

# RavenR Tutorial

---

May 26, 2017

Robert Chlumsky

## Introduction

This tutorial is meant to provide an introduction to the RavenR package in the R Statistical language, which was created specifically for analysis of output files generated by the Raven hydrologic modelling framework. The RavenR package currently provides support for the following files:

- Hydrographs.csv
- ForcingFunctions.csv
- WatershedStorage.csv
- WatershedMassEnergyBalance.csv
- ReservoirStages.csv
- CustomOutput files
- Ostrich files

The following tutorial is provided to provide an example of the functions provided and introduce the user to practical examples using the tools for hydrologic analysis. A reference table is included on the next page, with a summary of all RavenR functions demonstrated in this tutorial, and some that are not yet included in the tutorial. This tutorial also serves in part as a very brief introduction to the R language. A brief section on quirks in R programming for readers coming from a different programming language is included, which will be useful in interpreting the code snippets found in this tutorial.

The tools presented here are still under development and will likely see future iterations. For any bug reporting or requests for future functionality, or comments on the tutorial itself, please submit a request directly to Robert Chlumsky ([rchlumsk@uwaterloo.ca](mailto:rchlumsk@uwaterloo.ca)). The RavenR package is under a [Creative Commons license](#), and is thus available for redistribution but provides no warranty on the product.

It is recommended that the user reviews the first few sections in the tutorial on installing the package and finding help in R prior to proceeding with the remainder of the tutorial. A basic knowledge of R is assumed, although anyone with basic programming in similar languages (Matlab, python, etc.) should not have too much trouble in following along.

The preferred environment working in R is RStudio; however the work can be done from within the basic R interface, from a command line, or other means. RStudio is recommended for analysis work as it provides an easy-to-use interface for running code, viewing stored objects, and viewing plots simultaneously, as well as a help interface that is easily accessed.

## RavenR Tutorial Functions Reference Page

Category	Function Name	Handles	Description	Section
Read/Process	forcings.read	ForcingFunctions.csv	Reads in forcings file	6
	watershed.read	WatershedStorage.csv	Reads in watershed storage file	7
	watershedmeb.read	WatershedMassEnergyBalance.csv	Reads in the water mass energy balance file	8
	custom.read	Custom Output files	Reads in custom output files	9
	hyd.read	Hydrographs.csv	Reads in hydrograph file	10
	hyd.extract	hydrograph	Extracts subbasin-specific series from hyd.read output	10
	res.read	ReservoirStages.csv	Reads in reservoir stages file	11
	res.extract	reservoir	Extracts subbasin-specific series from res.read output	11
	ECflow.rvt	WSC Gauge Data	Convert multiple WSC flow data to Raven format	NA
	ECmet.rvt	EC Climate Data	Convert EC climate station data to Raven format	NA
Diagnostics	hyd.plot	hydrograph	Plots hydrograph	10
	res.plot	reservoir	Plots reservoir stage	11
	customoutput.plot	Custom Output files	Plot custom output	
	forcings.plot	forcings	Plot the main forcings	NA
	flow.spaghetti	hydrograph	Creates spaghetti plot of overlapped flows	10
	cum.plot.flow	hydrograph	Cumulative flow plot	10
	Flowdurcurve.plot	hydrograph	Flow Duration Curve plot	NA
	flow.scatterplot	hydrograph	Scatterplot of sim and obs flows	10
	annual.volume	hydrograph	Compute/plot total volume in each water year	10
	annual.peak	hydrograph	Compute/plot annual peak flows	10
	annual.peak.error	hydrograph	Compute/plot annual peak errors	10
	annual.peak.event	hydrograph	Compute/plot annual peak flows at obs events	10
	annual.peak.event.error	hydrograph	Compute/plot annual peak errors at obs events	10
	annual.peak.timing.error	hydrograph	Compute/plot annual peak timing errors	10
	flow.residuals	hydrograph	Plot flow residuals	10
	monthly.vbias	hydrograph	Compute/plot monthly flow volume bias	10
Time Series Analysis	apply.wyear	hydrograph	Applies a function for the October 1st water year	6
	wyear.indices	hydrograph	Retrieves water year indices for a time series, based on October 1st	6
	wyear.indices.aus	hydrograph	Retrieves water year indices for a time series, based on July 1st	6
	wyear.indices.custom	hydrograph	Retrieves water year indices for a time series, based on a custom defined water year start date	6
Ostrich	Ost.read	OstModel0.txt	Read Ostrich calibration results	12
	Ost.bestparams	Ostrich	Get best params from Ostrich file	12
	Ost.plot	Ostrich	Various plots of Ostrich results	12
Misc	addColTrans		Adds transparency to colours	6

## Table of Contents

1	Brief Introduction to R Syntax.....	1
1.1	Commenting.....	1
1.2	Naming conventions .....	1
1.3	Object Handling .....	1
1.4	Assignment .....	1
2	Installation of RavenR .....	2
3	Loading Libraries.....	2
4	Getting Help in R (and RavenR).....	3
5	Starting the Tutorial .....	4
6	Forcing Functions File .....	5
7	Watershed Storage File .....	9
8	Watershed Mass Energy Balance File .....	12
9	Custom Output Files .....	12
10	Hydrograph File .....	13
10.1	Common Flow Diagnostics.....	14
10.2	Automating Flow Diagnostics .....	24
11	Reservoir Stages File .....	25
12	Ostrich Files .....	26
13	Running R from Command Line .....	28
13.1	Rscript is not working.....	28

# 1 Brief Introduction to R Syntax

This section is a (very) brief introduction to R syntax, which is particularly useful for readers familiar with other programming languages but not necessarily with the quirks of R. A few key concepts and syntax examples are illustrated here.

## 1.1 Commenting

Similar to some languages, a comment character in R is the number sign (#). Lines of code prefixed by the symbol, or code on a line after this symbol, will not be executed. This is used throughout this tutorial to provide comments on the code snippets for clarity. See the following as a simple example.

## 1.2 Naming conventions

Like most programming languages, R is case-sensitive. This means that ‘myvariable’ is not the same as ‘MyVariable’ or ‘myVARIABLE’. Another key note is that periods and underscores are allowed in variable and function names (the period is often used to access properties of objects in object-oriented programming, which is not true in R; discussed more below). For example, ‘my.variable’ is an acceptable name for a variable in R syntax.

## 1.3 Object Handling

Similar to object-oriented programming, R objects can contain sub-objects, and those can contain their own sub-sub-objects, etc. Each object is accessed using the ‘\$’ operator. For example, the syntax for accessing sub-objects from an object called ‘Family.pet’ is shown below.

```
Family.pet$dog
Family.pet$cat
# sub-object of dog
Family.pet$dog$bone
```

Each object stored in this way could be of different types, for example, a list, a string, an array, a boolean, etc.

## 1.4 Assignment

Assignment in most languages is done using the equal sign (=). In R the equal sign is still valid, however you will commonly see an arrow used for assignment as well (-> or <-). The arrow is constructed using a hyphen and a either the greater than or less than symbol, depending on the direction of the assignment. Two advantages of the arrow assignment syntax are:

1. It helps to distinguish assignment from comparison, i.e.  $x = y$  from  $x == y$ , more readily
2. The assignment can be done in either direction, whereas in most languages the variable that adopts a value is placed on the left, and the value being assigned is on the right.

The following is an example to show the assignment syntax in R.

```
# two equivalent statements
x = 5
x <- 5
```

```

# also equivalent statements
y = x
y <- x

# assign x value to z
x -> z

# re-assign z
10 -> z

# assign z value back to x
x <- z

# check value of x (two different ways)
x
print(x)

```

For those programmers who find the arrow operator to be in poor taste, please use the equal sign as per normal in your own code and be prepared to recognize the arrow operator as an assignment in some parts of this tutorial.

## 2 Installation of RavenR

The RavenR package can be installed similarly to other packages available in R. From a GUI such as RStudio, go to Tools → Install Packages, select to install from Package Archive File, and browse to the supplied .tar.gz file provided. From a command line, the RavenR package can be installed from file as:

```
install.packages('RavenR_1.0.1.tar.gz', repos = NULL, type="source")
```

Note that in order to install RavenR properly, you will need to also install the dependent libraries. Since RavenR is installed from source and the dependent libraries are installed from CRAN, you will need to install the dependent libraries first if they are not already installed. This can be done from Tools → Install Packages → Install from Repository (CRAN), or from a command line as per below. The dependent packages that need to be installed are `lubridate`, `xts`, `hydroTSM`, and `hydroGOF`.

```
install.packages(c("lubridate", "xts", "hydroTSM", "hydroGOF"))
```

## 3 Loading Libraries

One of the great features of R is the vast quantity of libraries available, which can all be easily downloaded and installed free of charge. Thus the loading of libraries is something to be familiar with in R. Installing and loading libraries is explained in more detail later, however it is important to understand how naming conflicts are handled when libraries are loaded.

To load RavenR (or any package), the command `library()` can be used.

```
> library(RavenR)
```

When the library function is called (with the name of the package to load as the argument), the packages that are also required in the operation of that package are loaded. For example, RavenR also make use of the lubridate, xts, zoo, hydroTSM, and hydroGOF packages.

If a package is loaded and there is a naming conflict for a particular function, R will let you know and will actually ‘mask’ or hide the earlier function in place of the newly defined one. In other words, if a package is loaded with a function name that already exists, the new function will be the accessible one. For example, when RavenR is loaded (and consequently the lubridate package is also loaded), the ‘date’ function is masked from R.

```
> library(RavenR)
Loading required package: lubridate
Attaching package: 'lubridate'
The following object is masked from 'package:base':
    date
```

This means that the original function ‘date’ in base R can no longer be accessed by simply calling ‘date’, as this will refer to the ‘date’ function in the lubridate package. However, the original date function is still accessible by specifying the ‘base’ package. Note that calling date() alone will throw an error, since the lubridate version requires an argument.

```
# call the original date() function; returns the date and time
> base::date()
[1] "Mon Apr 10 10:00:08 2017"
# call the new date() function from lubridate package; throws an error
> date()
Error in as.POSIXlt(x, tz = tz(x)) :
  argument "x" is missing, with no default
```

Once RavenR is installed and the library has been loaded, you should be able to view the list of functions available in the package using the command:

```
ls("package:RavenR")
```

## 4 Getting Help in R (and RavenR)

RStudio has a great way to access help files and search for functions. To look in the help pages for a specific function, use a ‘?’ prior to the function name. Using a ‘??’ will search the help files for that keyword, which is useful if you are unsure of what function to use or are browsing the help.

To use a function (or search for a function from a specific package), use the package name followed by two colons. For example, RavenR::hyd.plot refers to the hyd.plot function specifically in RavenR.

Finally, the code for a given function can be easily accessed by simply entering the function name as a command without brackets. This will show the code for that function in the Console.

The comments above are summarized in the code snippet below.

```
# help can be found in a few ways
?hyd.plot # lookup a single function
??hyd.plot # search for this word in help pages
??hydrograph # search help pages for functions related to 'hydrograph'
?RavenR::hyd.plot # search for this function in the RavenR package
specifically
hyd.plot # quickly look at the function script itself
```

## 5 Starting the Tutorial

The files in this tutorial are explained here:

- *model\_output* folder contains output files from Raven that are used in the tutorial.
- *RavenR\_1.0.0.tar.gz* is the zipped RavenR package, which can be used to install RavenR.
- *RavenR\_tutorial.R* is the R script for the tutorial. Use this script or following along with the commands in this document. Note that the script version has a few extra commands and comments that expand upon some of the notes in this tutorial.
- *Sample\_OstModel0.txt* is the sample Ostrich output file used in this tutorial.
- *Test1.R* is a simple script used in demonstrating how R files can be run from a command line.

The tutorial is roughly organized by the type of file being read in for that section, however there are various functions applicable to any data set that are discussed in each. For a full review of the tools available in RavenR, it is recommended to review all of the modules in the tutorial.

In this tutorial, you will explore the available functions in RavenR and gain some experience in the R language. This tutorial will walk through the main file types that you would examine in a typical workflow for real hydrologic modelling applications, including: forcing functions, watershed storage and mass energy balance, custom output files, hydrographs, and reservoir stages. In the course of this tutorial you will also learn how to create a time series plot, and create some common diagnostics plots for flow simulations. A section on the analysis of Ostrich files for single objective optimization, which is also commonly done in any hydrologic modelling workflow. Finally, an example of running R scripts from a command line is included. This can be used to automate an analysis with the RavenR package, and included in a batch or bash script. Much of the code included in the tutorial may be directly useful in your own modelling projects; feel free to copy and modify code snippets as you please.

