **`dbwfmeas`** to compute spectra; consult **`dbwfmeas`** man page, find some example data to work with and do what is needed to compute spectra
- Determine exactly what parameters need to be specified by the user; from this come up with a front-end “storyboard” for the input GUI:

Station code	entry widget
Channel code	entry widget
Start/end time	entry widget
Computation time window	entry widget
Output spectrum units	radiobutton widgets
Taper function	radiobutton widgets
Time slice values	entry widgets
Execute	button widget

Example - **dbnoise**



Noise Analysis Control

Enter noise analysis parameters:

station A04A

channel BHZ

○ start time ● end time 2006150:23:44:00

time window 3600.0

spectrum units: ○ acceleration ● velocity ○ displacement ○ counts ○ volts

taper function: ● gaussian

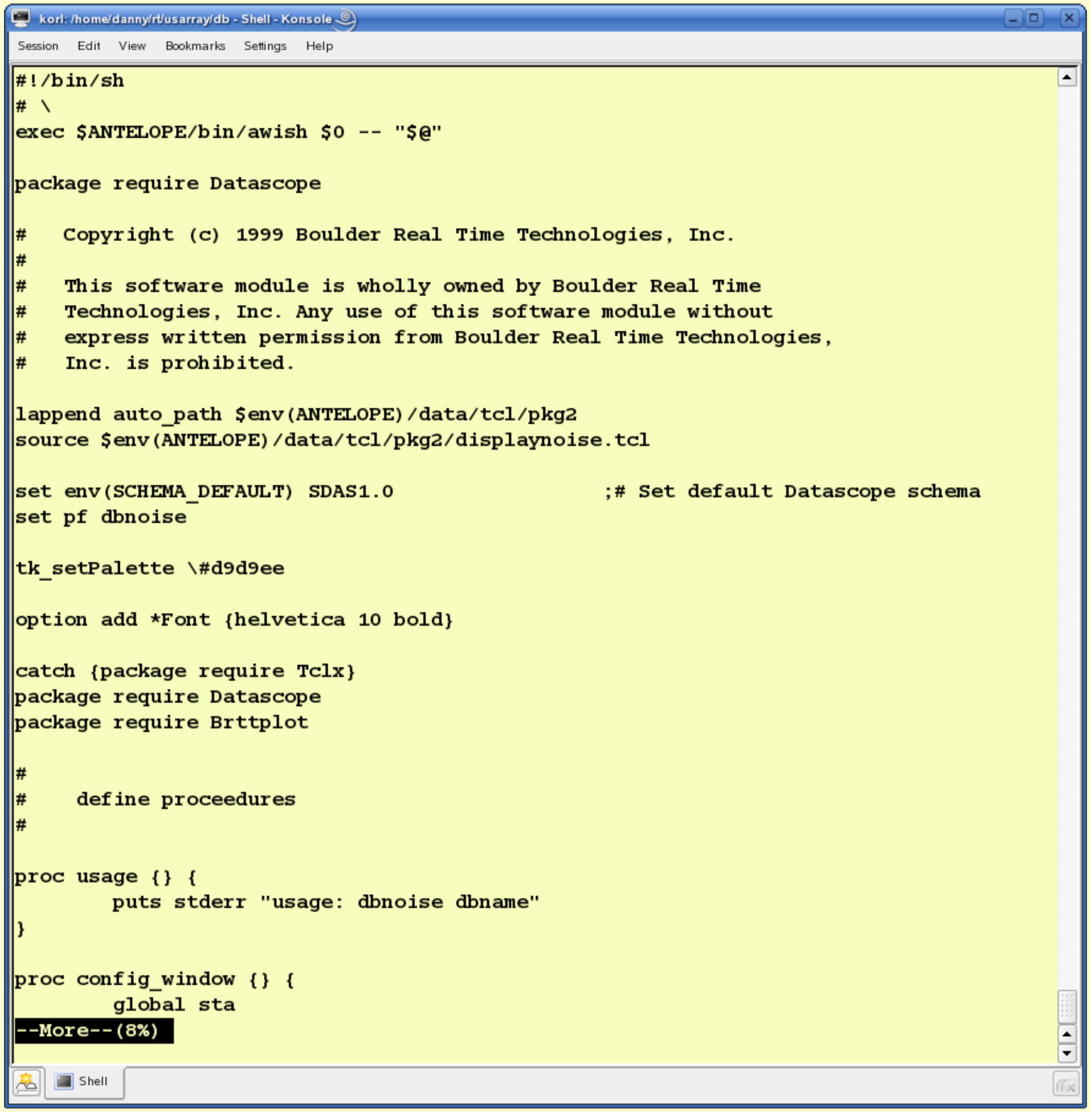
slice width 100.0

slice overlap 100.0

Status: Specify parameters

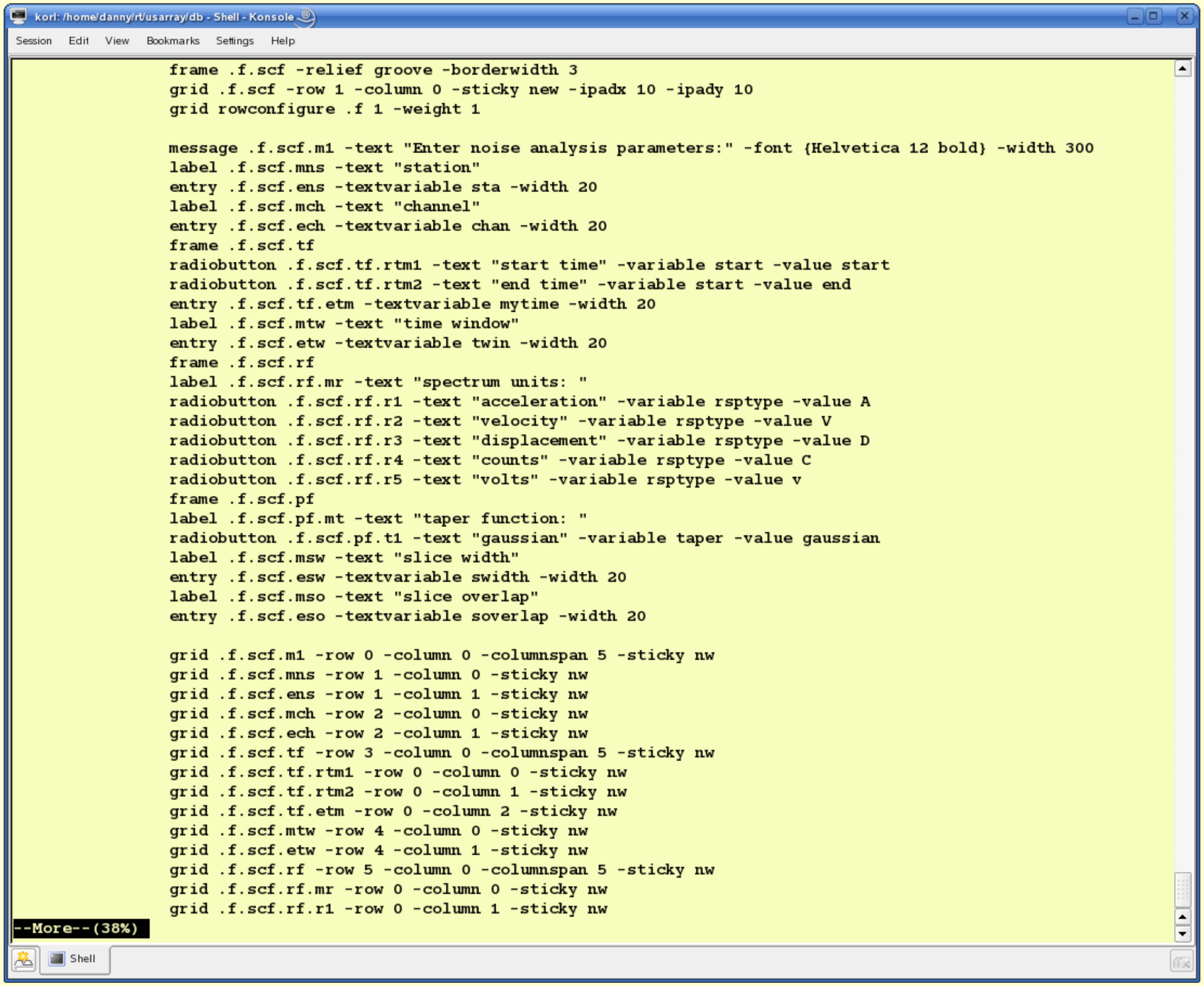
- Write a skeleton **`tcl/tk`** script that makes the GUI without actually doing anything – iterate until it looks right and includes the right information

