SpecTcl Root ntuple plugin

Ron Fox

SpecTcl Root ntuple plugin

by Ron Fox

Revision History

Revision 1.0 February 5, 2008 Revised by: RF
Original Release

Table of Contents

List of Figures

Chapter 1. Introduction

Many users of SpecTcl have various ad-hoc methods to move NSCL event data into Root. The Root ntuple plugin hooks into the SpecTcl filter mechanism and supplies a generalized scheme for migrating pre-sorted data from any data that SpecTcl can analyze to Root ntuple files.

This plugin has the following pre-requisites:

1. Installation of SpecTcl-3.2 or later (no pre-releases of 3.2).
2. Installation of Root. The plugin has been tested with Root 5.14 but should work with almost any version of Root.

The remainder of this document:

- Describes how to configure and install the plugin
- Describes how to load and use the plugin.
- Provides an appendix that provides a sample extension to the folder Gui that allows you to specify filter formats for inactive filters.

Chapter 2. Installation

While the plugin can be installed anywhere, I recommend installing it in the SpecTcl installation directory. This makes loading it easier. The plugin is installed by following the usual two step **configure** and **make install** procedure used by most Unix/Linux open source software.

The **configure** script configures the Makefiles for build and installation. It accepts the followign switches that are specific to the plugin:

`--help`

Prints out exhaustive help describing the options and variables recognized by the **configure** script.

`--prefix=`*`path`*

`path` is the top leve directory for the installation tree for the plugin. The plugin itself is installed in the `TclLibs` subdirectory of this tree and has the name `librootfilterformat.so`

`--with-tcl-header-dir=`*`path`*

The configure script searches for `tcl.h` in the most likely places. If it reports it is not able to find it, you can tell configure where it actually lives by supplying the *`path`* parameter to this option.

`--with-spectcl-home=`*`path`*

The plugin must link to the SpecTcl libraries. By default, these are assumed to be located in the same directory tree as the one specified by the `--prefix` option. If, however you choose to install the plugin elsewhere, you must specify this option and *`path`* to be the path to the top level directory of the SpecTcl installation.

`--with-rootsys=`*`path`*

The plugin must also link to the Root libraries. These libraries cannot be automatically located; ther is a wide variation in where they are installed from system to system. Specify this option with the *`path`* argument specifying the value you would specify for the `ROOTSYS` environment variable when using Root.

Example 2-1. Building at the NSCL

```
./configure --prefix=/usr/opt/spectcl/3.2 --with-rootsys=/opt/etch/root/root5.16
...
make install
```

Chapter 3. Usage

This chapter describes how to use the plugin. Specifically:

- How to load the plugin
- Some background on SpecTcl filters is provided
- How to specify that an filter should be written in root ntuple format.
- What the root ntuple file contains.

3.1. Loading the plugin

The Tcl **load** command loads plugin libraries. You must specify the path to the plugin library completely. If the plugin has been installed in the SpecTcl installation, you can use the `SpecTclHome` variable to shorten the path.

The plugin will add a format selection to the SpecTcl **filter -format** command. The following dialog loads the plugin from the SpecTcl installation directory tree and ensures that it has installed correctly.

Example 3-1. Loading the plugin

```
% load $SpecTclHome/TclLibs/librootfilterformat.so ❶
% filter                                             ❷
Usage:
 filter [-new] filtername gatename {par1 par2 ...}
 filter -delete filtername
 filter -enable filtername
 filter -disable filtername
 filter -regate filtername gatename
 filter -file filename filtername
 filter -list ?glob-pattern?
 filter -format filtername format

filter creates pre-sorted event files

filter formats are:
xdr        - NSCL XDR system independent filter file format
rootntuple - Root file containing an ntuple named spectcl.    ❸
```

❶ This loads the plugin library assuming it was installed in SpecTcl's installation. The Tcl variable `SpecTclHome` is set by SpecTcl's initialization to be the path to its installation.

