Jarle Brinchmann
Leiden Observatory
jarle@strw.leidenuniv.nl

Homework set

The homework will explore some of the topics discussed in the lectures. We start out with an exploration of how you can estimate parameters of a density - this is the parametric density estimation which was mentioned but not discussed in detail in the lecture. It is mostly a theoretical problem.

In the second problem you will explore the use of different density estimators to determine the distribution of low-mass planets and in the third problem you will use the same dataset to familiarise yourself with the bootstrap method to estimate uncertainties.

The final problem is on cross-validation and there is an accompanying "problem" which is not to be handed in, but which is there for reference to see how cross-validation is carried out in IDL and R.

Most of these problems are likely to be easier to do in R than in IDL or other languages, but you can use what you prefer of course.

When you work with the data below you might want to be able to create plots with different colours. This is rather different in IDL and R so it might be useful to have a very brief introduction to this here:

R

In R there are several different graphical systems, and the details of plotting differ somewhat but the colour handling is fairly standard. Colours can be referred to by name and to see an illustration of the possible colours you can do:

```
> colors()
```

which provides a list of all known colour names. It is not very enlightening! To get a more visual impression you can install the DAAG package (install.packages(DAAG)) or simply go to

In addition to knowing the names of the colours it is useful to know how to use them. Most plotting commands has an argument `col` which is used to specify the colour (I will use the exoplanets table for plotting, it is used repeatedly in the following as well):

```
> x = log10(e$St_Dist)
> y = e$St_Mass
> plot(x, y, pch=19, col="blue")
```

You can also provide this colour in other formats:

```
# Using Red-Green-Blue notation in hexadesimal:
> plot(x, y, pch=19, col="#0000FF")
# Or using the rgb function (here values go from 0 to 1):
> plot(x, y, pch=19, col=rgb(0, 0, 1))
```

In some situations you might get a lot of overlapping datapoints and then you might want to have partially transparent points. This is supported on some devices and is provided as an **alpha** value in the rgb function (last argument):

```
> plot(x, y, pch=19, col=rgb(0, 0, 1, 0.5))
```

You can also set the colour using Hue-Saturation-Value with the hsv function which also allows for alpha values, and Hue-Chroma-Luminosity with the hcl function. These are often preferred colour spaces for some application but not crucial for our work here.

You can also give the colours so that each point has a unique colour. This is a tad more involved but very useful. Imagine for instance that you want to make a plot of stellar distance, stellar mass as before but that you want to colour each point by the metallicity. The first step is then to create a colour vector. This can either be between 1 and 255 to give an argument to col or between 0 and 1

and given as an argument to rgb/hsv/hcl etc. Thus in both cases it is useful to have a scaling function:

```
> scl = function (x) {
+      mn = min(x, na.rm=TRUE)
+      mx=max(x, na.rm=TRUE)
+      dx = mx-mn
+      return( (x-mn)/dx)}
```

which scales a vector x to lie between 0 and 1. This can then be used to make colour vectors:

```
> z = e$St_Metal
> z.scaled = scl(e$St_Metal)
> plot(x, y, col=z_scaled*255)
```

but often it is necessary to clip the colour-scale. One way to do this is:

```
> z = e$St_Metal
> z[z < -0.3]=-0.3
> zc = scl(z)
> plot(x[ok], y[ok], pch=19, col = rgb(zc[ok], 0, 1-zc[ok], 0.5))
```

Ok, that should get you going but there are many other aspects. For images you want the `palette` function, but you could also look at the ColorBrewer (http://colorbrewer2.org/) - and interface exists for R (`library(RColorBrewer), help(brewer.pal);`)

Finally, it can be useful to know how to make hard-copy plots in R:

```
> pdf(file='myplot.pdf')
> plot(x[ok], y[ok], pch=19, col = rgb(zc[ok], 0, 1-zc[ok], 0.5))
> dev.off()
```

Similarly you can create PNG, JPG and quite a few other outputs - see `help(pdf), help(png)` etc.