Begin the tutorial by opening the *RavenR\_tutorial.R* file in the tutorial package. If this file is opened directly into an R interface or RStudio, the current directory should be set to one which this file is found. This is important, since the tutorial file uses relative file paths from this folder. The current working directory can be checked with the command `getwd()`. If you find that the current working directory is not the one you want (perhaps if another instance of R was opened previously, and the working directory is set to the location of another file), you can also change the working directory with the `setwd()` command.

## 6 Forcing Functions File

The forcing functions file can be read in with the `RavenR::forcings.read` function. The basic read function for Raven files using RavenR is to just supply a file path, and let the function do the rest. The usual return is an object with a data structure as well as units, and sometimes other items as well. All data objects are returned as a time series object in the `xts` format (see the `xts` package notes, `?xts`).

Try the following lines of code:

```
# read in the ForcingFunctions file; store into 'myforcings'
myforcings <- RavenR::forcings.read('model_output/ForcingFunctions.csv')

# what is the architecture like?
# data stored in $forcings; use head() to view first 5 lines
head(myforcings$forcings)

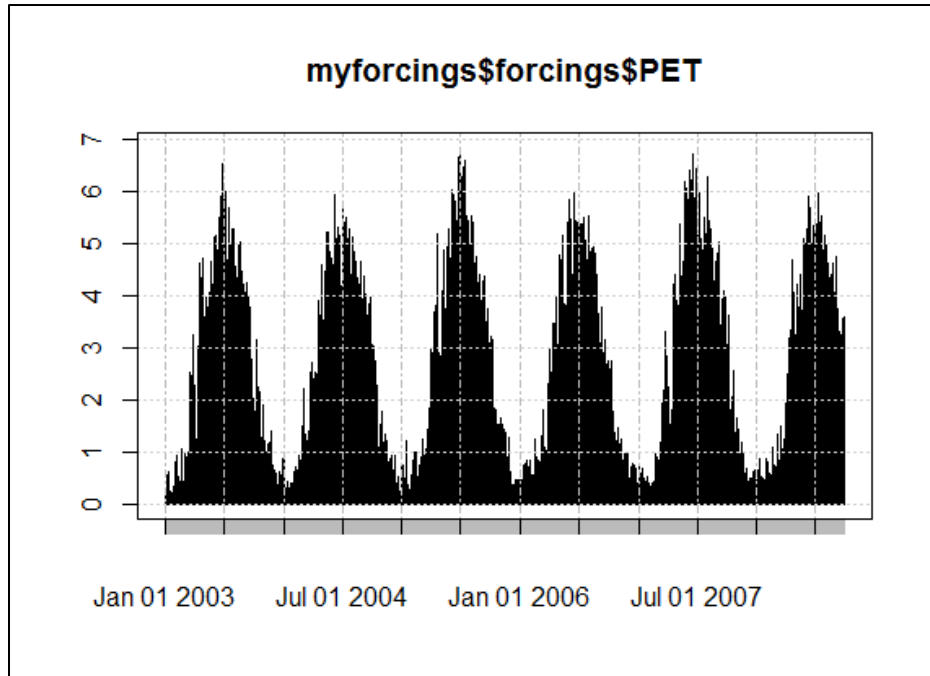
# units stored in $units (don't need head function, only one row of units)
myforcings$units
```

These lines of code read in the `ForcingFunctions` file and store it into an object called `myforcings`. In R, an analogous accessor to classic object oriented programming is done via the ‘\$’ operator, for accessing named sub-objects of R objects, where they are available. The `head` function shows the top five lines of a matrix or data frame, so the second line gives a preview of the ‘forcings’ object stored in ‘`myforcings`’. The units from the read file are stored into a sub-object called ‘units’, which can be viewed using the last command.

Now that we have a file read and stored, we can do some plotting and analysis. As mentioned, a nice feature of the RavenR functions is that everything is returned as a time series object directly (where it makes sense). For example, the command below will create a time series plot of the watershed-averaged PET for this model.

```
plot(myforcings$forcings$PET, type='h')
```



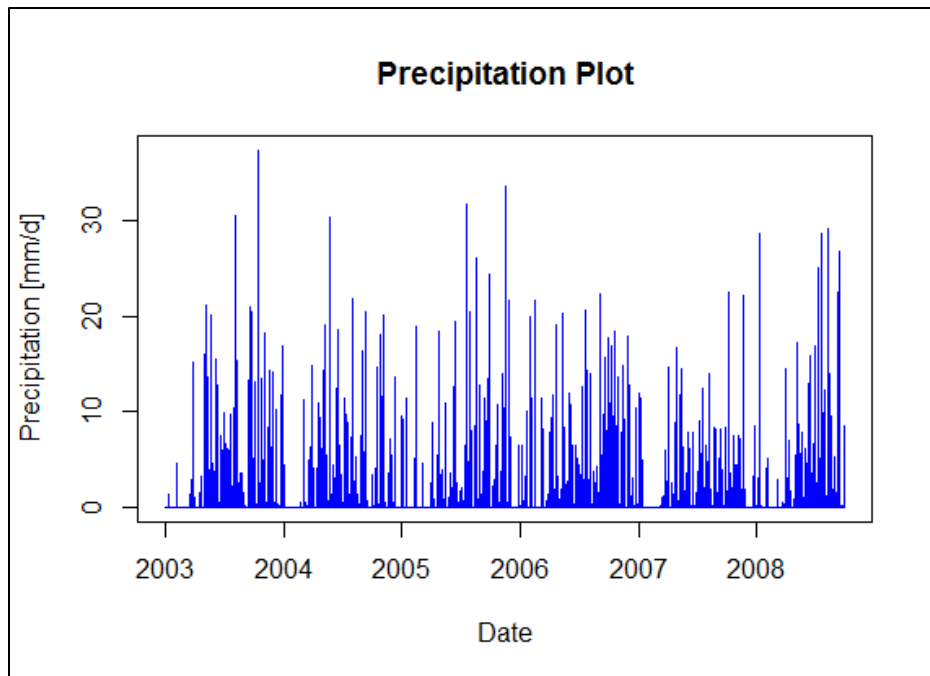


**Figure 1. PET time series plot generated from the ForcingFunctions file**

The y value provided to the plotting command is used automatically as a title, which results in the rather unappealing title given to the plot. The x axis is already formatted as dates, since the object is a time series object. The additional `type='h'` argument creates a line from the axis to the value, which is more appropriate for a daily-averaged PET plot.

A plot of the rainfall in the file can be done with:

```
plot(lubridate::date(myforcings$forcings$rain),myforcings$forcings$rain,main='Precipitation Plot',ylab='Precipitation [mm/d]',xlab='Date',col='blue',type='h')
```



**Figure 2. Precipitation time series plot from the ForcingFunctions file**

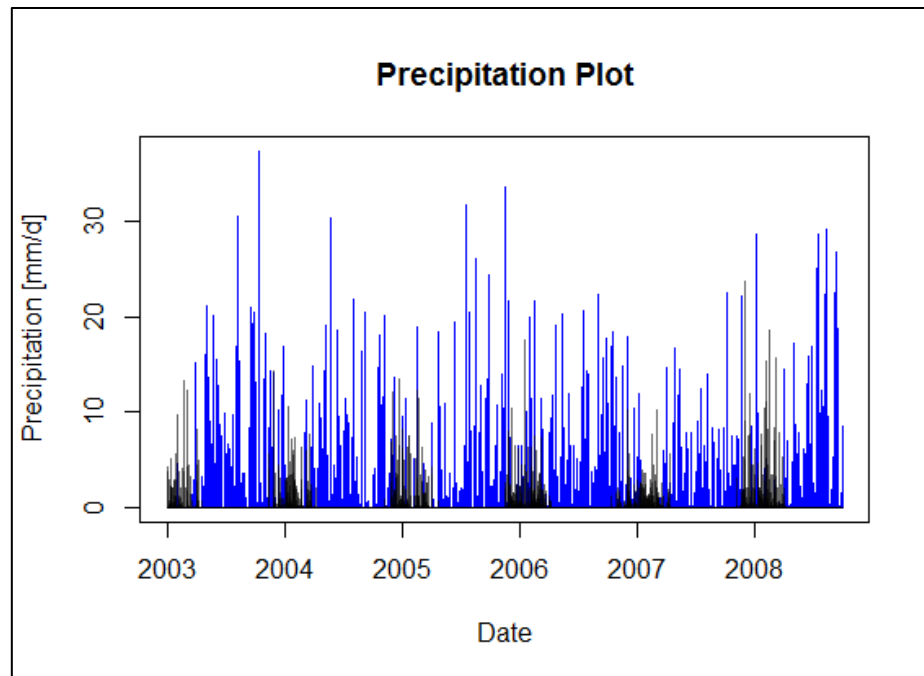
Here we actually specify a few more arguments to the plot function, such as plot title (`main=`), axes labels (`xlab`, `ylab`), and plot symbol colour (`col='blue'`).

A useful function in the RavenR package, collected from an online post on StackOverflow, is the `addColTrans` function, which adds transparency to a specified colour. Here we use the function to create a transparent black colour, and plot on top of the existing plot.

```
plot(lubridate::date(myforcings$forcings$rain),myforcings$forcings$rain,main='Precipitation Plot',ylab='Precipitation [mm/d]',xlab='Date',col='blue',type='h')

# define a transparent black colour using RavenR::addColTrans
my.colour <- RavenR::addColTrans('black',150)

# add additional lines to the same plot using lines()
lines(lubridate::date(myforcings$forcings$snow),myforcings$forcings$snow,col=my.colour,type='h')
```



**Figure 3. Precipitation time series plot with snow and rain separated**

The plot shows that the snowfall falls nicely within the breaks in precip, although there are some time with both snow and rainfall. This is still a little hard to see, but the advantage of transparent colours is hopefully clear.

A useful tool available from the xts package is the `apply.<period>` functions, such as `apply.monthly`, `apply.yearly`, etc. The `apply.monthly` function can, for example, be used to apply a generic function to each month in a time series. For example, to get the maximum monthly snowfall and determine when it happens, the following code can be used. Take a few minutes to break down the code to figure out how the arguments work.

```
?xts::apply.monthly
# store time series of maximum monthly snowfalls
temp <- apply.monthly(myforcings$forcings$snow, max)
ind.max <- which.max(temp) # find index of maximum monthly max value
sprintf("max monthly snowfall is %.2f mm, occurring in
%s", max(temp), date(temp[ind.max,]))
# outputs the following
# [1] "max monthly snowfall is 23.72 mm, occurring in 2007-12-31"
```

A useful version of this function available from the RavenR library is the `apply.wyearly` function, which applies a generic function to the water year in the time series. This is a version of the `apply.yearly` function that is particularly useful for hydrology. Note that the water year is hardcoded for October 1<sup>st</sup>, however additional functions are provided for finding indices from a time series for a July 1<sup>st</sup> water year (`?wyear.indices.aus`), or for finding indices from a time series for a custom-defined water year (`?wyear.indices.custom`).

```

RavenR::apply.wyearly(myforcings$forcings$snow, sum)
# outputs the following
# date.end      snow
# 1 2004-10-01 184.1346
# 2 2005-10-01 223.7970
# 3 2006-10-01 219.0879
# 4 2007-10-01 172.7210
# 5 2008-09-30 324.2193

```

The output above is the first five entries from the `apply.wyearly` function, which computed the cumulative snowfall in each water year of the snowfall series provided. The dates are in a period ending format, for example the first entry (2004-10-01) is the cumulative snowfall depth in mm for the 2003-10-01 to 2004-09-30 period.

