Statistics with MATLAB®/Octave

Jeremiah Mans

December 18, 2017

For use with PHYS3605W (Modern Physics Laboratory) at the University of Minnesota

1 Introduction

This manual serves as a guide to the use of MATLAB® or GNU Octave for statistical data analysis and plotting for the PHYS3605W course at the University of Minnesota. For the purposes of the document, I will generically refer to MATLAB. Octave is the open-source tool from the GNU project which shares most of the basic functionality of MATLAB and many of the examples presented will work with Octave as well as MATLAB.

For all the examples below, the text typed by the user is shown as **`bold typewriter font`** and the response from MATLAB is shown as `regular typewriter font`.

2 Basic operations

First, we want to set up the display to work nicely. This is one of few points where Octave and MATLAB are different. In Octave, we can set the precision of the display, while in MATLAB we cannot. As a result, we have different recommendations depending on which you are using.

- **For Octave**

  ```
  >> format shortG
  >> output_precision(6)
  ```

- **For MATLAB**

  ```
  >> format longG
  ```

2.1 Mean, median, variance, and histograms

MATLAB provides a strong set of tools for working efficiently with vectors and matrices. Indeed, MATLAB considers to be the same type of object – a vector is a matrix with only one column or one row.

Let us consider the data set on accelerator radius measurements from Table 1 of the statistics note. We can load this into MATLAB with the following command:

```
>> r = [ 3.10014, 3.10010, 3.10010, 3.09997, 3.09977, 3.10010, 3.10003, 3.10004, 3.09997,
         3.09976, 3.10007, 3.10009,3.09986, 3.10011, 3.10003]
r =
 Columns 1 through 9:
   3.10014   3.10010   3.10010   3.09997   3.09977   3.10010   3.10003   3.10004   3.09997
 Columns 10 through 15:
   3.09976   3.10007   3.10009   3.09986   3.10011   3.10003
```

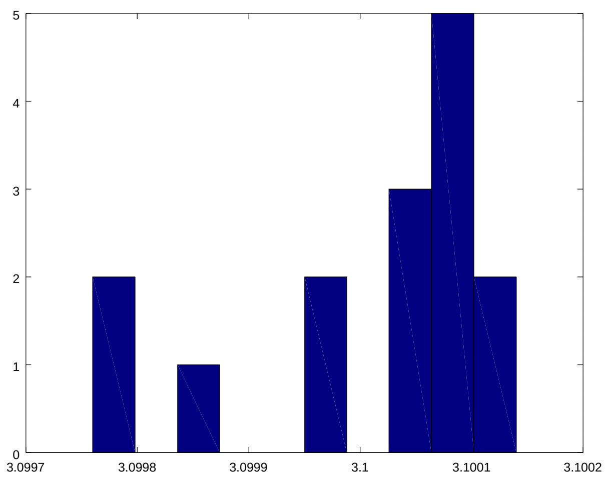


Figure 1: Simple histogram created by MATLAB

MATLAB includes built-in functions to calculate the most important statistics for a set of data, including the mean, median, mode, and variance.

```
>> mean(r)
ans =  3.10001
>> median(r)
ans =  3.10004
>> mode(r)
ans =  3.10010
>> var(r)
ans =  0.0000000149067
>> sqrt(var(r))
ans =  0.000122093
```

We can also easily create a histogram of the observed data using `histo(r)` with the output shown in Fig. 1.

To enter a matrix, we can use a similar form:

```
>> m = [ 1, 3, 5 ; 2, 4, 8; -1, -5, -9]

m =
  1   3   5
  2   4   8
 -1  -5  -9
```

2.2 Important mathematical functions

MATLAB contains implementations of many of the most-important mathematical functions. Besides the usual trigonometric and logarithmic functions, MATLAB contains a number of functions which are directly relevant to statistical analysis.