❷ The simplest way to check that the filter loaded correctly is to issue an invalid **filter** command. Like most SpecTcl commands, **filter** will then provide usage information. The **filter** command's usage text dynamically generates the list of supported output formats from the list of available formats.

❸ The `rootntuple` format is the filter output format that is added by the plugin. Its presence in the list of available formats indicates the plugin successfully loaded.

3.2. SpecTcl Filters

To use the plugin you need to understand SpecTcl filters. A SpecTcl filter cuts the data in two directions:

1. Only events that satisfy a filter's gate are allowed through the filter and written to the filter output file.
2. Only the specified subset of SpecTcl parameters are written to the filter output file for events that satisfy the gate.

In other words, a filter can produce a subset of parameters from a subset of events.

To use a filter you must therefore:

1. Create a gate that will select the events that are written to the output file.
2. Use the **filter -new** command to create a filter, specifying its name, gate and list of parameters.
3. Specify the file to which filter data will be written via the **filter -file** command.
4. Enable the filter to allow it to write data to file via **filter -enable** and disable the filter when done writing to a file via **filter -disable**

3.3. Specifying filters to output root ntuple files.

SpecTcl 3.2 introduced a new subcommand to the **filter** command **filter -format**, as well as an extensible filter output format subsystem. The original filter file format is the default format and is now called `xdr` format. The root plugin extends the set of output formats by adding the `rootntuple` file format.

A filter's output format can be set whenever it is not enabled. The **filter -format** command is used to set the format of the next filter file written.

The example below shows the command line based creation of a filter named root, setting it to write in root ntuple format, selecting an output file and enabling the filter to write data.

Example 3-2. Creating a root n-tuple output filter.

```
filter root filtergate [list event.raw.00 event.raw.01 event.raw.02]
filter -format root rootntuple
filter -file spectcl.ntuple root
filter -enable root
```

If you are creating the filter with the folder GUI, use the filter wizard as you normally do, however create the filter in the disabled state. Once created you can use the **filter -format** command to set the filter format to `rootntuple`, and then use the folder GUI to subsequently enable the filter.

The Appendix shows how to extend the folder Gui so that you can select the filter format for any disabled filter using that GUI.

3.4. Contents of a root ntuple filter.

So you have a filter file in root ntuple format. In order to analyze this file with root you need to know what it contains. This section describes that.

A SpecTcl Root filter file contains a single n-tuple named `spectcl`. The n-tuple contains all events SpecTcl analyzed to satisfy the filter gate while the filer was enabled to write to this file.

The n-tuple parameter names are the same as the names of the SpecTcl parameters that were selected for the filter. In the event a parameter was not valid for an event, its slot is filled with a silent ’Not a Number’ or *NaN* as they are called. These ’values’ are floating point values that will not be histogrammed, and can be deteced via the function `isnan` or `fpclassify`.

Appendix A. Sample GUI extensions to specify filter output format.

This section provides a Tcl/Tk script that adds functionality to the folder GUI that allows you to set the format of filter files from amongst the formats known to SpecTcl at the time. The code is provided in the filter tarball. This code is sample code and intended for instructional purposes. You may use it freely under the Gnu Public License, but it should not be considered as NSCL supported software.

The entire gui is part of the distribution tarball, and is the file `selectFilterFormat.tcl`. We are going to present this file a bit at a time, rather than just posting a large-ish chunk of code. The script relies on the snit Tcl package that is part of the tcllib. snit is an object oriented extension to Tcl that also supports the creation of *megawidgets*. A megawidget is a collection of simple widgets that operate to a client script as if they were an ordinary Tk widget.