## 7 Watershed Storage File

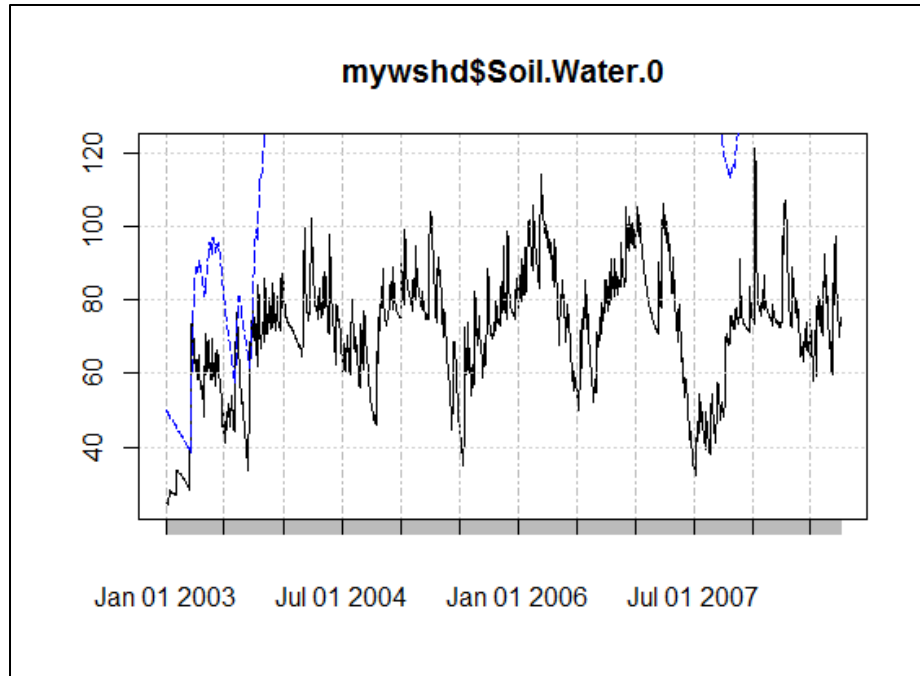
The reading in of the `WatershedStorage.csv` file can be done in a very similar manner to the `ForcingFunctions.csv` file. For example, here is a read in of the Watershed Storage file and basic plot of the `Soil[1]` storage and `Soil[0]` storage.

```

# for convenience, store this directly as the $watershed
mywshd <-
RavenR::watershed.read('model_output/WatershedStorage.csv')$watershed
head(mywshd) # check the first five rows

# plot the Soil[0] storage
plot(mywshd$Soil.Water.0)
# add Soil[1] storage
lines(mywshd$Soil.Water.1,col='blue',lty=5)

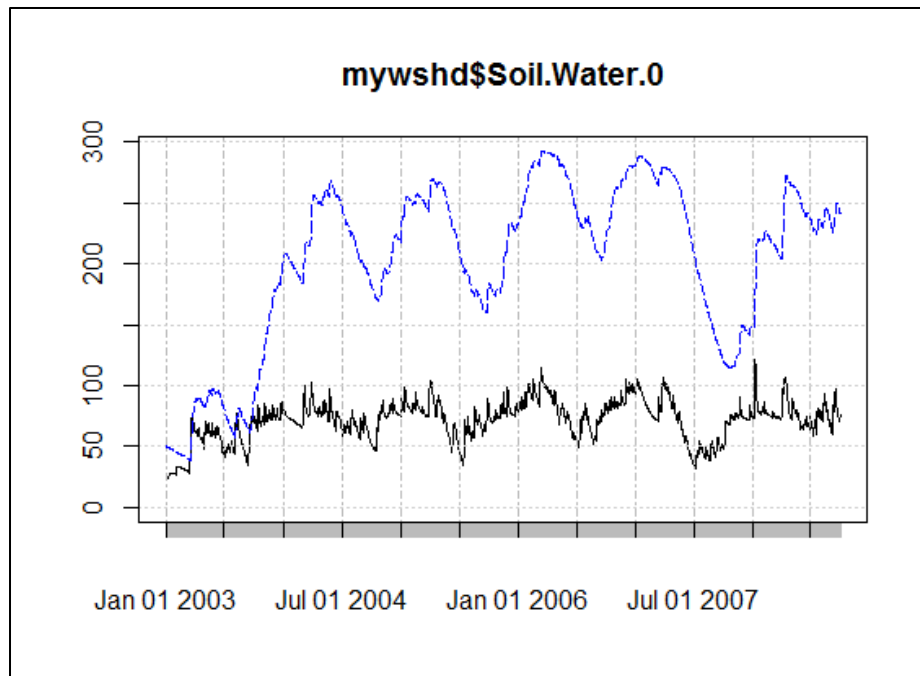
```



**Figure 4. Soil storage in top two soil layers**

The plot in Figure 4 is clearly out of bounds for the blue line (Soil[1] storage) but is fine for the black line (Soil[0] storage). This is because when a plot is made using the `plot()` function, the default axis limits are based on what is plotted. The `lines()` command, on the other hand, only adds lines to an existing plot and cannot therefore change other properties of that plot. In our example, the y axis limits are defined by the Soil[0] values, and the Soil[1] values are too large for the plotted limits.

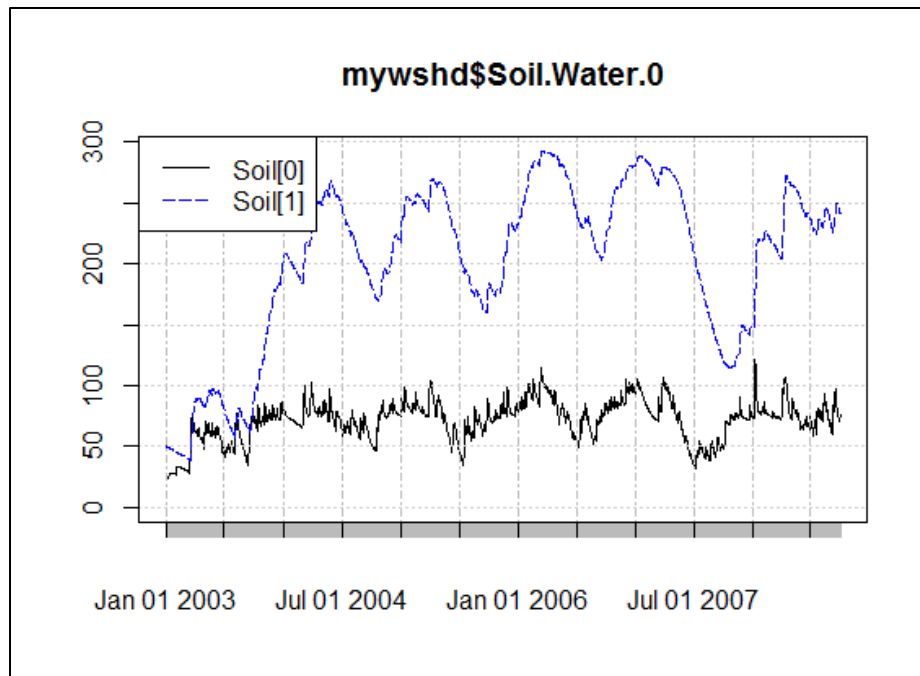
To overcome this, we can use the `ylim` argument in the plot function when we first create our plot. The following example will set the y axis limits to the maximum of both of the series to be plotted. The minimum value can be set as zero, since we know that the storage will not go below zero in this model.



**Figure 5. Soil storage in top two soil layers with appropriate axis limits**

This creates a plot with the appropriate limits for both series, making a much cleaner plot. We should also include a legend to let people know which lines are which. The `legend()` function operates similar to the `lines()` function in that it adds to the last plot created. For the plot above, we would like our legend to be in the topleft corner and appropriate text labels. The legend function builds up the different components of the legend separately, in terms of line colours, line thicknesses, labels, etc. See if you can determine which argument does which in the following command; refer to the help on the legend function with `?legend` for more information and additional controls.

```
legend(x='topleft', legend=c('Soil[0]', 'Soil[1]'), lty=c(1, 5), col=c('black',
'blue'))
```



**Figure 6. Updated soil storage plot with a legend**

## 8 Watershed Mass Energy Balance File

The WatershedMassEnergyBalance.csv is optionally created by Raven, and contains more detailed information on the mass and energy fluxes in the watershed. This file includes a listing of the ‘to’ and ‘from’ compartments that each flux is operating on. This file may not be examined during every analysis, however it can be read in with the `watershedmeb.read()` function from RavenR. The following is an example of reading in the file, previewing the data, and showing the from and to connections that are returned as well.

```
mywshd.meb <-
RavenR::watershedmeb.read('model_output/WatershedMassEnergyBalance.csv')
head(mywshd.meb$watershedmeb,7) # preview first 7 rows of data

# show the from and to in the file
rbind(mywshd.meb$from,mywshd.meb$to)
```

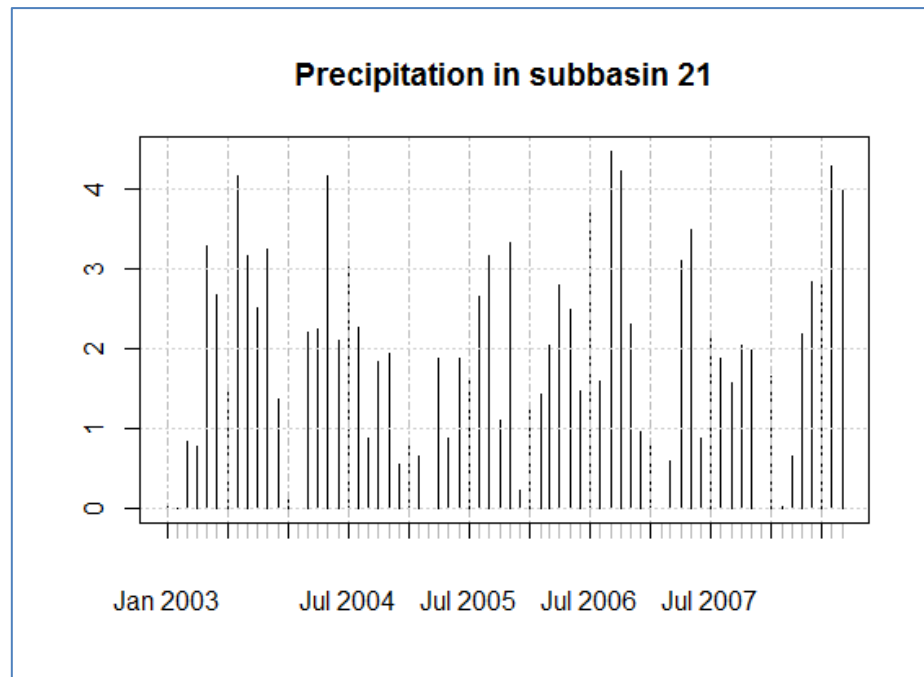
## 9 Custom Output Files

The custom output files produced by Raven with the `:CustomOutput` option are available for reading in as well. Here the function `custom.read` is capable of parsing the filename (as produced by Raven) and determining the proper format accordingly. The format will change slightly if the file is produced for a different spatial or temporal output, and thus this is a useful tool that does more than just a basic read in of a csv file. The following is an example of a basic plot for the monthly rainfall by subbasin custom output file.

```
# read in custom output rainfall file, store into 'custom1'
custom1 <-
RavenR::custom.read('model_output/RAINFALL_Monthly_Average_BySubbasin.csv'
)

head(custom1)

# Create a basic plot with a custom title
plot.title <- sprintf('Precipitation in subbasin
%s', colnames(custom1)[21])
plot(custom1[,21], type='h', main=plot.title) # precip in subbasin 21
```