Example - **dbnoise**



```
#!/bin/sh
# \
exec $ANTELOPE/bin/awish $0 -- "$@"

package require Datascope

#   Copyright (c) 1999 Boulder Real Time Technologies, Inc.
#
#   This software module is wholly owned by Boulder Real Time
#   Technologies, Inc. Any use of this software module without
#   express written permission from Boulder Real Time Technologies,
#   Inc. is prohibited.

lappend auto_path $env(ANTELOPE)/data/tcl/pkg2
source $env(ANTELOPE)/data/tcl/pkg2/displaynoise.tcl

set env(SCHEMA_DEFAULT) SDAS1.0                 ;# Set default Datascope schema
set pf dbnoise

tk_setPalette \#d9d9ee

option add *Font {helvetica 10 bold}

catch {package require Tclx}
package require Datascope
package require Brttplot

#
#    define proceedures
#

proc usage {} {
        puts stderr "usage: dbnoise dbname"
}

proc config_window {} {
        global sta
--More--(8%)
```

Example - **dbnoise**

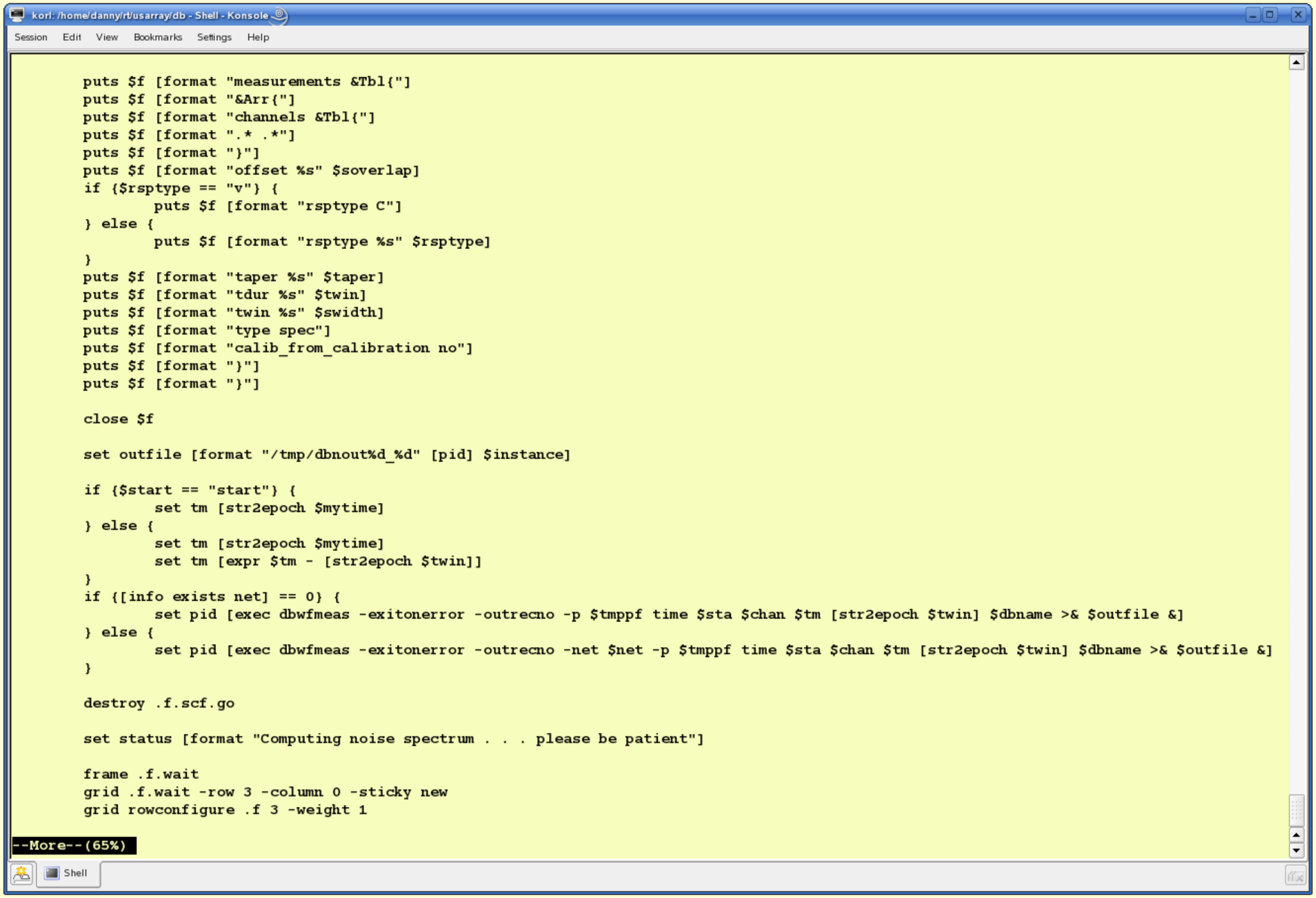


```
                frame .f.scf -relief groove -borderwidth 3
                grid .f.scf -row 1 -column 0 -sticky new -ipadx 10 -ipady 10
                grid rowconfigure .f 1 -weight 1

                message .f.scf.m1 -text "Enter noise analysis parameters:" -font {Helvetica 12 bold} -width 300
                label .f.scf.mns -text "station"
                entry .f.scf.ens -textvariable sta -width 20
                label .f.scf.mch -text "channel"
                entry .f.scf.ech -textvariable chan -width 20
                frame .f.scf.tf
                radiobutton .f.scf.tf.rtm1 -text "start time" -variable start -value start
                radiobutton .f.scf.tf.rtm2 -text "end time" -variable start -value end
                entry .f.scf.tf.etm -textvariable mytime -width 20
                label .f.scf.mtw -text "time window"
                entry .f.scf.etw -textvariable twin -width 20