Function	Description	Mathmatical definition
`poisspdf(`x`,`μ`)`	Probability density function for the Poisson for the value (or vector of values) x and the mean expectation μ.	$\frac{\mu^x}{x!}e^{-\mu}$
`poisscdf(`x`,`μ`)`	Sum of the Poission probability density function from zero up to the the value x, given the mean expectation μ.	$e^{-\mu}\sum_{y=0}^{x}\frac{\mu^y}{y!}$
`exppdf(`t`,`τ`)`	Probability density function for the exponential distribution for the value t and the mean expectation τ.	$\frac{1}{\tau}e^{\frac{-t}{\tau}}$
`expcdf(`t`,`τ`)`	Integral probability density function from zero for the exponential distribution for the value t and the mean expectation τ.	$1-e^{\frac{-t}{\tau}}$
`normpdf(`x`,`μ`,`σ`)`	Probability density function for the Gaussian or "normal" distribution for the value x, the mean expectation μ, and standard deviation σ.	$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{(x-\mu)^2}{2\sigma^2}}$
`normcdf(`x`,`μ`,`σ`)`	Integral probability density function for the Gaussian distribution from $-\infty$ for the value x, the mean expectation μ, and standard deviation σ.	$\frac{1}{\sigma\sqrt{2\pi}}\int_{-\infty}^{x}e^{-\frac{(y-\mu)^2}{2\sigma^2}}\,dy$
`lognpdf(`x`,`μ`,`σ`)`	Probability density function for the log-normal distribution for the value x, the median expectation μ, and standard deviation σ.	$\frac{1}{x\sigma\sqrt{2\pi}}e^{-\frac{(\ln(x)-\mu)^2}{2\sigma^2}}$
`logncdf(`x`,`μ`,`σ`)`	Integral probability density function for the log-normal distribution from 0 for the value x, the median expectation μ, and standard deviation σ.	$\frac{1}{\sigma\sqrt{2\pi}}\int_{0}^{x}\frac{1}{y}e^{-\frac{(\ln(y)-\mu)^2}{2\sigma^2}}\,dy$

2.3 Vector/matrix operations

There is a second way to calculate the mean, which begins to show the power of MATLAB a bit more clearly. We can use the `sum()` function to sum up all the items in the data set and then use the `numel()` function to determine the number of elements in the data set.

```
>> sum(r)/numel(r)
ans =  3.10001
```

MATLAB contains a large set of useful operations on matrices which can be easily applied to datasets. A selection of these operations are given below, but more examples are available in the documentation.

Operation	Function	How it might be used
`X + Y`	Entry-by-entry addition	If `X` and `Y` are matrices, they must have the same number of rows and columns or it must be possible to expand one of the two to the size of the other by an integer multiplication on both rows and columns, which is called *broadcasting*. The simplest case of broadcasting is adding a scalar (which can be considered a 1x1 matrix) to each element of a matrix, but it can also be used in other situations.
`X - Y`	Entry-by-entry subtraction	
`X * Y`	Matrix multiplication	The usual rules on the relationship of the rows and columnes of `X` and `Y` apply. The concept of broadcasting applies here as well, which allows scaling a matrix.
`X .* Y`	Entry-by-entry multiplication	Each entry of `X` is multiplied by the matching entry in `Y`. This operator is very useful when constructing operations on a dataset within a sum.
`X ./ Y`	Entry-by-entry division	Each entry of `X` is divided by the matching entry in `Y`.
`X .^ Y`	Entry-by-entry power	Each entry of `X` is raised to the power of the matching entry in `Y`. This operator is very often used in broadcast mode, where `Y` is a scalar.
`inv(X)`	Array inversion	Calculate the inverse of the matrix `X`
`X > a`	Comparison operator	Each entry of `X` is compared with the value `a` to create a new matrix. The entry in the new matrix is 1 if the comparison is true, and zero otherwise. This function can be used along with `sum()` to count the number of elements in `X` which meet the requirement.

There are also a number of useful tools for creating and manipulating matrices.

- *To create a constant vector* we can use the `ones(rows,cols)` function.

```
>> ones(1,10)
ans =
   1   1   1   1   1   1   1   1   1   1
>> 5.3*ones(2,8)
ans =
   5.3   5.3   5.3   5.3   5.3   5.3   5.3   5.3
   5.3   5.3   5.3   5.3   5.3   5.3   5.3   5.3
```

- *To create identity matrix* we can use the `eye(size)` function.

```
>> eye(3)
ans =

Diagonal Matrix

   1   0   0
   0   1   0
   0   0   1
```

