# AOS 452 - Lab 8 Handout
# C-shell scripts and four-panel plots

**Tip of the Day:** You may find the following information handy. GEMPAK contains two commands, `save` and `restore`, that are used to save and recall parameter lists. For example, let's say you are running **`gdplot`** and have it set up just right so that it produces a plot of 500-mb geopotential heights over North America. You could then type (in the GEMPAK prompt) `save hght500` (or whatever you want to call it). At any point in the future, you could then type `restore hght500` within **`gdplot`** to restore the settings you saved, so long as you are working in the same directory in which you saved the settings.

## Introduction

I am sure many of you have found that creating certain GEMPAK plots can occasionally be a time-consuming and frustrating task. However, matters can be simplified by doing a little computer programming. Do not get nervous! The programming to which you will be introduced today requires you to remember only a few simple commands. After looking through a few examples in this lab and working through some on your own, you will find that the programming required for creating maps in GEMPAK is quite easy.

Today, we are going to cover C-shell scripts. A shell is a program that provides an interactive interface between a user and the operating system. Basically, the main function a shell performs is to read in from the terminal what one types, run the commands, and show any output. The C-shell is one popular Unix shell, although there are many others. A C-shell script then is simply a text file, which is written in a particular format that contains all of these commands that would normally be typed into a terminal window, such as GEMPAK parameters. C-shell scripts will greatly reduce the time spent while working with GEMPAK programs.

## Creating a New C-shell Script for GEMPAK

Begin by opening up a new text file using your favorite text editor (**gedit**, **vi**, **emacs**, etc.). The very first line of the text file needs to be a 'flag' that identifies the file as a C-shell script:

```
#!/bin/tcsh -f
```

This line needs to be at the beginning of every C-shell script you create and make sure there are no spaces or blank lines in front of this command. A subsequent line that needs to appear in the script is a statement to start one of the GEMPAK programs. For example, if we wanted to run the GEMPAK program `gdplot` in the script, the line would look like:

```
gdplot << EOF
```

This will tell the machine to run the program `gdplot`, until EOF appears later in the script (as the '<<' symbols instruct the program to stop running at that point). This command is essentially the same as typing `gdplot` directly in the terminal window. Next, a list of all the parameter names that need to be changed must appear, using the same syntax as if you were changing them directly while running the GEMPAK program. After the list of parameter changes, you need the commands `run` (or `r`), `exit` (or `e`), and **`gpend`**.

So, the text file for your C-shell script would look something like this:

```
#!/bin/tcsh -f

gdplot << EOF
GDFILE   = $MODELDATA/16092800_eta.gem
GDATTIM  = f00.
.
.
(set all other parameters in gdplot)
.
.
.
run

exit
EOF
gpend
```

**NOTE: Make sure there is a blank line following every `run` (or `r`) command you place in your script. You may remember that most GEMPAK programs request that you hit `<Enter>` before they will make a plot. That is what the blank line is for.**

The easiest way to create this C-shell script is to begin by running the GEMPAK program you intend to use in the script and to find the list of parameter settings from within the program. Then, just *copy and paste* the entire set of parameters into the C-shell script and edit them to meet your needs. It is not necessary to have all of the parameters in the script, as you just need to enter the ones you intend to change. However, it is easiest to just copy and paste all of them into the script so that you do not forget about any of them.

Once the entire parameter list is included in the script, be sure to save it under any name you choose. For ease of use, try to make sure you save all of your scripts with names that have the extension, *.csh,* attached to the file name (e.g., hght500.csh).

In order to run a C-shell script, you have to make sure the script is an executable file, not just a simple text file. To make the text file executable, enter the following at the Unix prompt:

```
chmod 700 scriptname.csh
```

To run the script, simply type the name of the script into the terminal (*scriptname*.csh) and hit `<Enter>`. The script will then enter each line of the text file, as if you were typing it right to the terminal. (Note that some of your accounts may be set up such that you need to type `./scriptname.csh` to run the script.)