                frame .f.scf.rf
                label .f.scf.rf.mr -text "spectrum units: "
                radiobutton .f.scf.rf.r1 -text "acceleration" -variable rsptype -value A
                radiobutton .f.scf.rf.r2 -text "velocity" -variable rsptype -value V
                radiobutton .f.scf.rf.r3 -text "displacement" -variable rsptype -value D
                radiobutton .f.scf.rf.r4 -text "counts" -variable rsptype -value C
                radiobutton .f.scf.rf.r5 -text "volts" -variable rsptype -value v
                frame .f.scf.pf
                label .f.scf.pf.mt -text "taper function: "
                radiobutton .f.scf.pf.t1 -text "gaussian" -variable taper -value gaussian
                label .f.scf.msw -text "slice width"
                entry .f.scf.esw -textvariable swidth -width 20
                label .f.scf.mso -text "slice overlap"
                entry .f.scf.eso -textvariable soverlap -width 20

                grid .f.scf.m1 -row 0 -column 0 -columnspan 5 -sticky nw
                grid .f.scf.mns -row 1 -column 0 -sticky nw
                grid .f.scf.ens -row 1 -column 1 -sticky nw
                grid .f.scf.mch -row 2 -column 0 -sticky nw
                grid .f.scf.ech -row 2 -column 1 -sticky nw
                grid .f.scf.tf -row 3 -column 0 -columnspan 5 -sticky nw
                grid .f.scf.tf.rtm1 -row 0 -column 0 -sticky nw
                grid .f.scf.tf.rtm2 -row 0 -column 1 -sticky nw
                grid .f.scf.tf.etm -row 0 -column 2 -sticky nw
                grid .f.scf.mtw -row 4 -column 0 -sticky nw
                grid .f.scf.etw -row 4 -column 1 -sticky nw
                grid .f.scf.rf -row 5 -column 0 -columnspan 5 -sticky nw
                grid .f.scf.rf.mr -row 0 -column 0 -sticky nw
                grid .f.scf.rf.r1 -row 0 -column 1 -sticky nw
--More--(38%)
```