- *To create vectors with a linear sequence of values*, we can use **range expressions**. The commands for a range expression is `[ a : b : c]` or `[ a : c]`. In these expressions, `a` is the first value to appear in the vector and `c` is the value not to be exceeded in the vector (often the largest value in the vector). If `b` is provided, that is the step size between elements, but it if is not provided then a step size of 1 will be assumed.

```
>> vec1=[7:11]
  vec1 =
    7    8    9   10   11
>> vec2=[2:0.2:4]
  vec2 =
  2.0000   2.2000   2.4000   2.6000   2.8000   3.0000   3.2000
  3.4000   3.6000   3.8000   4.0000
```

- *To extract a row, column, or submatrix* we can use **index expressions**.
  - *To extract just row 3 from the matrix `mymatrix`, taking all columns* use `mymatrix(3,:)`.
  - *To extract just column 2 from the matrix `mymatrix`, taking all rows* use `mymatrix(:,2)`.
  - *To extract the first half of elements from the vector `myvec`* use `myvec(1:end/2)`.
  - There are many other clever uses of index expressions which are discussed in the MATLAB documentation.

2.4 An example: the weighted mean

The entry-by-entry operations allow the conversion of complex sums of productions into simply-expressed MATLAB constructs which are similar in structure to the form which would be used in standard pencil-and-paper algebra. As an example, consider the weighted average as derived and the temperature dataset from the statistics notes. It is generally simplest to enter the values and uncertainties as separate vectors.

$$310.30 \pm 0.20\ \mathrm{K} \quad 310.70 \pm 0.50\ \mathrm{K} \quad 311.8 \pm 2.0\ \mathrm{K}$$

```
>> t=[310.30, 310.70, 311.8 ]
>> et = [0.20, 0.50, 2.0]
```

The expressions for the weighted mean value and the uncertainty on the weighted mean were:

$$\bar{x} = \frac{\sum \frac{x_i}{\sigma_i^2}}{\sum \frac{1}{\sigma_i^2}}$$

$$\sigma_{\bar{x}}^2 = \frac{1}{\sum \frac{1}{\sigma_i^2}}$$

We notice that the term $\frac{1}{\sigma_i^2}$ appears multiple times in these expressions. We can easily calculate the "inverse of the error on the temperature squared" (`etsi`) using MATLAB

```
>> etsi = 1 ./ (et .^ 2)
etsi =
   25.00000    4.00000    0.25000
```

We can then proceed with calculating the weighted mean of the temperatures (`tbar`) and the error on weighted mean temperature (`etbar`):

```
>> tbar = sum(t .* etsi)/sum(etsi)
tbar =  310.37
>> etbar = 1 / sqrt(sum(etsi))
etbar =  0.18490
```

Applying our standard rules on significant figures, we find $\bar{T} = 310.37 \pm 0.18$ K.

2.5 Loading Data from a Spreadsheet

During the process of data-taking, it is often very convenient to use a spreadsheet to record the data which is taken. Octave and MATLAB can support loading data from many spreadsheet file formats, but the most stable and reproduable technique is to save your data in comma-separated-value (CSV) format before trying to load it. This manual will use CSV files and the standard function `dlmread()` exclusively. Other loading techniques exist, but you are suggested to read the documentation carefully and your mileage may vary.

The `dlmread()` function generally is used with four arguments. The first argument is the name of the file to read. The second argument is the character which is used in the file to separate values. In most CSV files, this character will be the comma (',') but sometimes a CSV file uses a space or tab to separate the individual fields. If you have a choice in your spreadsheet program when exporting your data, using a comma is the simplest choice. The third and fourth arguments are the number of rows and columns in the file to skip. The ability to skip rows is particularly important if your CSV file has headers – text labels which help determine the meaning of each column. These headers are critical for understanding your spreadsheet, but MATLAB cannot use them directly so they need to be skipped while loading the file.

For example, a CSV file of the lead conductivity data from the statistics notes might look the following:

```
Pressure,Conductivity,Conductivity_Error
2000,1165.6,3.1
4000,1205.0,2.5
6000,1250.0,2.5
8000,1271.3,6.3
10000,1312.5,8.8
11900,1347.5,2.5
```

We can load this data into MATLAB using the `dlmread` command.

```
>> leaddata=dlmread('lead_conductivity.csv',',',1,0)
leaddata =
   2.0000e+03   1.1656e+03   3.1000e+00
   4.0000e+03   1.2050e+03   2.5000e+00
   6.0000e+03   1.2500e+03   2.5000e+00
   8.0000e+03   1.2713e+03   6.3000e+00
   1.0000e+04   1.3125e+03   8.8000e+00
   1.1900e+04   1.3475e+03   2.5000e+00
```

Notice that all the data has been loaded into a single table or matrix. Depending on exactly which technique is used to load the file, `leaddata` could be a table or a matrix. We can convert the table into a matrix using the function `table2array()`.