**Figure 7. Time series plot of precipitation in subbasin 21 from custom output file**

## 10 Hydrograph File

The hydrograph file is likely one of the most common files to be read in for many of the available diagnostics in hydrology, and the functionality for handling this read file is slightly more sophisticated than the other RavenR read functions thus far. Once the file is read in, the relevant simulated series for a particular subbasin are extracted and read in accordingly; the information is sorted by simulated observed, and inflow series. This makes the process of finding the relevant series for a particular subbasin very easy-to-use in plotting and diagnostics. This is best illustrated with an example.

```
# read in first hydrograph file - store all flow series
myhyd <- RavenR::hyd.read('model_output/Hydrographs.csv')

# extract specific subbasin data from subbasin 106
sub106.flow <- RavenR::hyd.extract('sub106', myhyd)
```



```

plot(sub106.flow$sim)

# note that in this subbasin, there are no observed flows or inflows,
# objects are null; only sim is not null

# trying to plot them will result in an error
plot(sub106.flow$obs) # will result in an error
plot(sub106.flow$inflow) # will result in an error

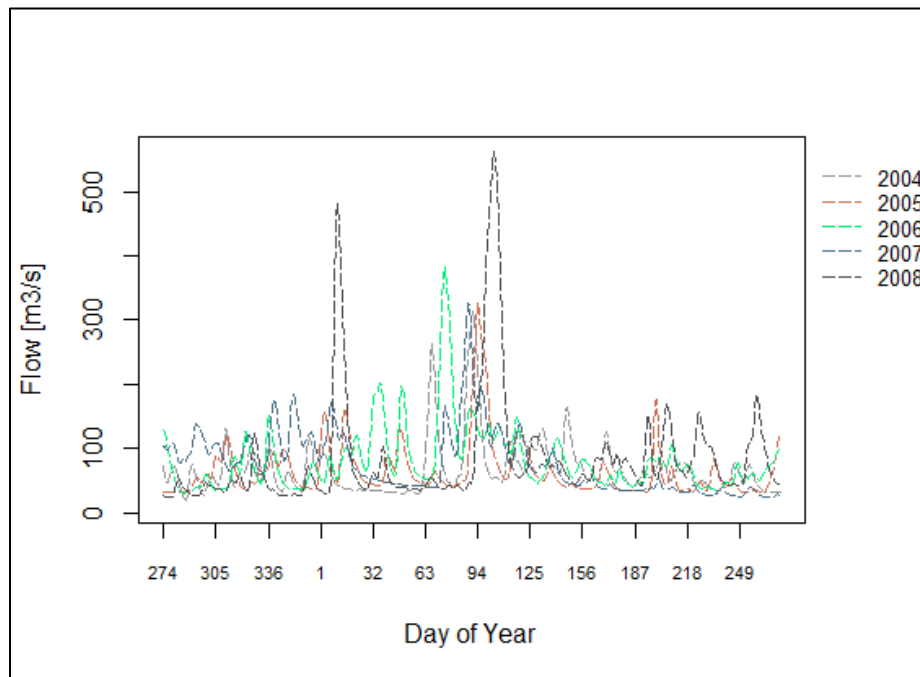
```

Here is simulated series for subbasin 106 is available in the read in Hydrographs file, so these can be plotted. The observed data and inflow data for that subbasin are not supplied, so the ‘obs’ and ‘inflow’ sub-objects are returned as NULL (and attempting to plot either will result in an error).

## 10.1 Common Flow Diagnostics

We can start to make use of some of the flow-dedicated diagnostics available in this package. With the previously extracted `sub106.flow` object, we can easily create a spaghetti plot using the `RavenR` function `flow.spaghetti()`, in which the annual flow series are plotted on top of each other in the same plot (allows the modeller to compare different years of simulation simultaneously). Note that the colours are, by default, randomly sampled from the available colours in R, so the colours will be different than the plot below. By default, a legend of the year is also included. Here we created spaghetti plot for the simulated flows, however the same could be done for any flow series.

```
RavenR::flow.spaghetti(sub106.flow$sim)
```



**Figure 8. Spaghetti plot of flows for each year in simulation**

A common task in diagnostics, especially if we are automatically generating diagnostics for different stations or in a calibration setup, is to save the plot to file. We can do this in a number of ways, however here is a simple way to save a .png file to the drive. The basic procedure is to create a blank png file of a particular name, create a plot, then remove the plot focus which releases editing in R on the file. A file called my.spaghetti\_plot.png should appear in your working directory once the following lines are run.

```
png(filename='my_spaghetti_plot.png')
RavenR::flow.spaghetti(subl06.flow$sim)
dev.off()
```

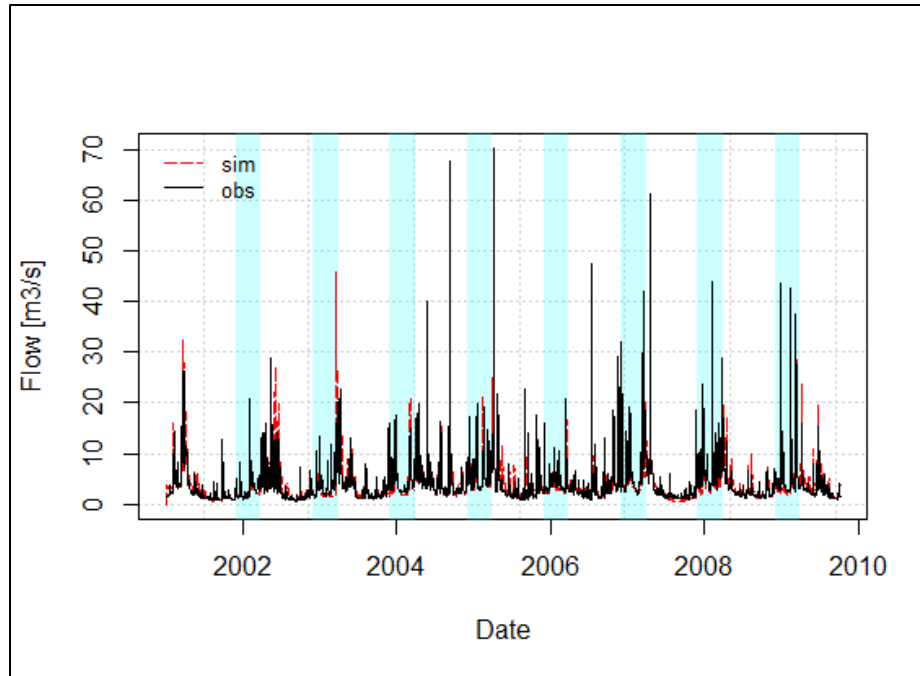
For many of the diagnostics related to flow, it is necessary to have observed flows to compare to. For this we will read in the second Hydrographs file for analysis, and read in the flows as well as the precipitation column. The hydrograph in this case is for the Irondequoit model, available in the Raven tutorial files. For convenience, we will assign the simulated and observed flows as sim and obs, respectively.

```
myhyd2 <- RavenR::hyd.read('model_output/run1_Hydrographs.csv')
head(myhyd2$hyd) # check hydrograph data for column names
# extract flow for the Irondequoit outlet; store into Iron.flow
Iron.flow <- RavenR::hyd.extract('Irondequoit',myhyd2)
# can also extract precip series from the hydrograph object
precip <- RavenR::hyd.extract('precip',myhyd2)$sim

# assign sim and obs, avoid typing out the extension
sim <- Iron.flow$sim # assign sim
obs <- Iron.flow$obs # assign obs
```

A simple hydrograph can be created easily now using the hyd.plot function. Here we just pass the simulated and observed series to the function, and the remaining arguments are left as default values.

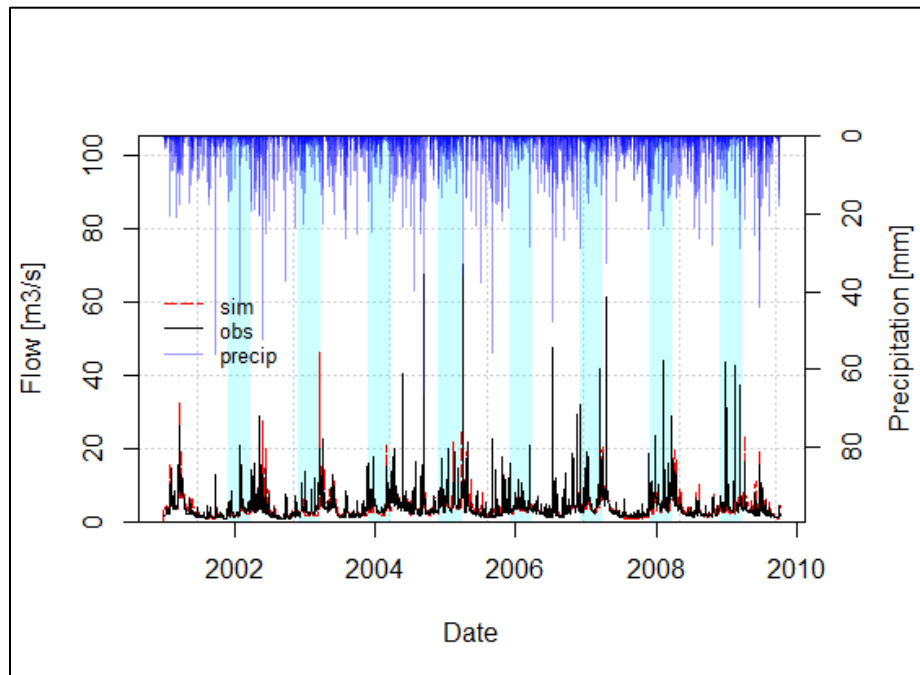
```
RavenR::hyd.plot(sim,obs)
```



**Figure 9. Hydrograph of simulated and observed flows**

A hydrograph with the precipitation data plotted on the secondary y axis (with the axis range reversed, as is typically done) can be done by including the precipitation data. Here the range of both the flows and precipitation is extended 1.5 times using the `range.mult` argument, which is not necessary but does help to create a nicer plot by ensuring that the hydrograph and precipitation lines do not overlap.

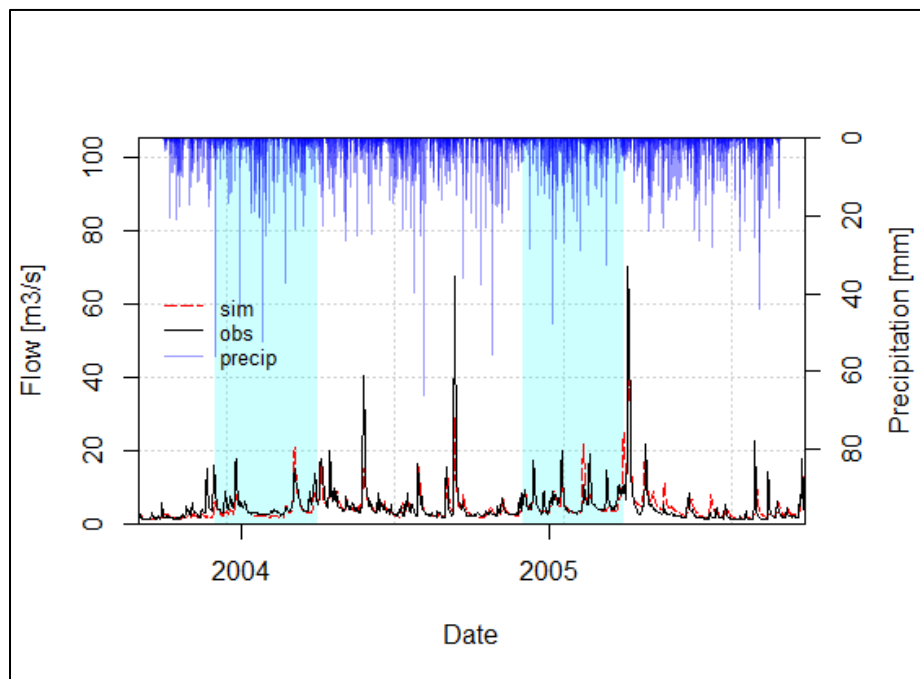
```
RavenR::hyd.plot(sim, obs, range.mult=1.5, precip=precip)
```



**Figure 10. Hydrograph of simulated and observed flows with precipitation included**

We can also choose to specify a particular plotting period in the `hyd.plot` function, which will only plot the data for that window of time. Note that in the background the `hyd.plot` function checks the simulated series (or one of the available series) and assigns the period for plotting accordingly, so the same could be achieved by modifying the actual `sim` series provided to the function. However, here we will use the same series and give the `prd` argument to the function to create a smaller plotting window. Note that the date format for the `prd` argument is always ‘YYYY-MM-DD’.

```
hyd.plot(sim, obs, range.mult=1.5, precip=precip, prd="2003-10-01/2005-10-01")
```

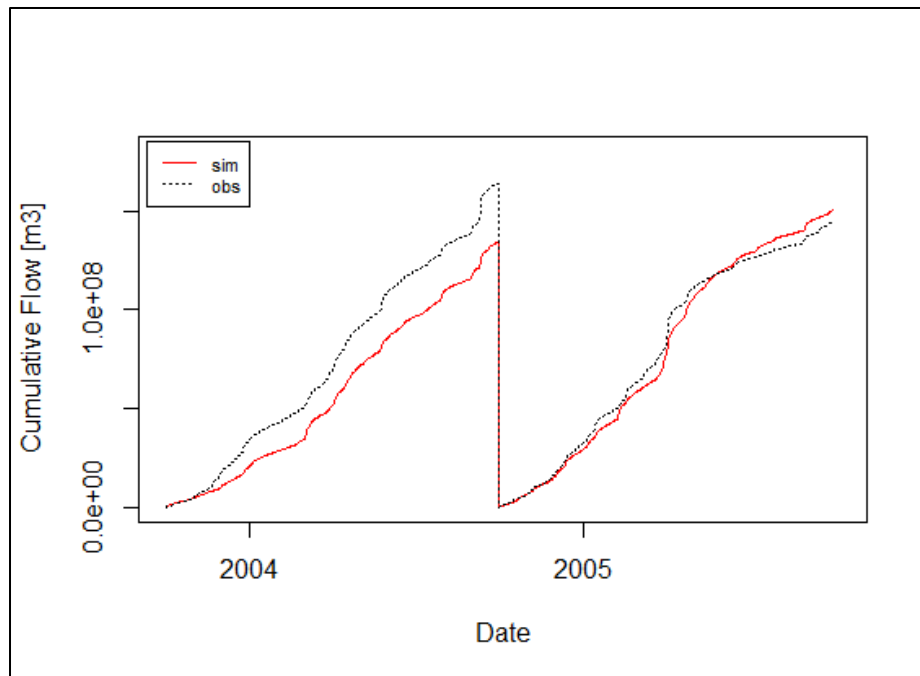


**Figure 11. Hydrograph plot for shorted duration**

The same plot is created, but only for the two year period specified. Other plotting options are available for the `hyd.plot` function, such as legend box and position, and whether to buffer the y axis; see `?hyd.plot` for more details.