It is possible to use a wide variety of settings for the device parameter in GEMPAK scripts, such as postscript (ps), X-window (xw) or gif. When using an X-window (xw) as your device, you may want to remove `gpend` from the script. This way, the figure will remain on the screen when your script is done running so that you may view it. However, it is **very important** that you remember to use `gpend` when you are finished viewing your figure. The command `gpend` will close the graphics window, as well as any other GEMPAK processes that may be running.

As you may now see, scripts are an easy way to create plots containing overlays of multiple variables or multiple-panel figures. As an example, copy this C-Shell script over to one of your directories using the following syntax:

```
cp /ef5/raid6/class/fall11/mbreeden/Desktop/452labs2014/500vars.csh .
```

**OR**

**Download the script from Learn@UW if it won't let you copy the script due to permissions**

First, make this script executable as was previously described. Then, run the script by typing the name of the script at a Unix prompt. The script should create a figure of the 300-700 mb thickness, absolute geostrophic vorticity, and thermal wind barbs from the 0-hour forecast of this morning's GFS model run. Once you have the script running successfully, type **`gpend`**. Take a look at the script, and note how I was able to plot multiple variables (overlaying them/using different colors for each variable). Note that my script used the GEMPAK program **`gdplot`**. However, you can use any of the GEMPAK programs within a script.

## FOUR-PANEL PLOTS

It is often advantageous to show several plots on one output display (e.g., sheet of paper, X-window) to get a sense of the time evolution of a particular variable or to compare variables at different levels of the atmosphere at one particular time. The idea behind creating multiple panel figures is similar to that of creating overlays. You begin in your script by first running one set of parameters to the first panel with *clear* = yes. Then, you will run subsequent sets of parameters to the same device with *clear* = no. The key parameter in making multiple panel plots is the *panel* parameter. For instance, the parameter list before the first `run` command would include a *panel* parameter setting such as:

```
panel = 1/1/1/1
```

This will set the panel to the upper-left quarter of the window/page (for a 4-panel plot), with a box drawn around the panel with color 1 and width 1. Then, you will have a second parameter list, with a `run` command following it, with parameter changes including

```
panel = 2/1/1/1
clear = no
```

as well as any other additional parameters that you wish to change. There are several ways to divide a figure into different panels. I suggest typing `phelp panel` from one of the GEMPAK programs for more information.

## More Advanced C-shell Scripting

The basic structure of C-shell scripts as described thus far should allow you to create any plot you wish in GEMPAK. The information that follows describes a few commands and techniques in C-shell scripting that serve primarily as time savers. I have created three sample scripts that use these commands and techniques. To copy these files into your directory, enter the following at the Unix prompt:

```
        cp /ef5/raid6/class/fall11/mbreeden/Desktop/testscr*.csh .
```

**OR**

**Again, Download from Learn@UW if you can't copy these files**

### Script #1: *Comments, special characters in the title parameter, and the* **sleep** *command*

Start **gedit** and open *testscr1.csh*. Take note of the first line of the script: **`#!/bin/tcsh –f`**
*This exact character sequence is required on the first line of every script you create.*

The next three lines of text are comment lines. A comment line is created by entering a pound sign or hashtag (#) in the first space of the line. The line will be ignored when the script runs (an exception is `#!/bin/tcsh -f` on the first line).

You may have noticed the "@" and "~" symbols in the values for the *title* parameter. These symbols are part of a group of special characters that are replaced by certain values from other parameters when the script is executed. The "@" symbol corresponds to the vertical level (specified in *glevel*), while the "~" symbol corresponds to the valid date/time of the forecast (specified in *gdattim*). Other special characters can be found in `phelp title`. These characters can alleviate the hassle of editing the title every time you change the vertical level, forecast time, etc., in the script.

Now, run the script. The plot should appear on the screen for a few seconds, and then disappear. The time in which the plot is on the screen when `gpend` is in the script likely is much too short for you to determine the quality of the plot. This shortfall can be overcome by using the **sleep** command.

When a script encounters the **sleep** command, the script will pause for the specified amount of time at the place where the command is located. In *testscr1.csh*, uncomment **sleep 10** between `EOF` and `gpend`. The script will pause for 10 seconds between closing **`gdplot`** and executing **`gpend`**. Run the script again. You will see that the plot will stay on the screen for a longer period of time (about 10 seconds) than the last run.