To separate out the individual columns of data and assign them to different variables, we can use index expressions. *Remember, if the loading process produced a table rather than a matrix, you'll need to use table2array!*

```
>> press=leaddata(:,1)
press=
    2000
    4000
    6000
    8000
   10000
   11900

>> cond=leaddata(:,2)
cond =
   1165.6
   1205.0
   1250.0
   1271.3
   1312.5
   1347.5

>> conderr=leaddata(:,3)
conderr =
   3.1000
   2.5000
   2.5000
   6.3000
   8.8000
   2.5000
```

3 Linear fitting

3.1 A linear fit based on the matrix technique

Octave and MATLAB have some built-in functionalities for linear fitting, but using the uncertainties of the data can be somewhat complex within this process. It is simple-enough to define a function which will carry out the desired fit using the techniques described in the statistics manual.

In the manual, we defined the following "sum-symbols":

$$S_x = \sum \frac{x_i}{\sigma_i^2} \quad S_{xx} = \sum \tfrac{x_i^2}{\sigma_i^2} \quad S_{xy} = \sum \frac{x_i y_i}{\sigma_i^2}$$
$$S_y = \sum \frac{y_i}{\sigma_i^2} \quad S = \sum \tfrac{1}{\sigma_i^2}$$

Given those symbols, we could write down the fit results and uncertainties in a compact manner:

$$\begin{aligned}
\Delta &= S_{xx}S - S_x^2 \\
a &= \frac{[S_{xx}S_y - S_x S_{xy}]}{\Delta} \\
b &= \frac{[S S_{xy} - S_x S_y]}{\Delta} \\
\sigma_a^2 &= \frac{S_{xx}}{\Delta} \\
\sigma_b^2 &= \frac{S}{\Delta}
\end{aligned}$$

We can implement this into a fitting function this way:

```
function rval = myfit(x,y,ey)
  sx = sum(x ./ (ey .^ 2) );
  sy = sum(y ./ (ey .^ 2) );
  sxx = sum((x .* x) ./ (ey .^ 2) );
  sxy = sum((x .* y) ./ (ey .^ 2) );
  s = sum(1 ./ (ey .^ 2) );
  delta=sxx*s-sx*sx;
  a=(sxx*sy-sx*sxy)/delta;
  ea=sqrt(sxx/delta);
  b=(s*sxy-sx*sy)/delta;
  eb=sqrt(s/delta);
  rval=[ a, b ; ea, eb ];
```

If we have loaded the data from the spreadsheet as shown in Section 2.5, we can carry out the fit quite simply:

```
>> fitres = myfit(press, cond, conderr)
fitres =
   1.1339e+03   1.8112e-02
   2.6205e+00   3.5294e-04
```

The first row of `fitres` contains the fit values of a and b while the second row contains the uncertainties σ_a and σ_b.

Given these values, we can apply the formula $y'_i = a + bx_i$ to obtain the value of conductivity predicted by the model at each point:

```
>> fitcond = fitres(1,1) + press .* fitres(1,2)
fitcond =
   1170.1
   1206.3
   1242.5
   1278.7
   1315.0
   1349.4
```

We can then easily calculate the χ and χ^2 values, both point and point and the total χ^2.

```
>> chi = (cond - fitcond) ./ (conderr)
chi =
  -1.44521
  -0.52128
   2.98950
  -1.18243
  -0.28095
  -0.75370

>> chi2 = chi .^ 2
chi2 =
   2.088627
   0.271731
   8.937116
   1.398137
   0.078932
   0.568062

>> sum(chi2)
ans =  13.343
>> sum(chi2) ./ (numel(chi2)-2)
ans =  3.3357
```