Example - **dbnoise**

- Extend the **tcl/tk** script to perform the steps that you worked out when manually executing **dbwfmeas**:
   1. Build up a temporary parameter file in **/tmp**
   2. Execute **dbwfmeas** with the proper command line arguments being careful to capture standard and error output
   3. Monitor to see when **dbwfmeas** is finished and determine if it ran successfully
   4. If **dbwfmeas** encountered an error, display the error message
   5. If **dbwfmeas** ran successfully, display its results
   6. Clean up, i.e. get rid of temporary files

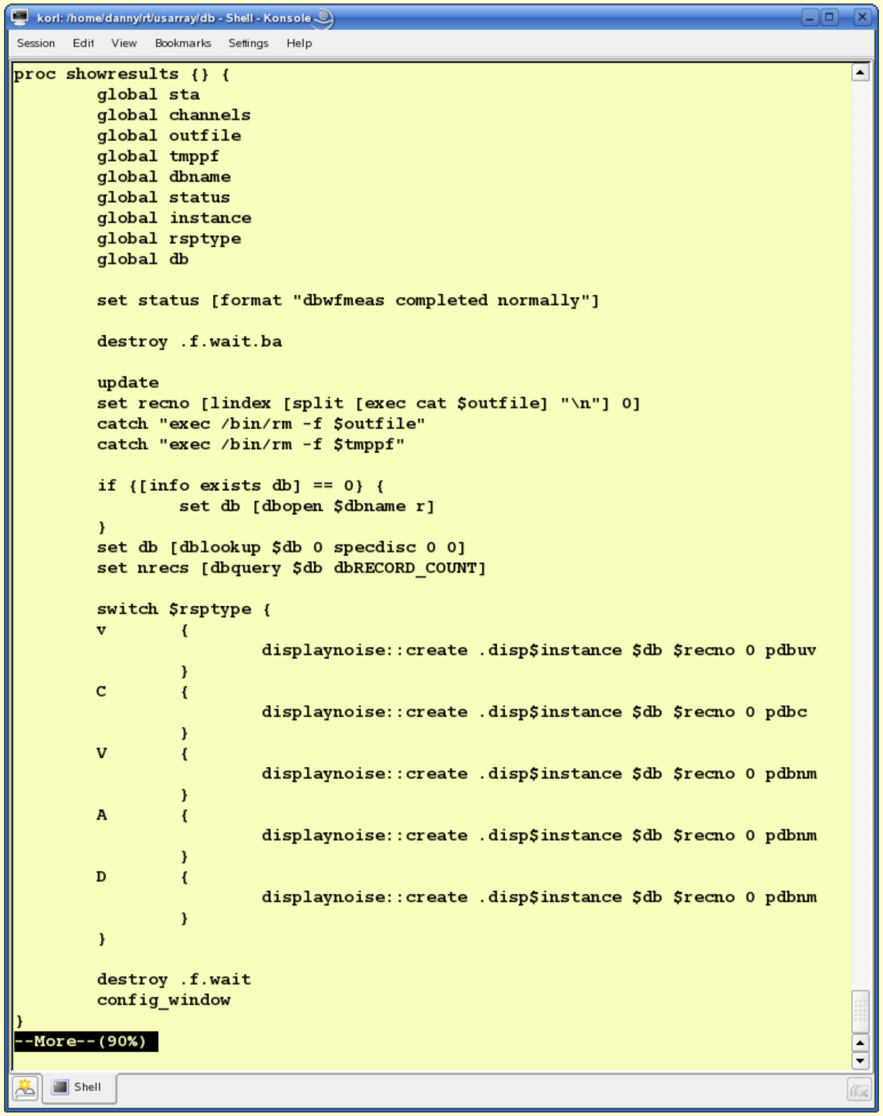
Example - **dbnoise**



```
        puts $f [format "measurements &Tbl{"]
        puts $f [format "&Arr{"]
        puts $f [format "channels &Tbl{"]
        puts $f [format ".* .*"]
        puts $f [format "}"]
        puts $f [format "offset %s" $soverlap]
        if {$rsptype == "v"} {
                puts $f [format "rsptype C"]
        } else {
                puts $f [format "rsptype %s" $rsptype]
        }
        puts $f [format "taper %s" $taper]
        puts $f [format "tdur %s" $twin]
        puts $f [format "twin %s" $swidth]
        puts $f [format "type spec"]
        puts $f [format "calib_from_calibration no"]
        puts $f [format "}"]
        puts $f [format "}"]

        close $f

        set outfile [format "/tmp/dbnout%d_%d" [pid] $instance]

        if {$start == "start"} {
                set tm [str2epoch $mytime]
        } else {
                set tm [str2epoch $mytime]
                set tm [expr $tm - [str2epoch $twin]]
        }
        if {[info exists net] == 0} {
                set pid [exec dbwfmeas -exitonerror -outrecno -p $tmppf time $sta $chan $tm [str2epoch $twin] $dbname >& $outfile &]
        } else {
                set pid [exec dbwfmeas -exitonerror -outrecno -net $net -p $tmppf time $sta $chan $tm [str2epoch $twin] $dbname >& $outfile &]
        }

        destroy .f.scf.go

        set status [format "Computing noise spectrum . . . please be patient"]

        frame .f.wait
        grid .f.wait -row 3 -column 0 -sticky new
        grid rowconfigure .f 3 -weight 1

--More--(65%)
```