### Script #2: *Input variables on the command line*

Now open *testscr2.csh* in a **gedit** window. The general structure of this script looks much the same as the first script. However, notice that there are some parameters with a value of a dollar sign followed by a number (e.g. *gdattim* = $1, *glevel*= $2, *cint* = $3, and *title* = $4). Basically, specific values in the parameters listed are replaced by variables $1, $2, $3, and $4. The values of these variables are specified in the command (terminal) line that runs the script.

Say we want an 18-hour forecast of 300-mb geopotential heights with a contour interval of 120. This can be done by entering the following at the Unix prompt:

```
testscr2.csh f18 300 120 '300 mb heights'
```

Notes:

→ You must be sure to specify the input values in the correct order in the command line. The first value after the script name corresponds to $1, the second value after the script name corresponds to $2, and so forth.

→ There is no limit to the number of $ variables you can use. For example, you can use $1,$2,…,$32 in place of **`gdplot`** parameters values in the script, and then specify the parameter values in the command line.

→ Input that contains spaces must be enclosed in single quotation marks to be considered a single value for one variable. For example, `'300 mb heights'` would be a value for one variable, while `300 mb heights` would be values for three variables (variable 1 = 300, variable 2 = mb, variable 3 = heights).

## Script #3: *Color fills, overlays and four-panel plots*

Producing overlays and multiple-panel plots using scripts is very easy compared to the drawn-out procedure we learned in previous labs. Open *testscr3.csh* in a **gedit** window. This script generates a four-panel plot with 300-mb height contours overlaying 300-mb isotachs, represented by color fills.

The first part of the script produces a plot in the upper-left quadrant, the second produces a plot in the upper-right quadrant, the third produces a plot in the lower-left quadrant, and the last part produces a plot in the lower-right quadrant. Notice that you do not have to list all the parameters after the first run of the GEMPAK program (in this case, **`gdplot`**). You need only to list those parameters you want to change.

Run this script. You can see each part of the script being run as the plot is created.

**Note**: For *any* GEMPAK plot you create that contains both color fills and contours, you must have the color fills produced first. If you plot color fills after contours, the color fills will cover up the contours.

## TOWARDS MORE COMPLICATED DIAGNOSTICS – (I.E. THINGS TO THINK ABOUT FOR A WX DISCUSSION)

Here are a few notes on using GEMPAK to compute complex diagnostics effectively…

1) First, decide if the diagnostic you are trying to show is a scalar or a vector. If it's a scalar, use `gfunc=`*`something`*; for vectors, use `gvect=`*`something`*.
2) I recommend using **`gdcntr`** (for scalars) or **`gdwind`** (for vectors) while you debug your diagnostic expressions. **`gdplot`** tends to give little information about what is going wrong.
3) `gfunc/gvect` does not work in isolation, but also depends critically on your settings for `gdattim`, `glevel`, `gvcord`, and `gdfile`. The resulting plot then depends on your scaling factor and contour interval, which work in tandem (i.e., `cint=60` with `scale=0` will produce the same contour lines as `cint=6` with `scale=-1`).
4) For layer functions, you need to specify two levels in `glevel`. For example, the 850–700mb thickness could be computed using:
```
glevel = 700:850
gfunc = ldf(hght)
```
→Use phelp gfunc to see what the function ldf does.
5) You can also use special symbols to override your `gdattim`, `glevel`, `gvcord`, and `gdfile` settings, as described in Appendix B1 (see the class website). For example, an alternate way to compute the 850–700-mb thickness is:
```
glevel = 700
gfunc = sub(hght,hght@850)
```

or you could say

```
glevel = 850
gfunc = sub(hght@700,hght)
```

6) How do I know the order of subtraction for the `ldf` and `sub` functions? Look at Appendix B1 of the GEMPAK documentation (available off of the class website), which is similar to `phelp gfunc,` but more detailed. For complicated vector functions, Appendix B2 can be very useful.
7) Functions can also be nested. For example, if I wanted to compute $-\hat{k} \times \vec{V}$, I can do: `gvect=smul(-1,kcrs(wnd))`

If you seem to be running into problems with your scripts, ask for help.