IDL

For IDL there is less variation in the plotting packages but there is also somewhat more inflexible colour specifications. To plot colour on the screen you can use a similar notation to R:

```
IDL> x = alog10(e.st_dist)
IDL> y = e.st_mass
IDL> plot, x, y, psym=6, /nodata
IDL> oplot, x, y, psym=6, color='ff0000'x
```

If you don't have the first call with /nodata you will get a blue plot box as well. Probably don't want that. But worse is the fact that this will *not* work if you make a Postscript plot. You might instead want to use the getcolor function in the Coyote IDL library (www.dfanning.com), or my get_colour_by_name function (see the course web-site):

```
IDL> oplot, x, y, psym=6, color=getcolor('Beige')
IDL> oplot, x, y, psym=6, color=get_colour_by_name('ForestGreen')
```

Note that the getcolor function has a relatively small number of names, while the get_colour_by_name function supports all colours in the X11 rgb.txt file (see http://sedition.com/perl/rgb.html). You can also use the colour brewer data in IDL - see http://www.dfanning.com/color_tips/brewer.html for details.

It is also good to be able to plot points with different colours. This is not supported by plot directly but plots does allow you to give a vector. As a convenience I have written a color_points_plot routine which you can find on the course web page. This allows you to create a coloured plot as you saw for R but with a bit easier interface:

```
IDL> loadct, 1
IDL> z = e.st_metal
IDL> color_points_plot, x, y, z, min=-0.5, max=1, /scale, table=1,
  psym=6
```
should plot a coloured display using colour table 1 as the color lookup table while adding a scale on the side.

That was for the direct graphics system, if you use the object graphics system you have some more flexibility but at a cost of a much more complex plotting interface. I don't find the object graphics system particularly useful for exploring data so will not detail this here - there is plenty of information in the IDL documentation.

I will close with a brief discussion of how to make hard-copies of plots in IDL. It is marginally more complicated than in R perhaps, but not much so. The easiest is to use the ps_on and ps_off routine I have put on the course web site:

```
IDL> ps_on, 'myfile.ps', /color
IDL> plot, x, y, color=get_colour_by_name('red')
IDL> ps_off
```
will create a portrait one. You can also change the aspect ratio

```
IDL> ps_on, 'myfile.ps', /color, aspect=0.5
IDL> plot, x, y, color=get_colour_by_name('red')
IDL> ps_off
```
but then you might want to change the length:

```
IDL> ps_on, 'myfile.ps', /color, aspect=0.5, xlen=10
IDL> plot, x, y, color=get_colour_by_name('red')
IDL> ps_off
```

1. Estimating densities from data

In the lecture we focused on non-parametric estimators of densities, but we did cover the necessary background from parametric density estimation, so let us explore one very fundamental density estimator: The Gaussian distribution, and how one goes about estimating the properties of this distribution.

For later reference and to define our notation, let us write the Gaussian distribution as

$$N(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2}\frac{(x-\mu)^2}{2\sigma^2}}$$

where σ is the standard deviation (spread) of the distribution and μ is the location of the peak of the distribution.

Assume that you have a set of observations, $\{x_i\}$ and assume that you believe that a Gaussian is a good representation of this distribution (because you plotted it or for physical reasons). You therefore need to estimate the values for μ and σ, we will refer to these as $\hat{\mu}$ and $\hat{\sigma}$ to indicate that they are not the true values, just estimates.

a) Explain why the likelihood of the whole sample $\{x_i\}$ is given by

$$p(\{x_i\}|\mu,\sigma) = \prod_{i=1}^{M} N(x_i|\mu,\sigma)$$

where M is the number of observations.

b) Take the logarithm of the equation and show that you get

$$\ln p(\{x_i\}|\mu,\sigma) = -\sum_{i=1}^{M} \frac{(x_i-\mu)^2}{2\sigma^2} - \frac{M}{2}\ln 2\pi\sigma^2$$

The first term after the equal sign is what you often will see as the χ^2 quantity that one usually minimises when fitting models to data. The reason for its ubiquitousness is this close link to the Gaussian distribution - if the uncertainties are

c) We now want to find the maximum likelihood estimate of μ. Maximise the equation for $p(\{x_i\}|\mu,\sigma)$ and show that the maximum is reached for

$$\hat{\mu} = \frac{1}{M}\sum_{i=1}^{M} x_i$$

d) Likewise show that

$$\hat{\sigma} = \frac{1}{M}\sum_{i=1}^{M} (x_i-\hat{\mu})^2$$

e) Calculate the expectation value of $\hat{\mu}$, (that is, $E[\hat{\mu}]$) and show that is equal to μ. This means that it is an *unbiased* estimator of the mean. It is possible to do the same for $\hat{\sigma}$ but it is a bit of arithmetic, and it turns out that $E[\hat{\sigma}] = (M-1)\sigma/M$ which means that $\hat{\sigma}$ always underestimates the true width of the Gaussian (this is not true if the mean is known though).