The first chunk I'd like to present is the initialization. Assuming that we've created a widget `filterFormat` that will display the filters and let you set their formats, the following code adds a menu item to the folder GUI's Filter which pops up that widget:

Figure A-1. Adding a menu entry to the Filter menu

```
proc filterFormatDialog {} {            ❶
    set name .filterformatdialog
    if {![winfo exists  $name]} {       ❷
 filterFormat $name             ❸
    }
}

#   Add the dialog to the filter menu:


.topmenu.filter add command -label {Filter Format...} \   ❹
                            -command [list filterFormatDialog]
```

❶ The `filterFormatDialog` **proc** will be responsible for creating the dialog in response to the user's menu invocation.

❷ The widget should only be created if it does not yet exist. **winfo exists** *`widgetname`* returns a boolean true if *`widgetname`* already exists.

❸ This code actually creates the widget.

❹ This line adds a menu item labeled `Filter Format...` to the filter menu, which is named `.topmenu.filter`. When invoked, the `filterFormatDialog` proc is called to actually create the dialog.

Now I'd like to walk through the process of creating the widget. There are three code snippets we need to look at. The first is a utility function that produces a list of the filters that are not enabled. This is used to create the list of filters that can have their format set. The second snippet is used to list the set of formats that are available. This is used both to provide some documentation of the available formats on the dialog as well as to stock a menu of possible formats for each filter. Finally, we want to see the construction of the megawidget.

The proc below `inactiveFilters` lists the filters that are not enabled. This is done by obtaining a list of the filter descriptions and building a return list of the filters whose state is `disabled`. This proc is a proc local to the filterFormat snit::widget.

Figure A-2. Getting a list of disabled filters

```
snit widget:: filterFormat {
...
    proc inactiveFilters {} {
 set filters [filter -list]                ❶
 set result [list]
 foreach filter $filters {                 ❷
     if {[lindex $filter 4] eq "disabled"} { ❸
  lappend result $filter
     }
 }
 return $result                            ❹
    }
    ...
}
```

❶ The list filter descriptions is captured in the `filters` variable.

❷ Iterate over the filters in the list. The variable `filter` will contain a single filter description. A filter description may look like this:

```
{afilter test testing.flt {event.raw.00 event.sum event.raw.03} disabled xdr}
```

A list that contains in order, the filter name, the gate, the output filename, the list of parameters to be written to the filter, the state of the filter (`enabled` or `disabled`), and the current filter format.

❸ The body of this if is executed if the filter is disabled, and adds the filter description as a list element in `result`.

❹ The (possibly empty) list of disabled filters is returned.

The proc `filterFormatList` obtains the the list of filter formats and their descriptions. It does this by processing the output of **filter -help**, which shown below:

Figure A-3. Filter help text

```
Usage:
filter [-new] filtername gatename {par1 par2 ...}
filter -delete filtername
filter -enable filtername
filter -disable filtername
filter -regate filtername gatename
filter -file filename filtername
filter -list ?glob-pattern?
filter -format filtername format

filter creates pre-sorted event files

filter formats are:                                       ❶
xdr        - NSCL XDR system independent filter file format  ❷
...
```

❶ This line is the delimeter between help text and the list of known filter formats.

❷ This is a sample format description. I make use of the fact that the - character separates the filter format keyword from its description.

Here's the code that decodes this text:

Figure A-4. Obtaining a list of known filter formats

```
snit::widget filterFormat {
...
    proc filterFormatList {} {

 # Our only handle on this is to anlyze the help text.
 # The lines after the line that reads
 # "filter formats are:"
 # are the descriptions.. with keyword - description.

 catch {filter -help} helpText       ❶

 set helpText [split $helpText "\n"] ❷
 set linenum 0
 foreach line $helpText {
     if {[string trim $line] eq "filter formats are:"} { ❸
  break
     }
     incr linenum
 }
 incr linenum;    #  line number of first format.
 set descriptions [lrange $helpText $linenum end]   ❹
 set result [list]
 foreach line $descriptions {
     set format [split $line -]
     set key   [string trim [lindex $format 0]]      ❺
     set descr [string trim [lindex $format 1]]

     lappend result [list $key $descr]

 }
 return $result

    }
...
}
```

❶ Since there's not actually a **filter -help** command, and we are relying on filter to give us usage text on errors, the command is issued in a catch command to allow the script to continue executing, and to capture the error text from the command in `helpText`

❷ The help text lines are split into a list, one line per list element.