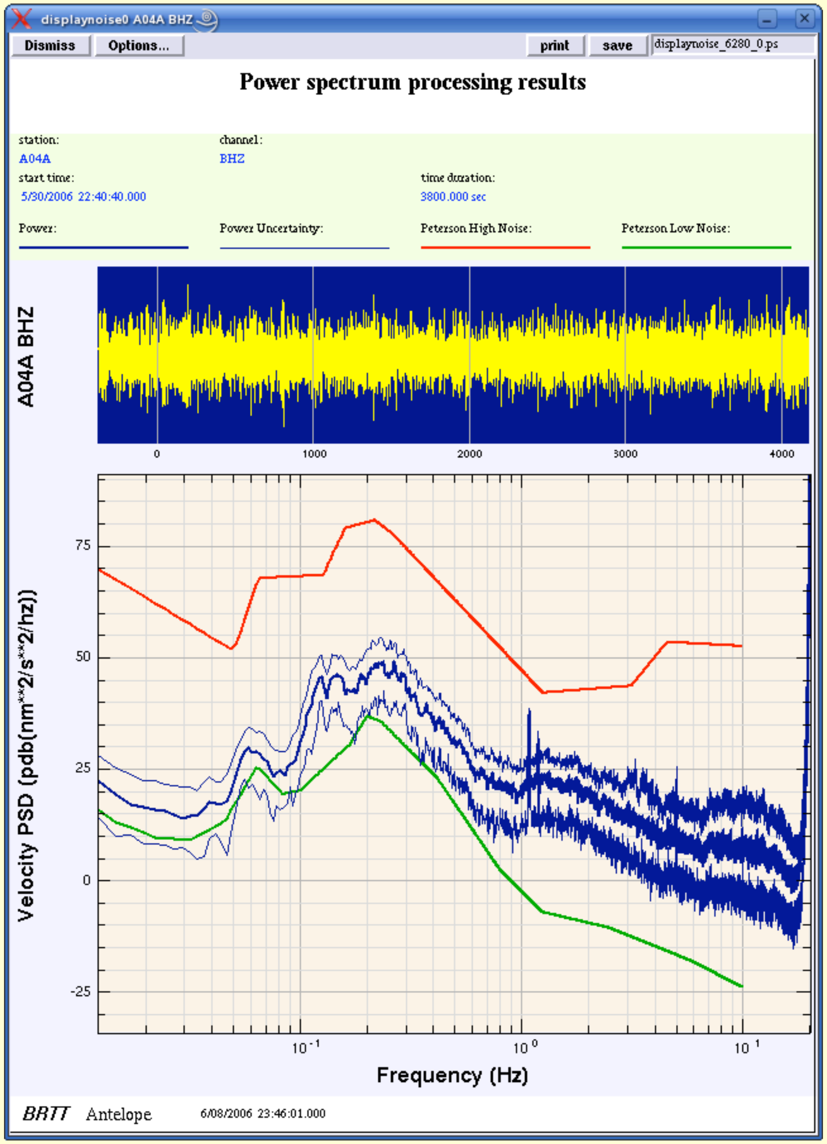
Example - **dbnoise**

- Display is another standalone **`tcl/tk`** script widget, named **`displaynoise`**, which will display the noise spectra as it is stored in an Antelope database



```
proc showresults {} {
        global sta
        global channels
        global outfile
        global tmppf
        global dbname
        global status
        global instance
        global rsptype
        global db

        set status [format "dbwfmeas completed normally"]

        destroy .f.wait.ba

        update
        set recno [lindex [split [exec cat $outfile] "\n"] 0]
        catch "exec /bin/rm -f $outfile"
        catch "exec /bin/rm -f $tmppf"

        if {[info exists db] == 0} {
                set db [dbopen $dbname r]
        }
        set db [dblookup $db 0 specdisc 0 0]
        set nrecs [dbquery $db dbRECORD_COUNT]

        switch $rsptype {
        v       {
                        displaynoise::create .disp$instance $db $recno 0 pdbuv
                }
        C       {
                        displaynoise::create .disp$instance $db $recno 0 pdbc
                }
        V       {
                        displaynoise::create .disp$instance $db $recno 0 pdbnm
                }
        A       {
                        displaynoise::create .disp$instance $db $recno 0 pdbnm
                }
        D       {
                        displaynoise::create .disp$instance $db $recno 0 pdbnm
                }
        }

        destroy .f.wait
        config_window
}
--More--(90%)
```

Example - **`dbnoise`**



displaynoise_6280_0.ps

Power spectrum processing results

station: A04A

channel: BHZ

start time: 5/30/2006 22:40:40.000

time duration: 3800.000 sec

Power:

Power Uncertainty:

Peterson High Noise:

Peterson Low Noise:

BRTT Antelope 6/08/2006 23:46:01.000

BRTT

June, 2006