2. Exoplanets - the abundance of low-mass planets

Go back to the exoplanet dataset you have worked with earlier. The aim here is to explore the connection between stellar metallicity and planet mass as well as simply studying the abundance of planets as a function of mass.

You will need to calculate the median or mean absolute deviation for a value. Simple routines for this in IDL and R are:

In R:

```
MAD = function(x) {
   # Calculate the median absolute deviation
   # We scale this by 1.4826 because it
   ok=is.finite(x)
   m = median(x[ok])
   return(median(abs(x[ok]-m)))
}
AAD = function(x) {
   # Calculate the mean/average absolute deviation
   ok=is.finite(x)
   m = median(x[ok])
   return(mean(abs(x[ok]-m)))
}
```

and the same functions in IDL:

```
function mad, x, status=status, scale=scale
   ;;
   ;; Return the median absolute deviation of x
   ;;
   if (n_elements(scale) eq 0) then scale = 1.4826
   ok = where(finite(x) eq 1, n_ok)
   m = median(x[ok])

   return, scale*median(abs(x[ok]-m))
end

function aad, x, status=status
   ;;
   ;; Return the mean absolute deviation of x
   ;;
   ok = where(finite(x) eq 1, n_ok)
      m = median(x[ok])
   return, mean(abs(x[ok]-m))
end
```

Note that we scale the MAD estimate because it is really an estimate of the inter-quartile range, ie. the range between the 25% and 75% percentile for a symmetric distribution because P(|x-med(x)|<MAD) = 0.5. Thus to compare it to the standard deviation which is between the 16th and 84th percentile we need to scale it up - the factor 1.4826 comes from the Gaussian distribution.

a) The first step is to examine the data to get a feel for the properties of the data. A useful first list of things to calculate are:

1. The median, mode and average of the data.

2. The standard deviation and the median absolute deviation.

3. The 0.25 and 0.75 quantiles as well as the range containing 95% of the points.

We want this for the stellar mass, stellar metallicity, planet mass and planet period at least. Which one of these are symmetric distributions? What is the number of planets with mass less than 0.1 M_{Jup}? [useful to know: To get the number of x satisfying a criterion you can do: `sum(x > 4)` in R and `dum=where(x gt 4, n); print, n` in IDL.]

b) Plotting the data is a useful complement to statistical summaries and can be quite enlightening. In the werkcollege we examined the use of conditioned plots. This is useful again here, you will probably find it useful to explore data using different conditioning variables. Choose your approach but your goal is to explore how the relationship between planet mass and planet period depends on metallicity and stellar mass. In particular you should explore the low-metallicity bin where the stellar metallicity is below -0.06 - where do planets that have low-mass central stars lie in this diagram?

It might also be useful to review the comments in the introduction to the problem set about colour and colouring points.

c) Now, let us move towards addressing one question: The abundance of low-mass planets. Create histograms of the mass of planets with masses < 1/10 M_{jup}. Start with a bin size of 0.01 and plot a histogram. You should see a peak at masses ~0.02 M_{Jup} with what appears to be a steady rise as you go down in mass?

Is it an artefact of the bin-size chosen? To test this create histograms with variable bin sizes and also kernel density estimates with different band-widths. Span the range 0.05 to 0.001 in bin sizes and similarly in bandwidths. What is your conclusion about the distribution of masses - does the number of planets appear to increase with decreasing mass? If so, what is your estimate of the derivative - ie. what is the rate of increase in the number of planets with decreasing mass? The extrapolation of these curves to Earth mass planets is one of the most keenly sought after numbers for the planning of future space missions to find habitable planets.

d) How does the results in c) change if you work with the logarithm of the mass instead of linear mass? Does this affect your conclusion about the extrapolation to Earth-mass planets?

3 Estimating the uncertainty on values - the bootstrap way

As we discussed in class, we often need to estimate densities. One that is of almost universal importance is the distribution of uncertainties around a value - without an estimate of the uncertainty of a value you get a scientific exploration is usually not very useful [although there are some cases where it is prohibitively difficult to do so].

When the uncertainties are close to Gaussian (true in many cases) then it is easy to estimate the uncertainties by estimating the standard deviation of the Gaussian (see problem 1). However in more complex situations we might know the uncertainties on our input data but the process to get to the results is complex and we do not know how to calculate the uncertainties on the end-results.

In this case it is very often useful to use the bootstrap technique. The procedure here is:

✓ Draw a new sample from your dataset **with replacement**. Call this sample X_i.