❸ The point of this loop is to figure out which line has the text just before the format list. In the end, `linenum` will be set to the number of the line that is the first format description line.

❹ This **lrange** command reduces the set of lines to just the format description lines.

❺ Each format description is split at the - and the format keyword and description are stored as a list in the final result list.

Now let's look at the code that builds the dialog. The dialog will be made up of a bunch of labels laid out using the **grid** command. The top set of labels will list the filters. The filter format will be clickable to pop up a menu of formats. The bottom set of labels will just be a reminder of the meanings of each format keyword.

Figure A-5. Laying out the widget

```
snit::widget filterFormat {                   ❶
    hulltype toplevel                         ❷
 ...

    constructor args {                        ❸



 # List the Filters               ❹

 label $win.filtertitle -text {Inactive Filters:}
 grid $win.filtertitle -

 label $win.filtername   -text Name
 label $win.filtergate   -text Gate
 label $win.filterfile   -text File
 label $win.filterfmt    -text Format
 grid $win.filtername   $win.filtergate $win.filterfile $win.filterfmt -sticky w

 set fnum 0

 foreach filter [inactiveFilters] {
     set name [lindex $filter 0]
     set gate [lindex $filter 1]
     set file [lindex $filter 2]
     set fmt  [lindex $filter 5]

     label $win.name$fnum -text $name
     label $win.gate$fnum -text $gate       ❺
     label $win.file$fnum -text $file
     label $win.fmt$fnum  -text $fmt

     bind $win.fmt$fnum <Button-1$gt;  \      ❻
                        [mymethod selectFormat $name $win.fmt$fnum %X %Y]

     grid $win.name$fnum $win.gate$fnum $win.file$fnum $win.fmt$fnum -sticky w

     incr fnum
 }

 # List the known filter formats:

 set formats [filterFormatList]


 label $win.formats -text {Key of filter formats: }
 grid $win.formats -


 foreach format $formats {
     set keyword [lindex $format 0]
     set descr   [lindex $format 1]
            label $win.${keyword}key  -text $keyword
     label $win.${keyword}descr  -text $descr   ❼
     grid  $win.${keyword}key  $win.${keyword}descr  -sticky w

 }
 # Method to dismiss:

 button $win.dismiss -text {Dismiss} -command [list destroy $self]
 grid $win.dismiss
    }
    ..
}
```

❶ The **snit::widget** command creates a megawidget class. The name of the class, `filterFormat` is the widget name command used to create widgets of this type.

❷ Snit megawidgets live in container widgets called *hull*s. The default type of hull widget is a **frame**. Since our widget will be a dialog, we want to be laid out directly in a **toplevel**. The **hulltype** command determines the type of widget used to contain the megawidget.

❸ The **constructor** method of a snit widget is invoked when the widget is created. Usually (and this is no exception), the constructor creates the widgets that make up the megawidget and lays them out in the hull.

❹ The next section of code provides titles for the list of inactive filters, and lays them out using the **grid** geometry manager.