We can also produce a cumulative flow plot with the simulated and observed flows. Note that we could also do this for a single series, however it is useful to check the volume of water (and timing of water outflow) with this plot. Here we limit the years of data supplied to the function by adding the [`<period>`] block to our time series, and is quite useful. The `cum.plot.flow` function does not have a `prd` argument as the `hyd.plot` function does, so this is our alternative. Note that the `cum.plot.flow()` function uses October 1<sup>st</sup> as the water year, which is evident from the plot date axis in the produced plot.

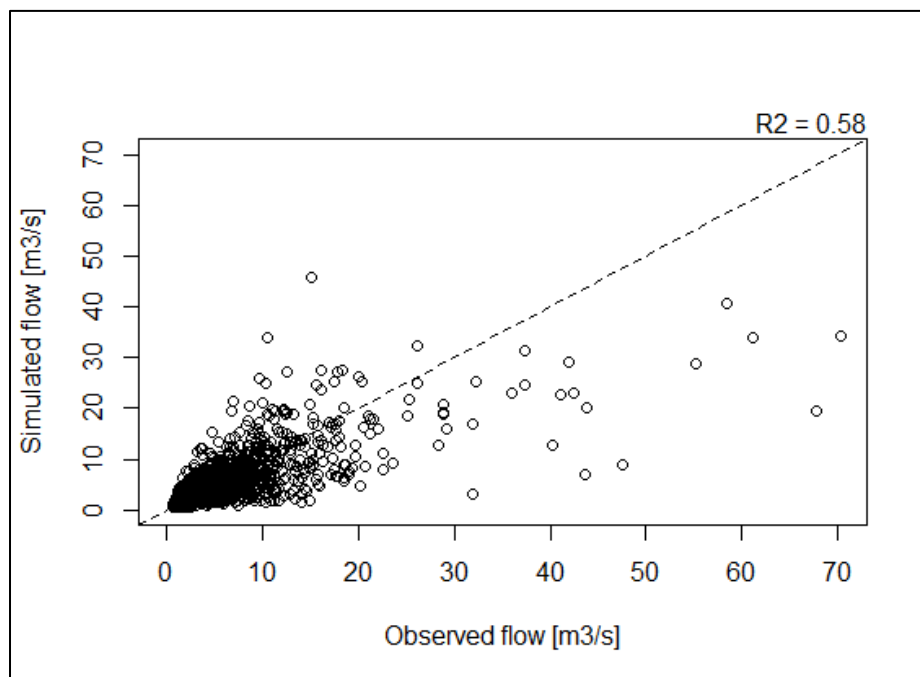
```
# create string with our desired period
myperiod <- "2003-10-01/2005-10-10"
# plot cumulative flows
RavenR::cum.plot.flow(sim[myperiod], obs[myperiod])
```



**Figure 12. Cumulative flow plot for two year period**

A scatterplot of the simulated and observed flows is occasionally used for checking the match of flows. Ideally points fall nicely on the 1:1 line, although in many cases, correlated errors will show up in this type of plot. The  $R^2$  metric can be optionally placed in the topright corner of the plot.

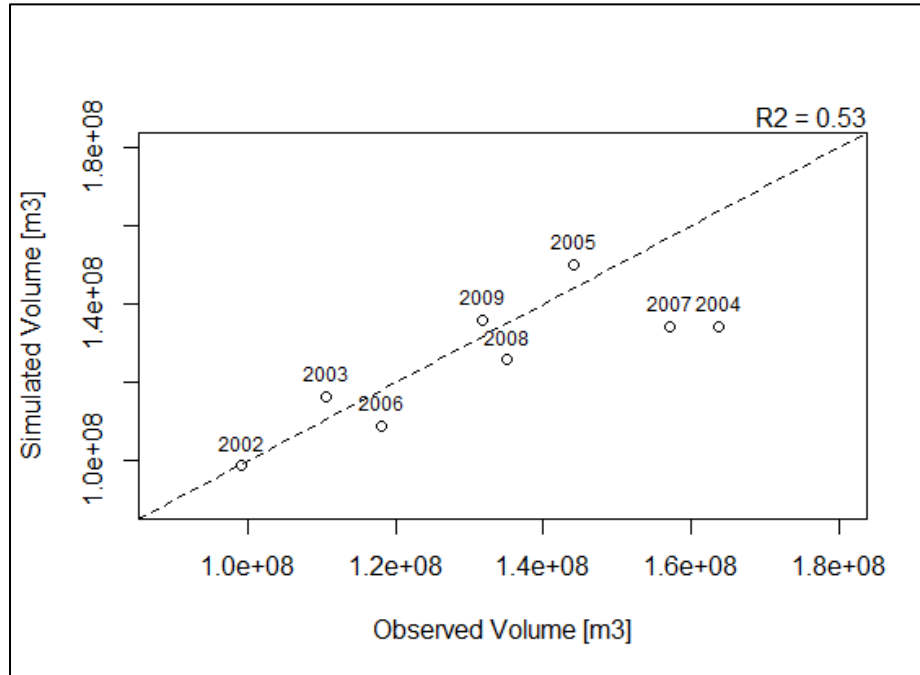
```
RavenR::flow.scatterplot(sim,obs,add.r2=T)
```



**Figure 13. Scatterplot of simulated and observed flows**

The annual volume can be plotted for simulated and observed flow series; again, in an ideal case the points fall on the 1:1 line. This provides a good visual check on whether the volume is generally overestimated or underestimated, and if there is any correlation in the various years.

```
RavenR::annual.volume(sim,obs,add.r2 = T)
```

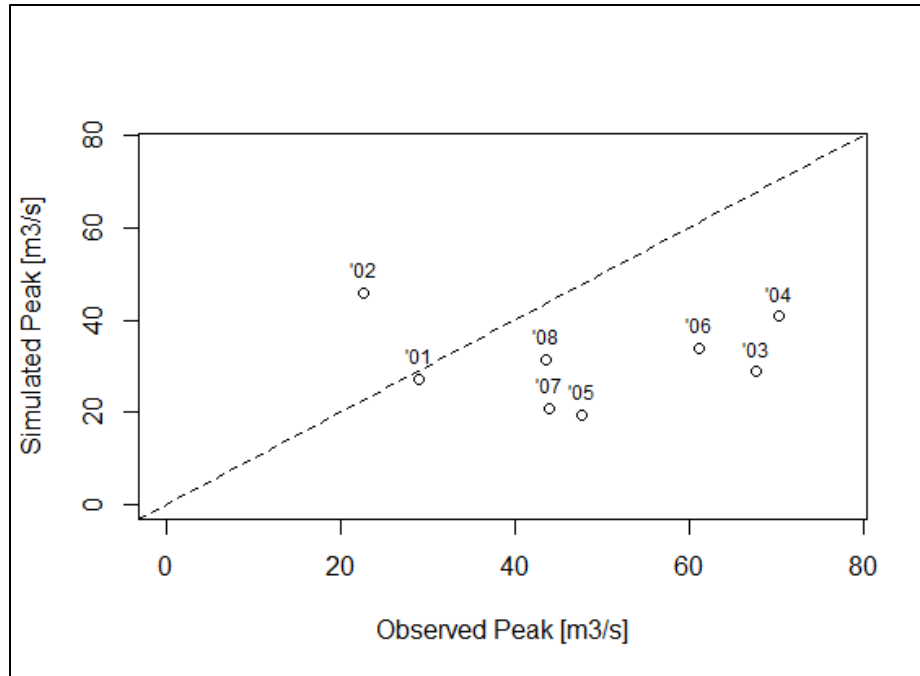


**Figure 14. Plot of simulated and observed annual volumes**

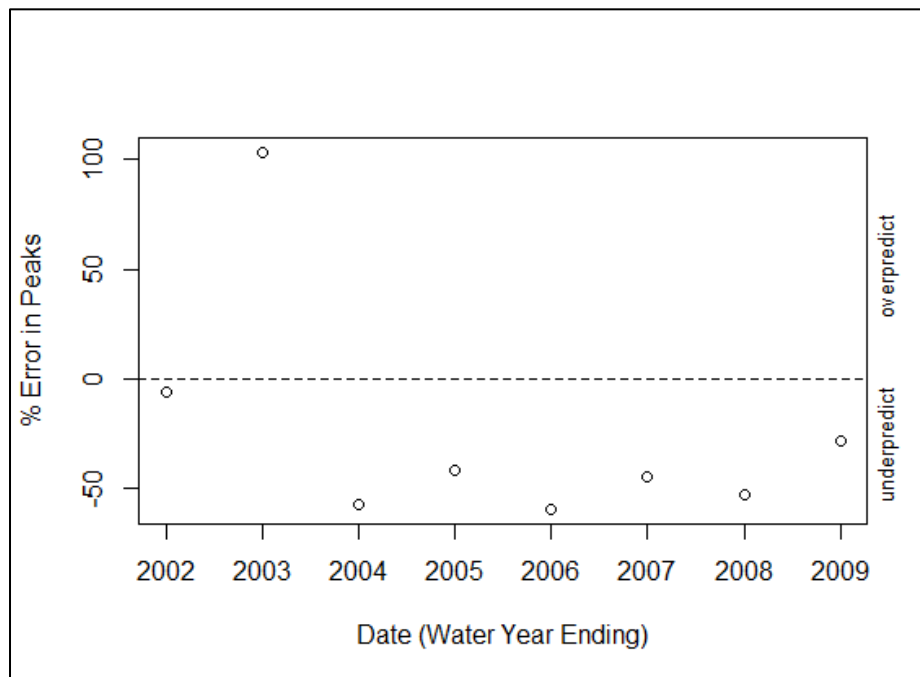
The annual peak event in each year can also be plotted for the simulated and observed series. This provides a check on the peak prediction of the model. The errors in the peaks can also be plotted explicitly. The `axis.zero` argument in the `annual.peak()` function will set the axis limits to zero instead of calculating the limits based on the minimum and maximum of the peak flows to be plotted, which may be preferred in visualizing the discrepancy in peak flows. By default, the `annual.peak.error()` function includes helpful labels on the right-side axis for 'overpredict' and 'underpredict' to aid in the interpretation of the plot, however these can be turned off by adding the `add.labels=F` argument to the function.

```
RavenR::annual.peak(sim,obs,axis.zero=T)
```

```
RavenR::annual.peak.error(sim,obs)
```