3.2 Using the built-in fitting with WEIGHTS

The default `fit` function in MATLAB does not require uncertainties. Instead, the function assumes that the uncertainties can be "inferred" from the data values themselves. Often, however, we have better information about the uncertainties or data quality than is immediately observable from the data. We can provide this information to the fitting process using weights. When a weight vector is provided, the fit minimizes the quantity

$$\sum_i w_i \left(y_i - f(x_i, \vec{p})\right)^2$$

As in the case of the weighted mean, the appropriate weights are the reciprocal of the squary of the uncertainty of the given measurement, so that

$$w_i = \frac{1}{\sigma_i^2}$$

which converts the minimization function into our familiar χ^2.

$$\sum \frac{(y_i - f(x_i, \vec{p}))^2}{\sigma_i^2}$$

To achieve this with our example above, we would:

```
>> fitw = 1 ./ (conderr.*conderr);
>> fitobj = fit(press, cond, 'poly1', 'Weights', fitw)
fitobj =

     Linear model Poly1:
     fitobj(x) = p1*x + p2
     Coefficients (with 95% confidence bounds):
       p1 =     0.01811  (0.01632, 0.0199)
       p2 =        1134  (1121, 1147)
```

The return value `fitobj` contains many important pieces of information from the fit which can be extracted using other functions.

To extract the fit values of the parameters, we use the function `coeffvalues`:

```
>> params=coeffvalues(fitobj)
params =
         0.018032822122932           1132.72079217486
```

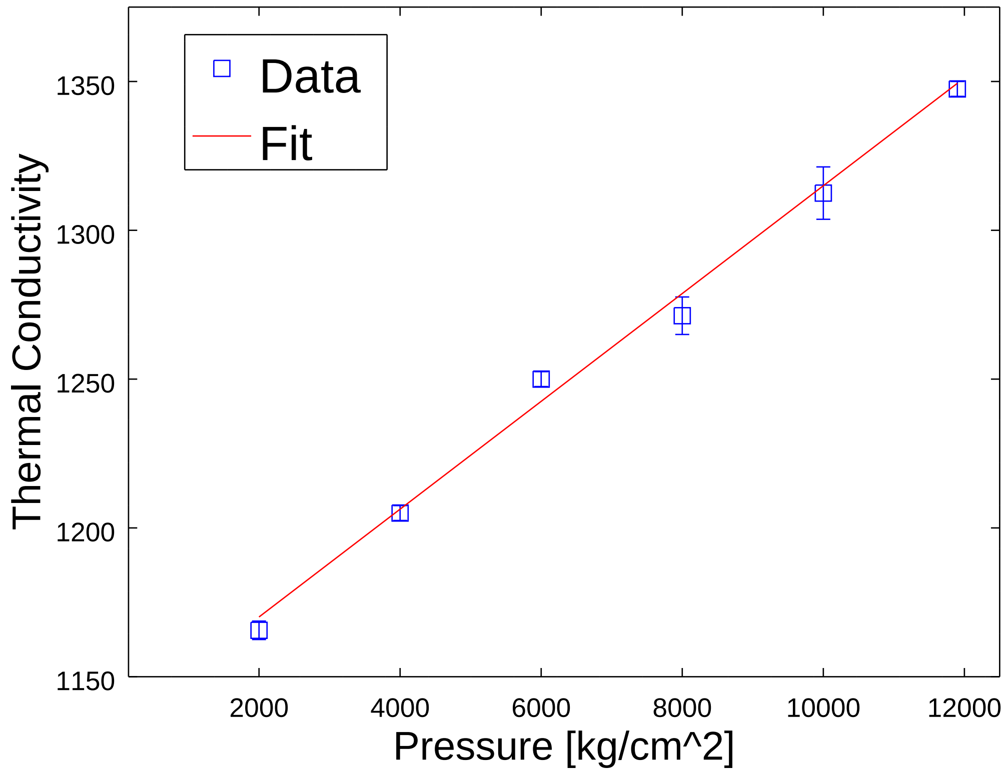
To extract the uncertainties on the fit values of the parameters, we use the function `confint` which requires a *confidence interval* arguement. Typically, we use a 68% confidence interval when we express the result of a fit in the form $p \pm \sigma_p$. However, you may wish to consider broader confidence intervals (90%, 95% or 98%) for specific purposes.

```
>> paramerrs=confint(fitobj,0.68)
paramerrs =
        0.0172685274345353           1126.78736751573
        0.0187971168113288           1138.65421683398
```

4 Plotting

Octave and MATLAB have powerful plotting capabilities. You are strongly encouraged to look into the documentation for the packages when you have a specific plotting challenge to address. We will look at a few specific plotting examples here. A plot will be shown, along with the script used to produce it, and each line of the script will be explained. You are *strongly* recommended to use scripting when creating plots. There are many steps to creating a good plot, including setting the axes, adjusting color, setting margins, and it is much easier to achieve a good result by writing your plotting commands separately and *saving them* for adjustment, reuse, or as an example for the future.