✓ Carry out your calculations using X_i and store the result as R_i.

✓ After a number of repetitions of this process, a common number is 999, carry out a statistical analysis of R_i. The distribution of R_i is then an estimate of the uncertainty distribution on your values.

As a simple example, let us try to estimate the uncertainty on the median using bootstrapping. The way to do this is easily illustrated using IDL (where x holds the quantity of interest):

```
n_boot = 999
meds = fltarr(n_boot)  ; R_i above
n_x = n_elements(x)
for i_boot = 0L, n_boot-1 do begin
   ;; Draw with replacement!
   I_set = floor(randomu(sss, n_x)*n_x)
   x_new = x[i_set]  ; X_i above
   meds[i_boot] = median(x_new)  ; R_i above
endfor
```

This assumes that your data have no uncertainty, if they do you could change the drawing of a new sample to also include a random element:

```
x_new = x[i_set] + randomn(sss, n_x)*dx[i_set]
```

Note that the devil is in the details here. The analysis of R_i (or meds in the example above), is a rather complex business. A complete (but somewhat advanced) discussion is given in Davison, A.C. and Hinkley, D.V. (1997) Bootstrap Methods and Their Application. Cambridge University Press. There is no implementation of their techniques in IDL to my knowledge but it is implemented in the `boot` library in R.

Doing bootstrapping in R you have two options: One is to use the comprehensive `boot` library and the other is to simply implement the same loop as above. The advantage of the `boot` library is that it does a much more careful job than the preceding example and I would strongly recommend that if you do bootstrap calculations in R you use on of the existing libraries for it to ensure your statistics are correctly carried out.

You need the boot library to make this work but it might not be installed. To install it type: `install.packages(boot)` inside R and it should be downloaded and installed.

As an example, here is how you do the preceding using the `boot` library:

```
# Calculate the median without taking into account uncertainties.
calc.median.noerr = function(d, inds) {
   median(d$x[inds], na.rm=TRUE)
}

r = boot(d, calc.median.noerr, 999)
plot(r)
```

and you can calculate confidence intervals using the boot.ci function:

```
boot.ci(r,
   type=c("norm", "basic", "perc", "bca"),
        conf=c(0.68, 0.9, 0.95))
```

where the argument to conf gives the percentile levels to calculate and there are different types of estimators available indicated with the type argument.

The one notable point is that you need to define a function that calculates the quantity you want to bootstrap, here called `calc.median.noerr` - this has to be given as argument to the `boot` function.

a) With those preliminaries out of the way, calculate the uncertainty estimate on the median stellar mass in the exoplanets table. Are the uncertainties symmetric around the median? [make sure you exclude the stars that have missing stellar mass estimates].

b) Repeat the calculation in a) for the standard deviation and mean of the stellar mass. If the data are distributed as a Gaussian you would expect that the uncertainty estimate on the mean should be mean+/-standard deviation (Problem 1). Is this what you find?

c) Finally, above we bootstrapped the estimate of a single value. As you can see this is pretty straightforward. But there is nothing in the bootstrap method that says it needs to be applied to single values instead. Your task here is to take either the IDL or the R program provided on the course web-site (`boot-stars.R` for R and `boot_stars.pro` for IDL) and modify them to calculate the bootstrap estimates on the mass density of low-mass planets following what you did in the previous problem.

Problem 4 - Cross-validation

As discussed in the lecture, cross validation is a very useful technique for assessing whether to make a more complex model or not. You do that by fitting/carry out a procedure using one set of data (training sample) and test it with another where the test sample typically is part of the full sample. In cross-validation it is common to take out 10%, fit with the rest and test with the 10% you put aside. It is a computationally intensive technique so there is by necessity some coding.

To help with doing this there is a step-by-step guide to cross-validation using the example sine curve I showed in the lecture after this problem. So I would first turn to that, go through it (maybe just reading it) and then come back and do this problem.

Ok. That done? Let's continue:

The example in the lecture (and below) might appear overly simplistic and too theoretical. However the cross-validation idea is a powerful one and quite useful when you have a choice of models and complex data. It is important to keep in mind that you should only use it when you have no overwhelming scientific reason to choose one over the other! As the following example will emphasise you should also not use it blindly - if you have plenty of data and no very clear relationship the power of the method might be limited.