**Figure 15. Scatterplot of simulated and observed peaks**



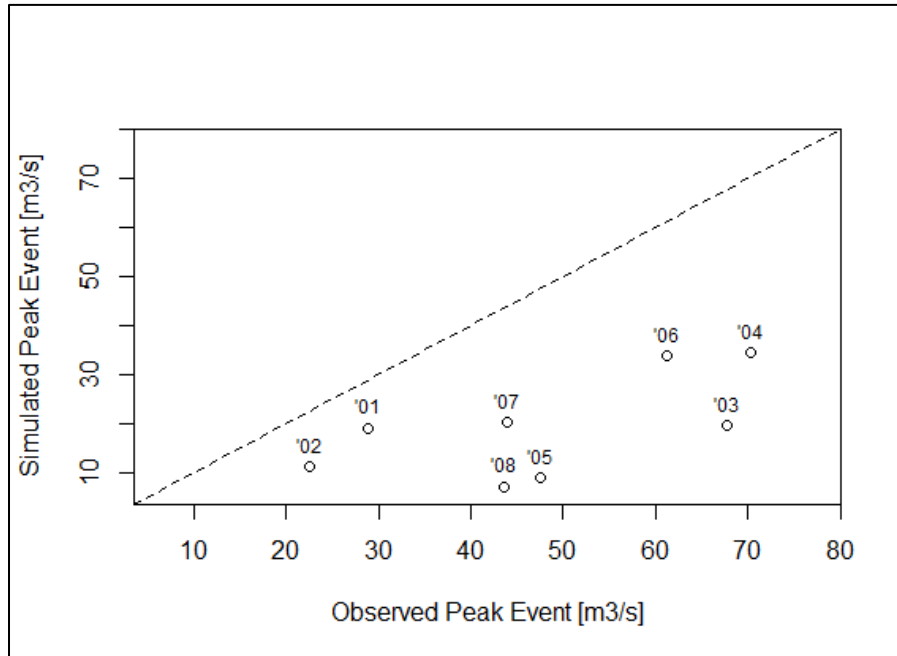
**Figure 16. Plot of percent error in peaks**

The `annual.peak()` function provides a check on the peak value in each year, but there is no consideration of when this event occurs, or if it is the same event at all. For this the `peak.event` can be used, which takes the simulated flow value at the same day as the observed peak value, and compares those two values. If the simulated peak occurs on the same day as the observed peak this plot should be

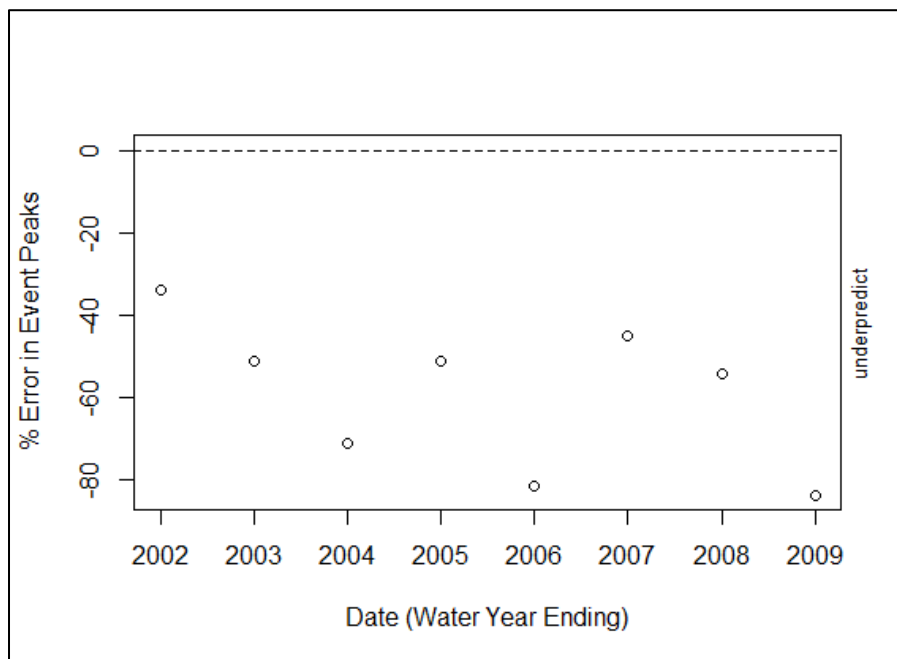
the same as the `annual.peak()` function, however in reality there is usually some error in the timing, so this fit will typically be worse. This function also assumes a daily flow simulation and daily flow observed values. The errors for the ‘peak events’ can also be explicitly plotted. Here the argument `add.r2=F` is added to remove the  $R^2$  diagnostic from the plot.

```
RavenR::annual.peak.event(sim,obs,add.r2=F)
```

```
RavenR::annual.peak.event.error(sim,obs)
```



**Figure 17. Scatterplot of simulated and observed peak events**

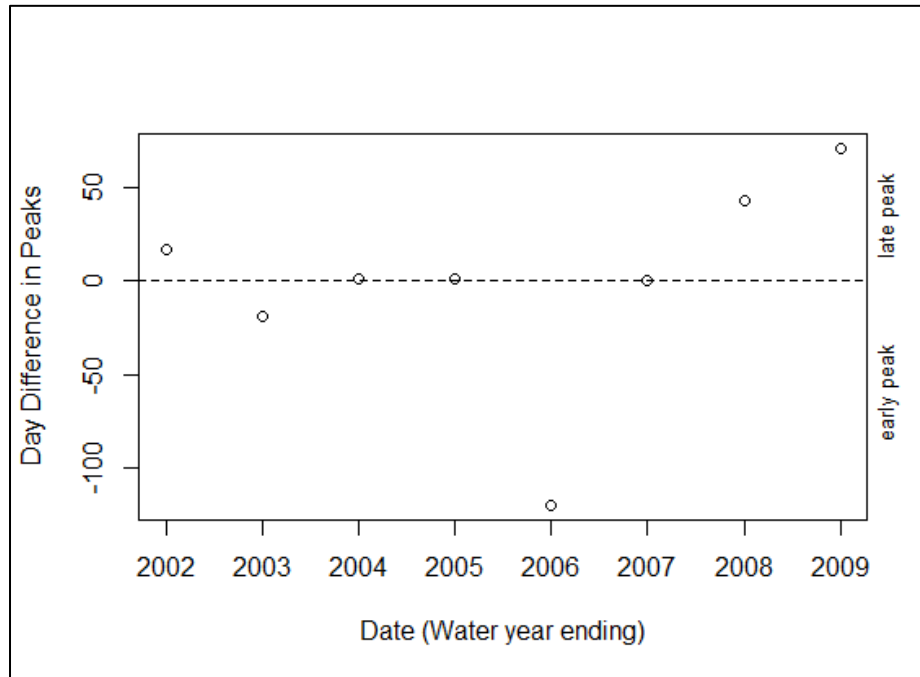


**Figure 18. Plot of percent error in peak events**



Finally, the errors in timing can be checked as well. The timing is implicitly checked using the `annual.peak.event()` function, however the `annual.peak.timing.error()` function will plot the difference in days between the peaks of the simulated and observed series. This is a good check to ensure that the peak event is actually the same event in both series, and that there is not an overestimated event at a different time of year which is coincidentally close in magnitude to the observed peak, for example.

```
RavenR::annual.peak.timing.error(sim,obs)
```

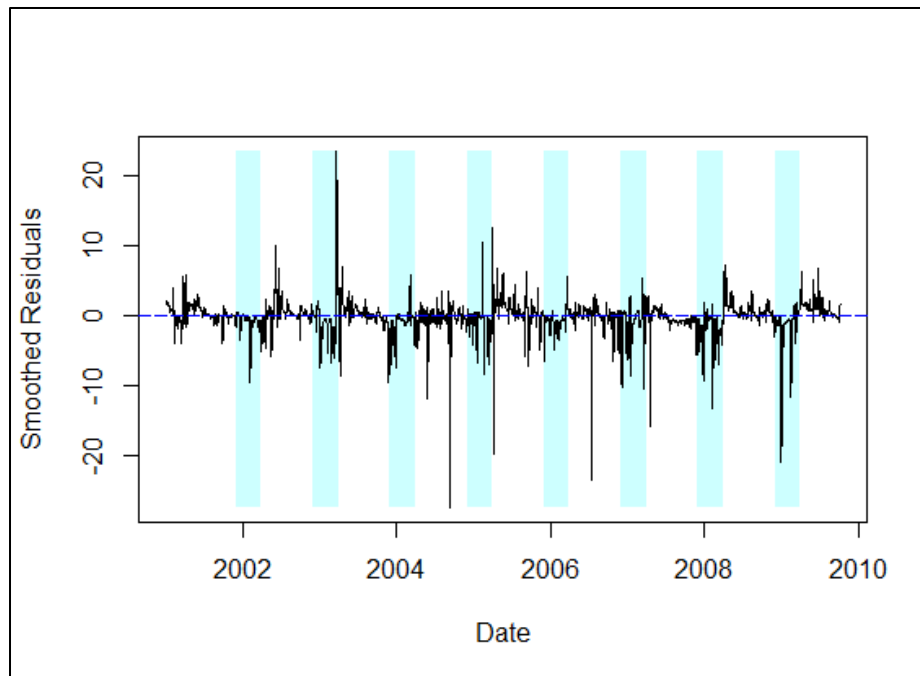


**Figure 19. Plot of errors in timing of simulated peaks**

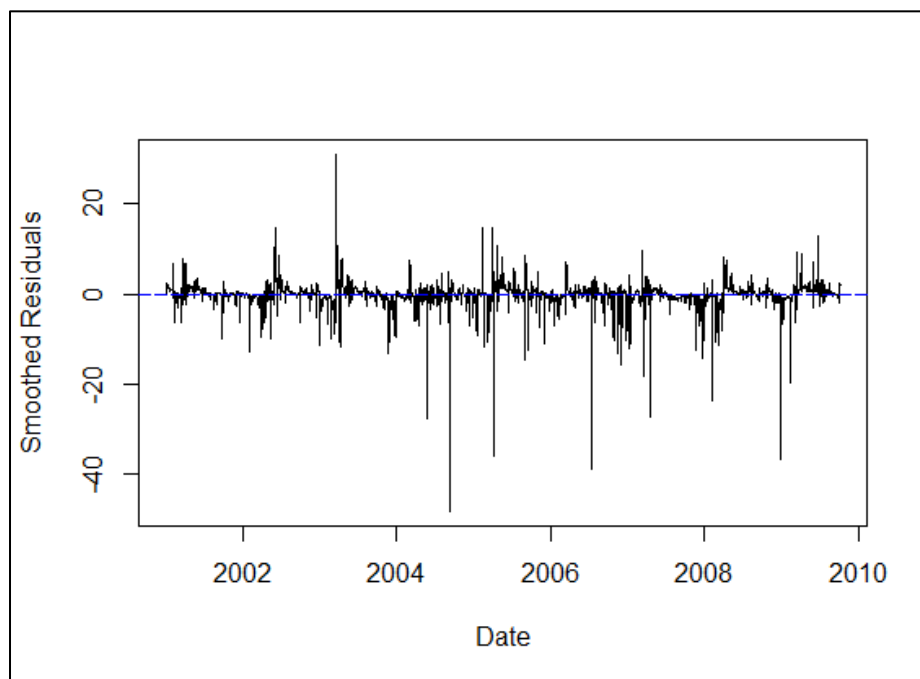
The plotting of residuals is very often a useful diagnostic to examine, and in hydrology this is no exception. The `flow.residuals()` function will calculate the residuals and plot them for you. An important note is that the default for this function is to use a smoothing of 3-point moving average; this is useful in removing noise to look at the plot, however one should be careful to not use the magnitude of the residuals. Compare the plots generated for the default settings and no smoothing at all (`ma.smooth=0`) (also note the use of the `winter.shading` argument). Also feel free to try out a much larger smoothing, e.g. `ma.smooth = 10`.

```
RavenR::flow.residuals(sim,obs)
```

```
RavenR::flow.residuals(sim,obs,ma.smooth = 0,winter.shading = F)
```



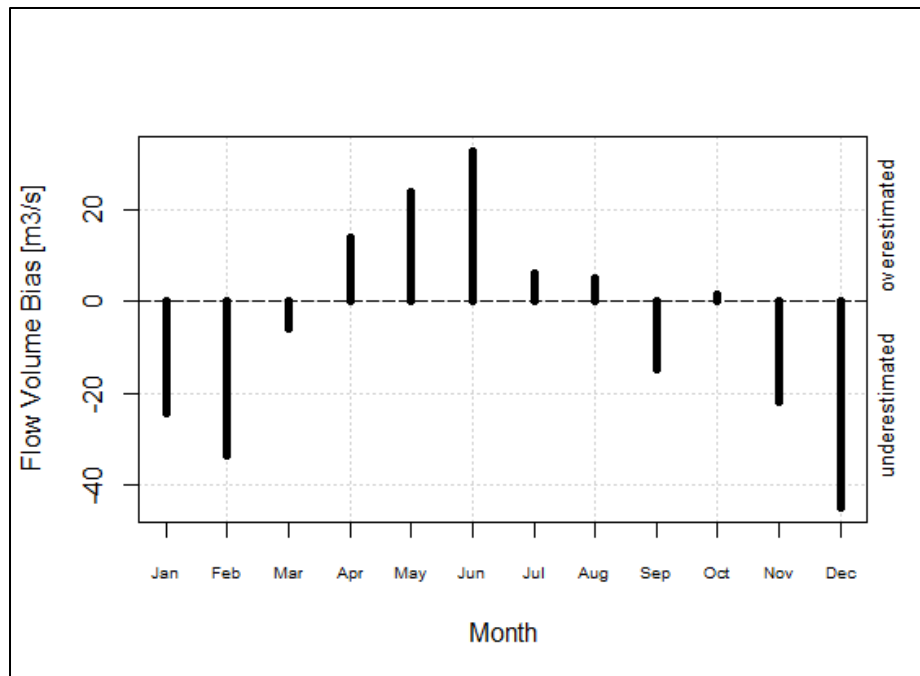
**Figure 20. Residual time series with default options**



**Figure 21. Residual time series without smoothing or winter shading**

Finally, a monthly volume bias can be generated for the model output. This can be normalized or unnormalized, depending on the analysis desired (default is to normalize the volume biases). Again, labels are included to aid interpretation, but can be removed if desired.

```
RavenR::monthly.vbias(sim, obs)
```



**Figure 22. Monthly average flow bias plot**

Note that many of the diagnostic functions above return the calculated values that are plotted, and can also be run without generating a plot. This is useful in case the user wishes to perform the calculation but plot the diagnostics differently, or perhaps overlay the plots from different models, for example.