4.1 Data and fit plot



This example uses the data, fit, and fit vector from Section 3.

```
[1]   errorbar(press,cond,conderr,'s','DisplayName','Data')
[1']   errorbar(press,cond,conderr,'s;Data')
[2]   xlabel('Pressure [kg/cm^2]','fontsize',15)
[3]   ylabel('Thermal Conductivity','fontsize',15)
[4]   axis([150, 12500, 1150, 1375])
[5]   hold on
[6]   plot(press,fitcond,'r','DisplayName','Fit')
[6']   plot(press,fitcond,'r;Fit')
[7]   hand=legend('location','northwest')
[8]   set(hand,'fontsize',18)
[9]   set(hand,'position',[0.18,0.73,0.18,0.16])
[10]  hold off
```

[1] Creates the basic plot using the x values from the vector `press`, the y values from `cond`, and the σ_y values from `conderr`. More advanced versions of `errorbar` allow for errors on the x variable or for *asymmetric errors*, different errors in the positive and negative direction for either axis.

In the options string `s`, `s` indicates that squares should be used as markers. Alternatives include `x` for a cross, `o` for a circle, `d` for a diamond and so on. A character can also be given here to define the color for the markers with options given in the documentation.

For MATLAB, we set the name of the data series by specifying the parameter 'DisplayName' to indicate the name of the data series is given so that it can be labelled clearly in the legend box.

[1'] For Octave, the name of the data series can be given in the options string, separated by a semicolon from the formatting information.

[2] Sets the label of the horizontal axis and also adjusts the font size to be somewhat larger and easier to read.

[3] Sets the label of the vertical axis and also adjusts the font size to be somewhat larger and easier to read.

[4] Adjusts the limits of the horizontal axis to be 150 to 12500 and the vertical axis to be 1150 to 1374.

[5] This command causes the next plots to occur on the same pane, rather than clearing the pane and drawing a new set of axes.

[6/6'] Creates the fit line curve using the x values from the vector `press` and the y values from `fitcond`. In the options string `r`, `r` indicates that the line should be red.

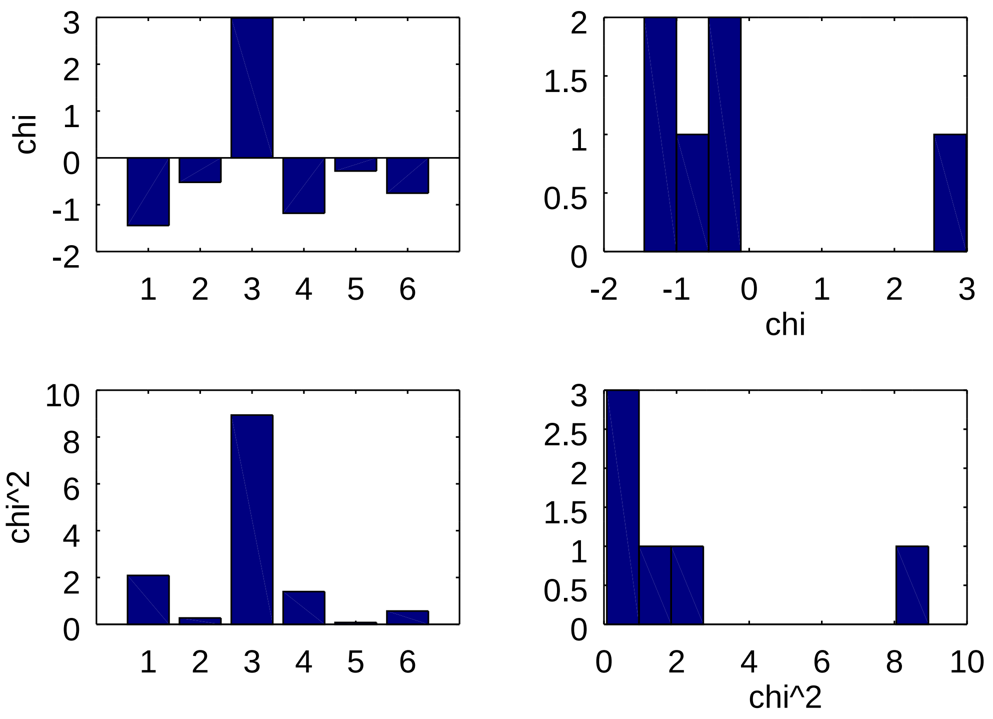
[7] Moves the legend to the northwest corner of plot and obtains a handle `hand` which can be used to adjust other characteristics of the legend.