Here we will use this technique to determine how reddening towards open clusters is related to their distance.

a) First get the catalog of Galactic open clusters by Dias et al (2002) ["Optically visible open clusters and Candidates (Dias+ 2002-2007)" on Vizier]. You need a CSV file for use with R. I will assume that you have loaded it into a variable `t` in R - the exercise can also be done with IDL.

b) Plot the data! We are particularly interested in the distance & E(B-V) values - these are in the keys Dist and E.B.V. It is useful to take these out and place in a different data.frame.

```
x = log10(t$Dist)
y = t$E.B.V
df = data.frame(x = x, y = y)
```

are there anything of note in the plot of x vs y? Are there any outliers?

d) You probably did not notice though that there are missing values - not every cluster has a known distance! We do not want to include missing data in our fit - you can try this and see what happens:

```
fit = glm(y ~ x, data=df)
```

This probably (hopefully!) failed with a warning about missing values. So you need to select a subset of valid data with distances > 0. A useful function is which: `ok = which(is.finite(x))` for instance.

You can then re-create the data frame using `x=x[ok]` and `y=y[ok]` in the `data.frame` call.

e) Now for the fit! Fit the relationship as a few polynomials (since you have already read the postlude below I don't need to explain this!)

```
fit1 = glm(y ~ poly(x, 1), data=df)
fit2 = glm(y ~ poly(x, 2), data=df)
```

Then calculate the cross-validation error as discussed below - you should use K=10 to reduce computing time! How does the validation error differ? Try also a polynomial with order 5 - what do you find?

f) If you are like me you found that a 5th order polynomial is a better description of the data than a first and second order one - does this make sense? Try the same but using log (E(B-V)) as well. Does this change the results? What about changing the subset to only include x > 2.5 and x < 4?

g) Based on f) choose one of the cases and calculate the validation error for one case as a function of the polynomial order. Plot the relationship between the polynomial order and the validation error- what does it tell you? What is your interpretation of this? Can you imagine ways to improve this? Did you use the right function?

OK, that's it!

A postlude: An introduction to cross-validation

THIS IS NOT TO BE HANDED IN!

In fact you do not need to do it at all but you will hopefully find it useful for Problem 4 (almost certainly actually!).

This show discussion introduces the technique of cross-validation - it is by nature computationally intensive so there will be coding below. And this problem set will use both R (where it is easy to do this) and IDL (where some programming is needed) so that you can see how it is done in practice.

First get the data-set. This is on the course web-site and is called `noisy-data.dat`. This is similar to the data used in the lectures in that it is sampled from a sine wave (sin(2π x) to be precise) with Gaussian noise added to it. We want to determine what order polynomial is the best description of these data.

You might want to refer to the `cross-validation.R` routine on the course web page for a coded-up implementation of this example.

We will start with R to get our hands dirty quickly:

a) The first step is to load the data - it is a space separated file (look at it!) with columns which give the row number, the x & the y value. It is written by R and hence you will find that if you wanted to access the file in, say, TOPCAT, then you might need remove the first column. The following command is a convenient way to do this on the command line in UNIX (NF gives the num-

ber of columns because the header line has only two columns which we want to keep, $0 refers to the whole line)

```
awk '{if (NF > 2) { print $2,$3} else { print $0}}' noisy-data.dat
```

Anyway, you can easily read it into R using `read.table`. I will assume you have read it into a variable `t`. Load it in and plot it - overplot sin(2π x).

b) The task now (a bit arbitrary) is to fit polynomials to these data and determine what complexity (order of polynomial) is the optimal choice. The first step is the learning step - we will use `glm` for this (`glm` is a slightly more general version of the `lm` function mentioned in the lecture). This can be done using:

```
> fit.deg1 = glm(y ~ poly(x, 1), data=t)
```

which tells `glm` to fit `y` (taken from the data frame in `t`) as a polynomial function of `x` where 1 is the order of the polynomial.

To evaluate this fit you can either get the coefficients by printing `fit.deg1` - or more generally use the predict function:

```
xvals = seq(0, 1, 0.01)
yvals = predict(fit.deg1, data.frame(x = xvals))
plot(t)
lines(xvals, yvals, col='red')
```

should predict yvals from the fit - notice that the input to predict is a data frame and the keys (x here) must have names that corresponds to the equation used when doing the fit earlier.

c) We now want to evaluate this fit. We do this using the `predict` function above and the test data in the `noisy-data-verification.dat` file on the course web-site. Read this in - calculate yvals using the `fit.deg1` value.