## 10.2 Automating Flow Diagnostics

With a little bit of coding, the diagnostics above can be wrapped into a loop and run automatically for multiple basins or models. The following is an example which declares the stations to run diagnostics for, generates the name of the subbasin to extract using `hyd.extract`, runs the diagnostics, and saves each plot to the drive. Experiment with this loop to see if you can modify which diagnostics are saved. As a bonus, try to store the data instead of plotting, and write it to a file instead.

```
myhyd <- hyd.read('model_output/Hydrographs.csv') # read in previous
Hydrographs file
stns <- c(15,52,97) # specify list of stations to plot diagnostics for
for (i in stns) {
  flows <- hyd.extract(sprintf('sub%i',i),myhyd)
  sim <- flows$sim

  # plot Hydrograph
  png(filename=sprintf('Hydrograph_%i.png',i))
  hyd.plot(sim,zero.axis=F)
  dev.off()
```

```

# create spaghetti plot
png(filename=sprintf('Spaghetti_Plot_%i.png',i))
flow.spaghetti(sim)
dev.off()

# plot cumulative flow
png(filename=sprintf('Cumulative_Volume_%i.png',i))
cum.plot.flow(sim)
dev.off()
}

```

The plots saved using the `png()` function will appear in the current working directory, which can be checked in R with the `getwd()` command.

## 11 Reservoir Stages File

The ReservoirStages file can be read in similarly to the Hydrographs file, and processed almost identically. There are currently no tools developed specifically for diagnostics of stage, however many of the same ones could like be used. As an example, a ReservoirStages file is read in and a stage plot is created for the simulated stage values.

```

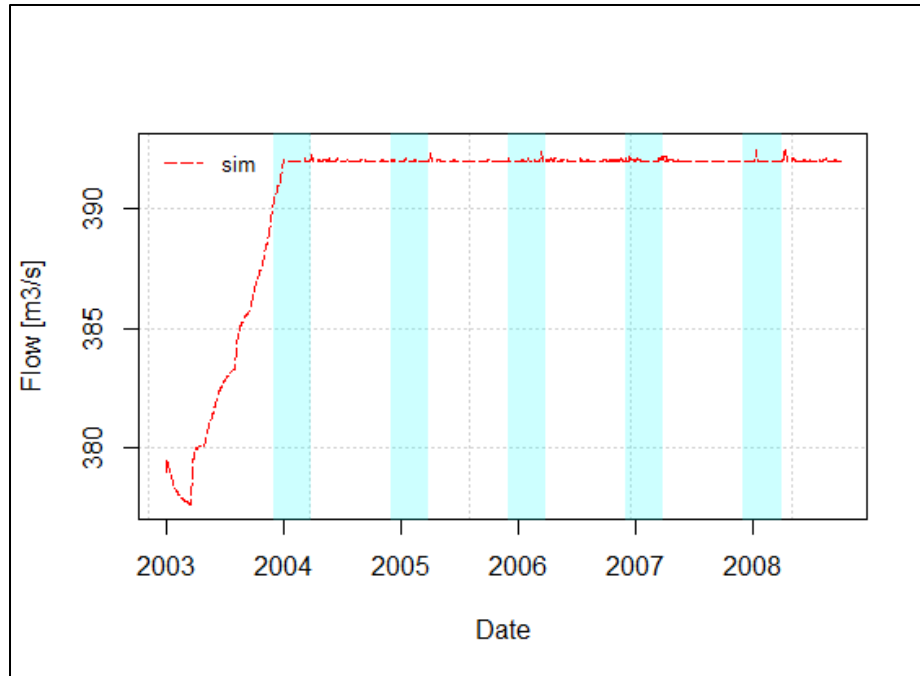
myres <- RavenR::res.read('model_output/ReservoirStages.csv') # read in
ReservoirStages files

sub11 <- RavenR::res.extract('sub11',myres) # extract simulated stage
values at subbasin 11

# check the first few stage values
head(sub11$sim)

# plot the stage at subbasin 11
RavenR::res.plot(sub11$sim,zero.axis = F)

```



**Figure 23. Reservoir stage plot for simulated and observed flows**

## 12 Ostrich Files

A handful of functions are supplied for reading the `OstModel0.txt` file produced from a single-objective Ostrich run. These are limited but potentially useful for simple analysis of calibration results. For example, the file can be read in as:

```
ost <- RavenR::Ost.read('sample_OstModel0.txt')
head(ost)
```

The best parameter set can be quickly found with the function `Ost.bestparams`:

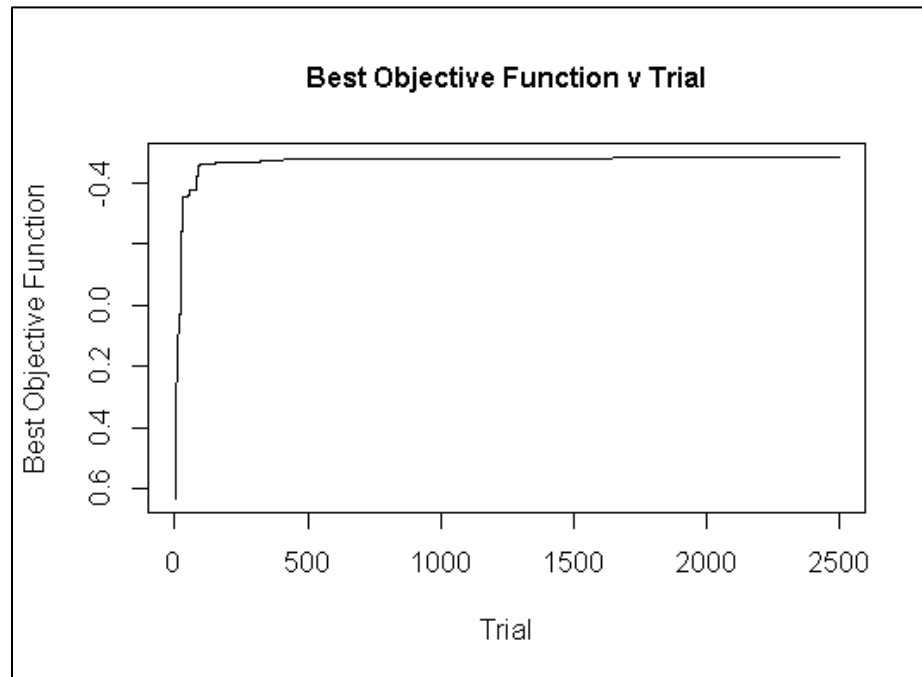
```
best.params <- RavenR::Ost.bestparams(ost,num.metrics=2,best.num = 1)
```

The same function can also be used to lift the top 100 (or any number, or top percentage) best parameter sets from the file. These could then be plotted to see the effect of that parameter on the objective function, in loose terms.

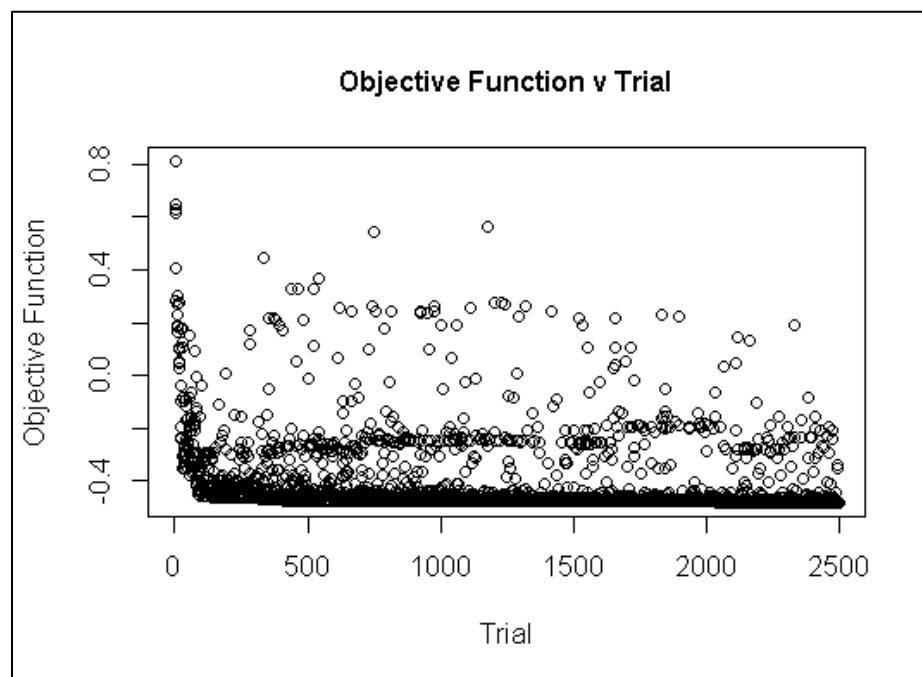
```
best.params.100 <- RavenR::Ost.bestparams(ost,num.metrics=2,best.num=100)
plot(best.params.100$X.A0TLZP.) # all static
plot(best.params.100$X.V0FLAS.) # slightly varied
```

A number of plots of the objective function versus trial (or iteration) can be plotted as well, which are helpful in determining visually whether the calibration has converged on a successful solution. The best objective function with trial, and a more scattered objective function with trial, are available for plotting.

```
RavenR::Ost.plot(ost, 'boft')  
RavenR::Ost.plot(ost, 'oft')
```



**Figure 24. Best objective function during calibration trials**



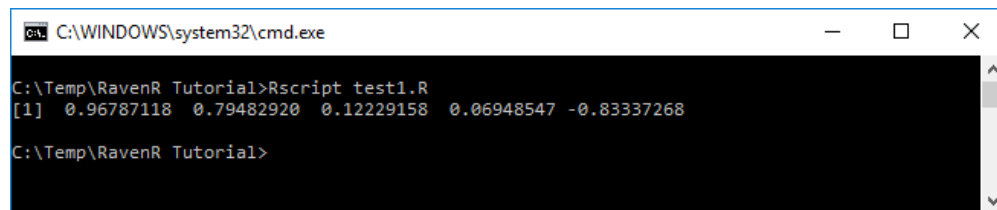
**Figure 25. Objective function during calibration trials**

Finally, a marginal parameter sensitivity from the calibration results can be done by simply plotting the parameter value with objective function value for a given parameter. As a review of saving figures in R,

the following will save the sensitivity plot for the first ten parameters in the model, saving the plots to the working directory.

## 13 Running R from Command Line

A useful exercise is to run a given R script from the command line (the same syntax is used to run R files from a batch file in Windows, or a bash script in Linux). On most systems, if R is installed then the Rscript command will be setup as well (see below if Rscript is not working). Simply use the command Rscript followed by the R filename to run the .R file. Do this now for the test1.R file, which will print out five random values generated from a standard normal distribution.