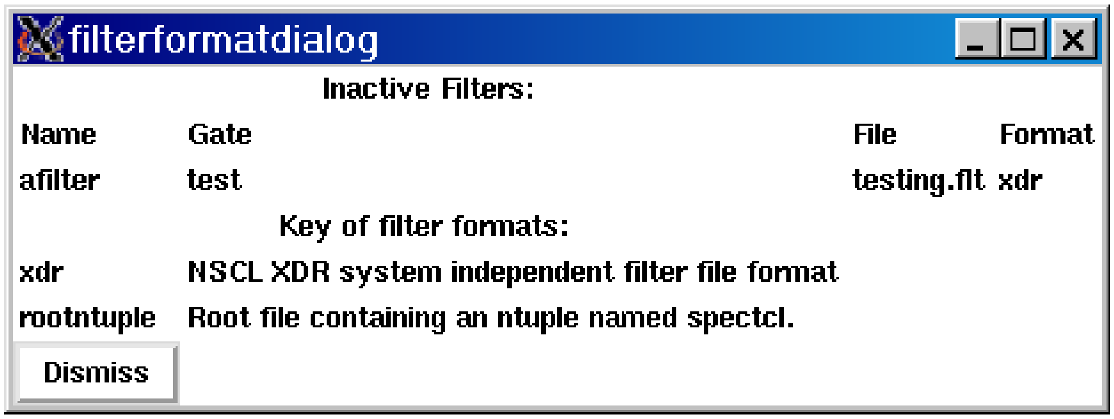
❺ Having used **inactiveFilters** already shown the descriptions are broken down into their parts, and labels built for each part.

❻ The **bind** command here ensures that if the user clicks the left mouse button when the pointer is over the format type, the proc **selectFormat** is called, which brings up the menu of format choices. `%X` and `%Y` are the screen coordinates of the pointer which provide a hint about where to pop up the menu. The other parameters are the name of the filter, and the name of the label widget that shows the current format. Both will be affected by changes in the format.

❼ This section of code produces a list of the supproted formats and their descdriptive text as label widgets.

When constructed, the final dialog might look something like this:

Figure A-6. The filter format dialog



Inactive Filters:

Name	Gate	File	Format
afilter	test	testing.flt	xdr

Key of filter formats:

xdr	NSCL XDR system independent filter file format
rootntuple	Root file containing an ntuple named spectcl.

The next chunk of code to look at is the **method** that responds to the mouse click on the format of a filter. The intent is for that to bring up a menu of radiobutton menu entries the user can select the new format type from.

Figure A-7. Generating the format menu

```
             ...
   method selectFormat {filter label x y} {              ❶

 set name $win.filterformatmenu
 destroy  $name                                    ❷
```

```
set formats [filterFormatList]

# construct the menu:

menu $name
$name add command -label $filter -command "" ❸
$name add separator
foreach format $formats {
    set fname [lindex $format 0]
    $name add radio -label $fname \
                           -variable ::${selfns}::format \ ❹
                           -value $fname \
             -command [mymethod formatSelected $filter $fname  $name $label]
}
tk_popup $name $x $y                        ❺
   }
   ...
```

❶ The `selectFormat` proc takes as parameters the name of the filter, the label widget and the screen coordinates of the mouse at the time the button was clicked.

❷ While it is an error to create an existing widget, it is not an error to destroy one that does not already exist. If the menu is already posted (e.g. you click on first one filter's format and then on a different filter's format without choosing a filter format for the first filter), it will be destroyed prior to making this new menu.

❸ The name of the filter is placed at the top of the menu as a command that does nothing. The name of the filter is separated from the radio buttons by a horizontal line **separator**.

❹ For each known format type, a radio button is added. The method `formatSelected` will be called in response to the selection, and is passed the filter name, the format name, the menu name and the name of the label widget associated with the format of the filter.

❺ Pops up the menu at or near the mouse position.

We now have one last bit of code to look at. The `formatSelected` method is called when one of the formats is chose from the popup menu:

Figure A-8. formatSelected - processing menu selections

```
...
   method formatSelected {filtername formatname menuwidget labelwidget} { ❶
 filter -format $filtername $formatname
 $labelwidget config -text $formatname
 destroy  $menuwidget
   }
...
```

❶ This method is relatively straightforward once you know what it must do and what the method parameters are. Method parameters are: The filter name *`filtername`*, *`formatname`* the format keyword selected from the popup menu. *`menuwidget`* the name of the popup menu widget. *`labelwidget`* the name of the label widget that indicates the current format for the *`filtername`* filter.

The method must set the new format for the filter, set the new text indicating the current format for the filter, and finally destroy the menu so it no longer is visible.