We then need to decide on a function that test the quality of our fit. Two possibilities are:

Mean Squared Error (MSE): $N^{-1} \sum (y - y_{pred})^2$ - normalised by the error if known

Median Absolute Deviation (MAD): median($|y-y_{pred}|$)

Try both these error functions on your data - do they give radically different results?

d) The absolute value of these error functions is of limited value (in the case of Gaussian noise the MSE can be interpreted as a χ^2 of a fit and hence as something related to the likelihood of the model). What is a lot more important here is that the relative values between different fits can be used to compare them various fits.

To test this, redo the fit done above for a polynomial of order 3. How does the MSE and MAD compare for this fit against the fit for the linear case? [The correct answer should be that MSE(linear fit) ~ 1.4 MSE (third degree fit)].

e) To carry out proper tests without a big validation sample as given here, we use cross-validation. In R this can be done quite easily using the `cv.glm` routine in the `boot` library. The procedure is as follows:

- Load the boot library: `library(boot)`

- Run a fit using `glm`. Store this in a variable called, say, `fit`.
- Call `cv.glm` to cross-validate.
- The return value from `cv.glm` contains a key delta. This is the output of the error function similar to what you calculated above - the default is MSE

In terms of code this translates to

```
> library(boot)
> fit = glm(y ~ poly(x, 1), data=t)
> validation = cv.glm(t, fit)
> validation$delta
```

The output of this can now be compared to your calculation above. How does the MSE you calculated above compare to the delta values returned from `cv.glm`? There are two delta values returned actually, the first is the one that you calculated above. The second is a corrected delta value with less bias[1] than the straight MSE but in most cases they are quite similar to each other - and probably somewhat different from the MSE you calculated above!

If you add K=10, say, in the cv.glm call you can do a 10-fold cross-validation, often the recommended choice - try this and see what you get. In this case you should really repeat it many times because the results of a particular run will depend on the random sampling of the data done by the routine. For the rest of the problem do not specify K - in that case there is no need to repeat the call many times but it has the disadvantage that if there is the odd outlier in your data you might have unstable results.

Now. The goal (which you might have forgotten by now!) is to determine what order polynomial is the best representation of your data. To do this you need to carry out the procedure above for a number of polynomial orders (do order 1 to 8), keep track of the delta values and plot polynomial order against delta.

What order polynomial gives the best representation?

Did you notice any problems with this process - could you extend this to complex fitting scenarios do you think?

f)

Doing the same in IDL.

Ok, so the above was done in R and it is quick but it hides all the details. Doing the same in IDL gives us more insight into what happens (you can of course also do it in R the same way!). The process then divides into the following steps:

- Read in the data - the readcol routine in the astronomy library is good for this here.
- To test you might want to run a simple fit similar to the above.
- You now need to divide the sample into K random subsets. Loop over these subset each time leaving one out, do the fit on the rest and calculate the MSE of the left-out subset.

In practice we can use the poly_fit routine to do the fitting. Thus the first steps are easily done:

```
readcol, <LEARNING DATA FILE>, dum, x, y, format='(A,F,F)'
```

[1] This is discussed in more detail in “Bootstrap methods and their Application” by Davison & Hinkley - in section 6.4. This book is a general book on bootstrap methods but the notation can be a bit challenging for non-statisticians.

```
readcol, <VALIDATION DATA FILE>, dum, xt, yt, format='(A,F,F)'
res = poly_fit(x, y, 3, yfit)
ypred = poly(xt, res)
mse = mean((yt-ypred)^2)
```

Would do this. Drawing K random subsets is more tricky perhaps. But it is not too tricky: The trick is to first draw N random numbers, where N is the number of learning data points.

```
IDL> r = randomu(seed, n_elements(x))
```

Then you sort these random numbers

```
IDL> si = sort(r)
```

and use this sorted array as indices into x & y:

```
x_r = x[si]
y_r = y[si]
```

Why does this work?

Then the way to proceed is to create a loop over this new sample of x_r in steps of K. This will define one "current" group - this is your test sample and you use this to do the validation. Ie. for a sample size of 2 you could do:

```
i_group = 0
indices = lindgen(n)
for i=0, n, 2 do begin
   ;; Why does the following work & how would you modify it for K
  groups?
    use = where(indices lt i_group or indices gt i_group + 2-1)
    res = poly_fit(x_r[use], y_r[use], 3)
     i_group = i_group+2
endfor
```

Of course this code snippet does not calculate the MSE as done above, nor does it store the output - implement this.

How do your results compare to the R version above?