A screenshot of a Windows Command Prompt window. The title bar shows 'C:\WINDOWS\system32\cmd.exe'. The command prompt shows the directory 'C:\Temp\RavenR Tutorial' and the command 'Rscript test1.R'. The output is a single line of five random values: '[1] 0.96787118 0.79482920 0.12229158 0.06948547 -0.83337268'. The prompt then returns to 'C:\Temp\RavenR Tutorial>'.

```
C:\WINDOWS\system32\cmd.exe
C:\Temp\RavenR Tutorial>Rscript test1.R
[1] 0.96787118 0.79482920 0.12229158 0.06948547 -0.83337268
C:\Temp\RavenR Tutorial>
```

Figure 26. Screenshot of running R script from DOS

This can be done to easily use R scripts in calibration or other jobs. Note that it is worth testing this functionality first and ensuring that the libraries are read properly in this way, which can cause the script to fail, even if it runs fine in a GUI such as RStudio.

### 13.1 Rscript is not working

If the Rscript command does not work, you may need to set it up by associating Rscript with the Rscript.exe executable in the installed R folder. The following will detail how to associate the path on a Windows operating system.

1. Find the Rscript.exe executable in the installed folders, the path should be similar to  
C:\Program Files\R\R-3.3.2\bin\x64.
2. Open a Command Prompt and enter the following command to append the path containing Rscript.exe to the global PATH variable, Set PATH=%PATH%; C:\Program Files\R\R-3.3.2\bin\x64;
3. Alternatively, go to the Environment Variables page in your settings, and manually add the path in Step 1 to the PATH variables.
4. Close any command prompts to reset, open a new command prompt, and Rscript should now function as expected.

There are a few alternatives to the Rscript approach, such as the R CMD BATCH command or associating a custom file type with the R executable directly, such as .Rexec. However, Rscript is the suggested method for general use.

## 14 Appendix – Tutorial R Code

```
1. ### install RavenR and other packages ----
2.
3. # not installed yet? Install from menu by Tools > Install Packages > browse to find .tar.gz, or...
4. install.packages('RavenR_1.0.2.tar.gz', repos = NULL, type="source")
5.
6. # load in RavenR library
7. library(RavenR)
8.
9. # list all functions found in the RavenR library
10. # wow lots of functions!
11. ls("package:RavenR")
12.
13. ### getting help in R ----
14. # help can be found in a few ways
15. ?hyd.plot # lookup a single function
16. ??hyd.plot # search for this word in help pages
17. ??hydrograph# search help pages for functions related to 'hydrograph'
18. ?RavenR::hyd.plot # search for this function in the RavenR package specifically
19. hyd.plot # quickly look at the function script itself
20.
21. ### let's take a look at the forcing functions ----
22. # read in the ForcingFunctions file; store into 'myforcings'
23. myforcings <- forcings.read('model_output/ForcingFunctions.csv')
24.
25. # what is the architecture like?
26. # data stored in $forcings; use head() to view first 5 lines
27. head(myforcings$forcings)
28. # units stored in $units (don't need head function, only one row of units)
29. myforcings$units
30.
31. # the data is conveniently already processed as a time series, plot the PET
32. plot(myforcings$forcings$PET,type='h')
33.
34. # make a nicer rain and snow plot
35. plot(lubridate::date(myforcings$forcings$rain),myforcings$forcings$rain,main='Precipitation Plot',ylab='Precipitation [mm/d]',
36.       xlab='Date',col='blue',type='h')
37. # define a transparent black colour using RavenR::addColTrans
38. my.colour <- addColTrans('black',150)
39. # how do I use this function?
40. ?addColTrans
41. # add additional lines to the same plot using lines()
42. lines(lubridate::date(myforcings$forcings$snow),myforcings$forcings$snow,col=my.colour,
43.       type='h')
44. # add a legend to the same plot using legend()
45. legend('topleft',legend=c('rain', 'snow'),col=c('blue',my.colour),lty=c(1,1),inset=0.01,
46.       cex=0.9)
47.
48. # find all the plot arguments here
49. ?plot
50.
51. # period based functions are available in the xts package
52. # find the max monthly snowfall in the simulation period?
53. ?xts::apply.monthly
54. # store time series of maximum monthly snowfalls
55. temp <- apply.monthly(myforcings$forcings$snow, max)
56. plot(temp)
```



```

55. ind.max <- which.max(temp) # find index of maximum monthly max value
56. sprintf("max monthly snowfall is %.2f mm, occurring in %s",max(temp),date(temp[ind.max,
  ]))
57.
58. # analysis of total snowfall in each water year
59. # reports the ending date of each water year and the snowfall amount in mm
60. # the apply.wyearly follows the template of this type of function
61. apply.wyearly(myforcings$forcings$snow, sum)
62.
63. # date.end      snow
64. # 1 2004-10-01 184.1346
65. # 2 2005-10-01 223.7970
66. # 3 2006-10-01 219.0879
67. # 4 2007-10-01 172.7210
68. # 5 2008-09-30 324.2193
69.
70. ### let's read in the watershed storage file -----
71. # for convenience, store this directly as the $watershed
72. mywshd <- watershed.read('model_output/WatershedStorage.csv')$watershed
73. head(mywshd) # check the first five rows
74.
75. # plot the Soil[0] storage
76. plot(mywshd$Soil.Water.0)
77. # add Soil[1] storage
78. lines(mywshd$Soil.Water.1,col='blue',lty=5)
79.
80. # plot the soil storages with appropriate y axis limits
81. plot(mywshd$Soil.Water.0, ylim=c(0,max(mywshd$Soil.Water.0,mywshd$Soil.Water.1)))
82. lines(mywshd$Soil.Water.1,col='blue',lty=5)
83.
84. # add a legend
85. legend(x='topleft',legend=c('Soil[0]','Soil[1]'),lty=c(1,5),col=c('black','blue'))
86.
87. ### read in the WatershedMassEnergyBalance ----
88. mywshd.meb <- watershedmeb.read('model_output/WatershedMassEnergyBalance.csv')
89. head(mywshd.meb$watershedmeb,7) # preview first 7 rows of data
90.
91. # show the from and to in the file
92. rbind(mywshd.meb$from,mywshd.meb$to)
93.
94. ### read in custom output files ----
95.
96. # read in custom output rainfall file, store into 'custom1'
97. custom1 <- custom.read('model_output/RAINFALL_Monthly_Average_BySubbasin.csv')
98. head(custom1)
99.
100. # Create a basic plot with a custom title
101. plot.title <- sprintf('Precipitation in subbasin %s',colnames(custom1)[21])
102. plot(custom1[,21],type='h',main=plot.title) # precip in subbasin 21
103.
104. ### read in Hydrograph file ----
105.
106. # read in first hydrograph file - store all flow series
107. myhyd <- hyd.read('model_output/Hydrographs.csv')
108.
109. # extract specific subbasin data from subbasin 106
110. # extracts all possible simulated flows, observed flows, inflows
111. sub106.flow <- hyd.extract('sub106',myhyd)
112.
113. # note that in this subbasin, there are no observed flows or inflows, object is null
114. # trying to plot them will result in an error

```

```

115. plot(sub106.flow$sim)
116. plot(sub106.flow$obs) # will result in an error
117. plot(sub106.flow$inflow) # will result in an error
118.
119. flow <- sub106.flow$sim
120.
121. # create spaghetti plot of simulated flows
122. flow.spaghetti(sub106.flow$sim)
123.
124. # ooh that's a nice plot, let's save it to file
125. png(filename='my_spaghetti_plot.png')
126. flow.spaghetti(sub106.flow$sim)
127. dev.off()
128.
129. # read in the second hydrographs file, need observed flows for many of the diagnostics

130. # this is the Irondequoit model, one of the models from a Raven tutorial
131. myhyd2 <- hyd.read('model_output/run1_Hydrographs.csv')
132. head(myhyd2$hyd) # check hydrograph data for column names
133.
134. # extract flow for the Irondequoit outlet; store into Iron.flow
135. Iron.flow <- hyd.extract('Irondequoit',myhyd2)
136. # can also extract precip series from the hydrograph object
137. precip <- hyd.extract('precip',myhyd2)$sim
138.
139. # assign sim and obs, avoid typing out the extension
140. sim <- Iron.flow$sim # assign sim
141. obs <- Iron.flow$obs # assign obs
142.
143. # create a nice hydrograph
144. hyd.plot(sim,obs)
145.
146. # create a hydrograph with precip as well
147. hyd.plot(sim,obs,range.mult=1.5,precip=precip)
148.
149. # create a hydrograph with precip as well for a specific subperiod
150. # always define time period as yyyy-mm-dd
151. hyd.plot(sim,obs,range.mult=1.5,precip=precip,prd="2003-10-01/2005-10-01")
152.
153. # check the cumulative flows
154. cum.plot.flow(sim,obs)
155.
156. # plot cumulative flows for specific period
157. # period is not built in to this function, but can be done for any time series object

158. # just have to make sure the period supplied is sensical
159. # the function will not plot beyond the water years, defined as October 1st
160. myperiod <- "2003-10-01/2005-10-10"
161. cum.plot.flow(sim[myperiod],obs[myperiod])
162.
163. # plot the flow scatterplot, produce an R2 metric
164. flow.scatterplot(sim,obs,add.r2=T)
165.
166. # check annual volume
167. annual.volume(sim,obs,add.r2 = T)
168.
169. # check annual peaks, set to zero on axis extents
170. annual.peak(sim,obs,axis.zero=T)
171. # add a title to this plot
172. title('Peak Flow Check')
173.

```

```

174. # check peak errors
175. annual.peak.error(sim,obs)
176.
177. # check peak event
178. annual.peak.event(sim,obs,add.r2=F)
179.
180. # check peak event errors
181. annual.peak.event.error(sim,obs)
182.
183. # check error in peak timing
184. annual.peak.timing.error(sim,obs)
185.
186. # check residuals
187. # default with moving average smoothing shading of winter months
188. flow.residuals(sim,obs)
189. # plot with more smoothing than the default 3
190. flow.residuals(sim,obs,ma.smooth=10)
191. # turn off the smoothing and winter shading
192. flow.residuals(sim,obs,ma.smooth = 0,winter.shading = F)
193.
194. # check the monthly volume bias; normalizes by default
195. monthly.vbias(sim,obs)
196. # check unnormalized monthly volume biases; see the larger volumes in spring freshet
197. monthly.vbias(sim,obs,normalize = F)
198.
199. # make a loop out of some diagnostics for multiple stations ----
200. myhyd <-
    hyd.read('model_output/Hydrographs.csv') # read in previous Hydrographs file
201. stns <- c(15,52,97) # specify list of stations to plot diagnostics for
202. for (i in stns) {
203.   flows <- hyd.extract(sprintf('sub%i',i),myhyd)
204.   sim <- flows$sim
205.
206.   # plot Hydrograph
207.   png(filename=sprintf('Hydrograph_%i.png',i))
208.   hyd.plot(sim,zero.axis=F)
209.   dev.off()
210.
211.   # create spaghetti plot
212.   png(filename=sprintf('Spaghetti_Plot_%i.png',i))
213.   flow.spaghetti(sim)
214.   dev.off()
215.
216.   # plot cumulative flow
217.   png(filename=sprintf('Cumulative_Volume_%i.png',i))
218.   cum.plot.flow(sim)
219.   dev.off()
220. }
221.
222. ### read in the ReservoirStages.csv file ----
223. myres <-
    res.read('model_output/ReservoirStages.csv') # read in ReservoirStages files
224. sub11 <-
    res.extract('sub11',myres) # extract simulated stage values at subbasin 11
225.
226. # check the first few stage values
227. head(sub11$sim)
228.
229. # plot the stage at subbasin 11
230. res.plot(sub11$sim,zero.axis = F)
231.

```

```

232. ### check Ostrich files ----
233.
234. # read in Ostrich file
235. ost <- Ost.read('sample_OstModel0.txt')
236. head(ost)
237.
238. # get the best parameter set from the calibration run
239. best.params <- Ost.bestparams(ost,num.metrics=2,best.num = 1)
240.
241. # get the top 100 best parameters from the calibration run
242. best.params.100 <- Ost.bestparams(ost,num.metrics=2,best.num=100)
243. # plot top 100 params for several parameters
244. plot(best.params.100$X.A0TLZP.) # all static
245. plot(best.params.100$X.V0FLAS.) # slightly varied
246.
247. # plot the best objective function vs trials
248. Ost.plot(ost,'boft')
249.
250. # plot the objective function with trial
251. Ost.plot(ost,'oft')
252.
253. # plot the parameter sensitivity for first five parameters
254. for (i in 1:10) {
255.   png(filename=sprintf('param_sensitivity_%i.png',i))
256.   Ost.plot(ost,'psa',param.index=i)
257.   dev.off()
258. }

```