[8] Uses the handle to make the font larger for the legend.

[9] Uses the handle to shift the location of legend away from the axis labels to avoid clipping the tick markers on the edge of the axis.

[10] This command releases the hold so that any subsequent plot would be drawn separately.

4.2 χ and χ^2 Plots



This example uses the χ (`chi`) and χ^2 (`chi2`) vectors from Section 3.

```
[1]   subplot(2,2,1)
[2]   bar(chi)
      ylabel('chi')
[3]   subplot(2,2,2)
[4]   hist(chi)
      xlabel('chi')
      subplot(2,2,3)
      bar(chi2)
      ylabel('chi^2')
      subplot(2,2,4)
      hist(chi2)
      xlabel('chi^2')
```

[1] Divides the canvas into space for four plots, two vrtrically and two horizontally and makes the upper left plot the active plotting area.

[2] Makes a bar plot of the χ vector. This sort of plot can be used to understand if there is a systematic trend in the disagreement between the data and the model used in the fit.

[3] Changes the active plot to be the upper right area.

[4] Makes a histogram of the χ vector. This is a called a *pull* plot and it should show a Gaussian distribution centered at zero with a sigma of 1 for a good quality of model and data.

4.3 Saving a Plot as a Figure

When creating a scientific paper, it is often helpful to save your figure in PDF format. The PDF format, as a vector file format, is very efficient for normal plots, and allows the reader to zoom deeply into the figure if needed without the view becoming grainy. For 2d-surface plots, a bitmap format such as JPG or PNG may be appropriate, but the PDF should be used for most figures.

It can be a bit tricky to get a good-quality output from MATLAB with appropriate margins. The script below may be helpful in this instance.

```
function printpdf(outfilename)
  set(gcf, 'PaperUnits','centimeters');
  set(gcf, 'Units','centimeters');
  pos=get(gcf,'Position');
  set(gcf, 'PaperSize', [pos(3) pos(4)]);
  set(gcf, 'PaperPositionMode', 'manual');
  set(gcf, 'PaperPosition',[0 0 pos(3) pos(4)]);
  print('-dpdf',outfilename);
```

5 Power Tools

5.1 Scripts

When working with a powerful system like MATLAB, it is often very helpful to collect the commands for a given data manipulation or for creating a specific plot into a script – a set of commands which are stored as a text file and can be edited and re-run until the desired result is achieved. Recent versions of Octave and MATLAB contain built-in editors which can be used to create scripts.

We will use the Live Script capability of MATLAB extensively. If you have an older version of MATLAB installed, you may need to upgrade it for Live Scripts to be available.

5.2 Loops

6 Random Numbers and Monte Carlo Techniques

6.1 Random Number Generators

Octave and MATLAB each contain a wealth of random number generators, using a uniform random number generator which implements the Mersine Twister algorithm. The generators can be used to produce vectors or matrices of random numbers (or random deviates, as they are sometimes called).

- `rand(R,C)` – Produces a matrix with `R` rows and `C` columns containing random values uniformly distributed between zero and one.
- `unifrnd(A,B,R,C)` – Produces a matrix with `R` rows and `C` columns containing random values uniformly distributed between `A` and `B`.
- `poissrnd(MU,R,C)` – Produces a matrix with `R` rows and `C` columns containing random values drawn from a Poisson distribution with expected value `MU`.
- `exprnd(TAU,R,C)` – Produces a matrix with `R` rows and `C` columns containing random values drawn from an exponential distribution with time constant `TAU`.
- `normrnd(MU,SIGMA,R,C)` – Produces a matrix with `R` rows and `C` columns containing random values drawn from a Gaussian (or normal) distribution with expected value `MU` and standard deviation `SIGMA`.
- `lognormrnd(MU,SIGMA,R,C)` – Produces a matrix with `R` rows and `C` columns containing random values drawn from log-normal distribution with log-median `MU` and fractional width `SIGMA`.

7 Non-linear fitting

7.1 Grid search fitting

```
function results=gridSearch(x,y,ey,a0,delta,func,steps)
  terminationChidelta=0.001; % Set the minimum chi2 decrease for terminating the fit
  ai=a0; % seed with the initial point
  % create a matrix with columns for each up/down variation
  deltam=[diag(delta), diag(-1*delta)];

  for step = 1:steps
    % find our current chi2
    chi0=sum(((y - func(x,ai))./ ey) .^2);
    % Print out our current position
    disp(sprintf('Step %3d : %f  (%f %f)',step,chi0,ai(1),ai(2)));

    % find the chi2 for each possible step (each column in deltam)
    chidelta=zeros(1,2*numel(a0));
    for iv=1:numel(chidelta)
      atest=ai + deltam(:,iv)';
      chidelta(iv)=sum(((y - func(x,atest))./ ey) .^2) - chi0;
    end

    % choose the best direction or decrease the step size
    if (min(chidelta)>0)
      % If no current step direction decreases chi2, halve the step size
      deltam = deltam ./ 2;
    else
      % Determine which step direction was the best
      [v,where]=min(chidelta);
      % Update the parameters vector to the new point
      ai=ai + deltam(:,where)';
      % If the improvement was too small, stop the process
      if (abs(v)< terminationChidelta)
        break;
      elseif (abs(v)>0.1)
        % record the delta matrix until the change in chi2 becomes
        % small.  We need this later to find parameter errors
        deltasmall=deltam;
      end
    end
  end

  % Now we search for the uncertainties on the parameters
  % using the delta(chi2)=1 requirement

  % First, we make an empty vector of errors
  errs=zeros(1,2*numel(a0));
  % We loop over the individual parameters, considering + and - variations
  for iv=1:numel(delta)*2
    dv=deltasmall(:,iv)';
    chid=0;
    aie=ai;
    % Search for when the chi2 increases by more than one unit
    while (chid<1)
      aie=aie+dv;
      chil=chid;
      chid=sum(((y - func(x,aie))./ ey) .^2) - chi0;
    end
    % Get the parameter associated with a given variation
    index=mod(iv-1,numel(a0))+1;
    % Interpolate between the last two points visited to estimate
    % where the chi2 increased by 1
    errs(iv)=(aie(index)-ai(index))+(1-chid)/(chil-chid)*(dv(index));
  end

  % Print out the final fit chi2
  disp(sprintf('Final chi2/dof = %f (%f/%d)',chi0/(numel(x)-numel(delta)),chi0,
                  (numel(x)-numel(delta))));
  for iv=1:numel(delta)
    % Print out the parameters and errors
    disp(sprintf('Parameter %d: %g + %g - %g',iv,ai(iv),errs(iv),
        -errs(iv+numel(delta))));
  end

  results = { ai, errs };

end
```

7.2 Using the minimizer

One of the strengths of MATLAB is the powerful set of minimizers and solvers available. We can use these directly to find a minimum in the χ^2 function. The script below, which can be used as a general tool for fitting, uses `fminunc` to minimize the χ^2. Besides the parameter values which minimize the χ^2, it also returns the Hessian matrix which (for a successful fit) contains elements $\frac{1}{\sigma_i \sigma_j}$. We use the diagonal of the Hessian matrix to obtain the errors for the parameters, but it can also be used to understand the correlation between the parameters.

```
function result=minfit1(x,y,ey,a0,func)
  chifunc=@(a) minfitchi2(x,y,ey,a,func);
  % Find the minimum point and chi2
  [fitpt,chi2,Xinfo,Xoutput,Xgrad,hessian]=fminunc(chifunc,a0);
  % Find the best chi2
  chi2=chifunc(fitpt);
  % Find the parameter uncertainties, which are 1/sqrt(hessian_diag)
  errs=1 ./ sqrt ( diag(hessian) );
  result={ chi2, fitpt, errs };
end
function chi=minfitchi2(x,y,ey,a,func)
  chi=sum( ( ( y - func(x,a) ) ./ ey ) .^ 2);
  end
```